<repository>

# easyMina

## overview

repository: https://github.com/EasyMina/easyMina/
userName: EasyMina
repository: easyMina
branch: main
date: 2023-11-10T14:03:44+01:00 (1699621424)

<file>

## path: /.npmignore

url: https://github.com/EasyMina/easyMina/blob/main/.npmignore

```
.mina/
workdir/
tsconfig.json
package-lock.json
node_modules/
package-lock.json
tsconfig.json
test/
index.mjs
```

</file>

<file>

## path: /README.md

url: https://github.com/EasyMina/easyMina/blob/main/README.md

build passing

# Easy Mina

Made for zk beginners and busy beavers

easy mina is a Node.js module that helps you create a bare-bones environment with minimal opinionated pre-configuration.

Helps you set up:
:heavy_check_mark: Environment variables and folders
:heavy_check_mark: Your smart contract workspace with a small example

:heavy_check_mark: TypeScript config file
:heavy_check_mark: Security checks to minimize the risk of security exploits.

## Quickstart

node

```
npm init -y
npm i easymina
```

index.mjs

```
import { EasyMina } from 'easymina'

const easyMina = new EasyMina()
await easyMina.setEnvironment( {
    'projectName': 'hello-world'
} )
```

## Table of Contents

## Documentation

Please visit https://easymina.github.io (https://easymina.github.io)

## Contributing

Bug reports and pull requests are welcome on GitHub at https://github.com/a6b8/easymina. This project is intended to be a safe, welcoming space for collaboration, and contributors are expected to adhere to the code of conduct (https://github.com/EasyMina/easyMina/blob/main/CODE_OF_CONDUCT.md).

## Limitations

- Currently in Alpha Stage

## Credits

- This project was supported by the zkIgnite (https://zkignite.minaprotocol.com) grant program.

## License

The module is available as open source under the terms of the Apache 2.0 (https://github.com/EasyMina/easyMina/blob/main/LICENSE).

## Code of Conduct

Everyone interacting in the EasyMina project's codebases, issue trackers, chat rooms and mailing lists is expected to follow the code of conduct (https://github.com/EasyMina/easyMina/blob/main/CODE_OF_CONDUCT.md).</file>

<file>

## path: /package.json

url: https://github.com/EasyMina/easyMina/blob/main/package.json

```
{
  "name": "easymina",
  "version": "0.1.1",
  "description": "EasyMina is a beginner-friendly Node.js module that assists you in creating a bare-bones environment for the Mina blockchain.",
  "main": "src/EasyMina.mjs",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "EasyMina",
  "license": "Apache-2.0",
  "dependencies": {
    "axios": "^1.4.0",
    "moment": "^2.29.4",
    "o1js": "^0.13.1"
  }
}
```

</file>

<file>

## path: /src/EasyMina.mjs

url: https://github.com/EasyMina/easyMina/blob/main/src/EasyMina.mjs

```
/*

2023 EasyMina

Disclaimer:
The use of this code is at your own risk. The entire code is licensed under the Apache License 2.0, which means you are granted certain rights to use, modify, and distribute the code under the terms of the license.
However, please be aware that this module was created for learning purposes and testing smart contracts.
This module is intended to provide a platform for educational and testing purposes only. It may not be suitable for use in production environments or for handling real-world financial transactions. The authors or
```

```
  Name
    EasyMina.mjs
  Description
    This class serves as a wrapper to connect the classes mentioned below. It also stores the state of
EasyMina globally.
  Tree
    config.mjs
      |--- PrintConsole.mjs
      |--- EasyMina.mjs
        |--- Account.mjs
          |--- Cryption.mjs
          |--- Faucet.mjs
            |--- PrintConsole.mjs
      |--- Credentials.mjs
      |--- SmartContract.mjs
      |--- Workspace.mjs
        |--- mixed.mjs
      |--- GraphQl.mjs
        |--- PrintConsole.mjs
  Blocks
    {
      'console': { ...config['console'] },
      'docs': { ...config['docs'] },
      'environment': { ...config['environment'] },
      'graphQl': { ...config['graphQl'] },
      'messages': { ...config['messages'] },
      'meta': { ...config['meta'] },
      'network': { ...config['network'] },
      'print': { ...config['print'] },
      'typescript': { ...config['typescript'] },
      'validations': { ...config['validations'] },
    }
  Public
    Variables
      state
      account
    Methods
      .setEnvironment( { silent=false }={} )
      .deployContract( {} )
      .health()
```

```
        .getConfig()
*/



import moment from 'moment'
import fs from 'fs'

import { GraphQl } from './interactions/GraphQl.mjs'

import { configImported } from './data/config.mjs'
import { Credentials } from './environment/Credentials.mjs'
import { Workspace } from './environment/Workspace.mjs'
import { SmartContract } from './environment/SmartContract.mjs'

import { Account } from './accounts/Account.mjs'
import { PrintConsole } from './helpers/PrintConsole.mjs'
import { findClosestString, keyPathToValue } from './helpers/mixed.mjs'


export class EasyMina {

    #printConsole
    #config
    state
    account


    constructor() {
        this.#setConfig()
        return true
    }


    #setConfig() {
        const config = configImported
        const n = [
            'deployers',
            'contracts'
        ]
            .forEach( key => {
                config['environment']['addresses'][ key ]['fullFolder'] = ''
                config['environment']['addresses'][ key ]['fullFolder'] += process.cwd() + '/'
                config['environment']['addresses'][ key ]['fullFolder'] += config['environment']['addresses']['root']
                config['environment']['addresses'][ key ]['fullFolder'] += config['environment']['addresses'][ key ]
['folder']
            } )

        const m = [
            'typescript',
            'build'
        ]
            .forEach( key => {
                let path = ''
                // path += process.cwd() + '/'
                path += config['environment']['workspace']['contracts']['root']
                path += config['environment']['workspace']['contracts'][ key ]['folder']
                config['environment']['workspace']['contracts'][ key ]['fullRelative'] = path

                config['environment']['workspace']['contracts'][ key ]['full'] = ''
                config['environment']['workspace']['contracts'][ key ]['full'] += process.cwd() + '/'
                config['environment']['workspace']['contracts'][ key ]['full'] +=
                    config['environment']['workspace']['contracts'][ key ]['fullRelative']
            } )

        config['typescript']['template']['compilerOptions']['outDir'] =
            config['environment']['workspace']['contracts']['build']['fullRelative']

        config['typescript']['template']['compilerOptions']['rootDir'] =
            config['environment']['workspace']['contracts']['typescript']['fullRelative']

        config['typescript']['template']['include'] = [
            config['environment']['workspace']['contracts']['typescript']['fullRelative']
        ]

        config['environment']['workspace']['contracts']['typescript']['fileName'] = [
            [ '{{name}}', config['meta']['name'] ],
            [ '{{splitter}}', config['environment']['addresses']['splitter'] ]
        ]
            .reduce( ( acc, a, index ) => {
                const [ search, value ] = a
                acc = acc.replace( search, value )
                return acc
            }, config['environment']['template']['source']['name'] )

        config['environment']['addresses']['deployers']['fileName'] = [
            [ '{{name}}', config['meta']['name'] ],
            [ '{{splitter}}', config['environment']['addresses']['splitter'] ],
            [ '{{unix}}', config['meta']['unix'] ]
        ]
            .reduce( ( acc, a ) => {
                const [ one, two ] = a
                acc = acc.replace( one, two )
                return acc
            }, config['environment']['addresses']['deployers']['fileNameStruct'] )

        this.#config = config

        return true
    }


    async setEnvironment( { silent=false }={} ) {
            this.#validUserInput( { 'method':'setEnvironment', 'args': arguments } )
            this.#setConfig()

            const logo = &nbsp;
         -- - - - - - - --*-- - - - - - - --
        |       ____           ____       |
        |      /\\  \\         /\\__\\      |
        |     /::\\  \\       /::|  |     |
        |    /:/\\:\\  \\     /:|:|  |     |
        |   /::\\~\\:\\  \\   /:/|:|__|__   |
        |  /:/\\:\\ \\:\\__\\ /:/ |::::\\__\\  |
        |  \\:\\~\\:\\ \\/__/ \\/__/~~/:/  /  |
        |   \\:\\ \\:\\__\\         /:/  /   |
        |    \\:\\ \\/__/        /:/  /    |
        |     \\:\\__\\         /:/  /     |
        |      \\/__/         \\/__/      |
        |                              |
         -- - -- E a s y M i n a -- - --
        | change the world with zk tech |
        ---------------------------------
    &nbsp;
            console.log( logo )
            console.log( 'PROJECT' )
            console.log( &nbsp;  Name                    ${this.#config['meta']['name']}&nbsp; )
            console.log( &nbsp;  ClassName               ${this.#config['environment']['template']['source']
    ['className']}&nbsp; )
            console.log( &nbsp;  Timestamp               ${this.#config['meta']['unix']}&nbsp; )
            console.log( 'ENVIRONMENT' )

            this.state = {
                'accounts': {
                    'deployers': [],
                    'contracts': []
                },
                'account': false
            }

            this.#printConsole = new PrintConsole()
            this.#printConsole.init( {
                'config': {
                    'print': { ...this.#config['print'] },
                    'messages': { ...this.#config['messages'] }
                }
            } )

            this.#addEnvironmentCredentials()

            await this.#addEnvironmentWorkspace()

            console.log( 'CREDENTIALS' )
            const { status, transactionHash }= await this.#addDeployers()
            if( status === 'pending' II status === 'new' ) {
                const graphQl = new GraphQl()
                const config = {
                    'network': { ...this.#config['network'] },
                    'graphQl': { ...this.#config['graphQl'] },
                    'messages': { ...this.#config['messages'] },
                    'print': { ...this.#config['print'] }
                }
                graphQl.init( { config } )

                const cmd = 'transactionByHash'
                const vars = {
                    'hash': transactionHash
                }

                await graphQl.waitForSignal( { cmd, vars } )
                console.log( &nbsp;  Faucet                    Found&nbsp; )
                process.exit( 1 )
            }


            console.log()
            return this
        }


        async deployContract( {} ) {
            console.log( 'Deploy Contract' )

            this.#validUserInput( { 'method':'deployContract', 'args': arguments } )
            this.#setConfig()

            let _accountPath = null
            if( this.account ) {
                _accountPath = this.account['state']['path']
            } else {
            }

            let [ accountPath, smartContractPath ] = this.#validUserInputTypescript( { 'accountPath': _accountPath
    } )

            if( this.account ) {
                process.stdout.write( '  Acccount             ' )
                this.account = await this.#initAccount( { 'mode': 'read', 'path': accountPath } )
                console.log( &nbsp;      ${accountPath}&nbsp; )
            }

            const config = {
                'meta': { ...this.#config['meta'] },
                'docs': { ...this.#config['docs'] },
                'environment': { ...this.#config['environment'] },
                'network': { ...this.#config['network'] },
                'console': { ...this.#config['console'] }
            }

            const smartContractClassName = this.#config['environment']['template']['source']['className']
            const smartContract = new SmartContract()
            await smartContract.init( { config, smartContractPath, smartContractClassName } )
            await smartContract.deploy( { accountPath } )

            return this
        }


        health() {
            return this.#config['meta']['easyMinaVersion']
        }


        getConfig() {
            return this.#config
        }


        #validUserInputTypescript( { accountPath=null}) {
            const messages = []
            const paths = {
                'deployerFileName': null,
                'smartContractFileName': null,
                'smartContractFileNameModule': null
            }

            if( accountPath === null ) {
                paths['deployerFileName'] = ''
                paths['deployerFileName'] += this.#config['environment']['addresses']['deployers']['fullFolder']
                paths['deployerFileName'] += this.#config['environment']['addresses']['deployers']['fileName']
            } else {
                paths['deployerFileName'] = accountPath
            }
    /*
            if( !fs.existsSync( paths['deployerFileName'] ) ) {
                messages.push( &nbsp;Path to "deployerFileName" does not exist:
    ${paths['deployerFileName']}&nbsp; )
            }
    */
            paths['smartContractFileName'] = ''
            paths['smartContractFileName'] += this.#config['environment']['workspace']['contracts']['build']['full']
            paths['smartContractFileName'] += this.#config['environment']['workspace']['contracts']['typescript']
    ['fileName']
                .replace( '.ts', '.js' )

            paths['smartContractFileNameModule'] = paths['smartContractFileName']
                .replace( '.js', '.mjs' )
            if( !fs.existsSync( paths['smartContractFileName'] ) ) {
                messages.push( &nbsp;Build path from "smartContractFileName" does not exist. Did you forget to
    run "tsc"?&nbsp; )
            }

            if( messages.length === 0 ) {
                const fileContents = fs.readFileSync( paths['smartContractFileName'] )
                fs.writeFileSync( paths['smartContractFileNameModule'], fileContents )
            }

            messages
                .forEach( ( msg, index, all ) => {
                    index === 0 ? console.log( &nbsp;  .deployContract():       Input error&nbsp; ) : ''
                    console.log( &nbsp;    - ${msg}&nbsp; )

                    if( index === all.length -1 ) {
                        let url = ''
                        url += this.#config['docs']['url']
                        url += this.#config['docs']['deployContract']
                        url += '#' + this.#config['docs']['options']

                        console.log()
                        console.log( &nbsp;  For more information visit: ${url}&nbsp;)
                    }
                } )

            if( messages.length === 0 ) {
                return [ paths['deployerFileName'], paths['smartContractFileNameModule'] ]
            } else {
                process.exit( 1 )
            }
        }


        #validUserInput( { method='', args={} } ) {
            const messages = []

            const language = 'en'
            const lookUp = Object
                .entries( this.#config['validations']['keyPaths'] )
                .reduce( ( acc, a, index ) => {
                    const [ key, value ] = a
                    value['methods']
                        .forEach( method => {
                            !Object.hasOwn( acc, method ) ? acc[ method ] = {} : ''

                            const niceName = value['userPath'][ language ]
                            acc[ method ][ niceName ] = key
                        } )
                    return acc
                }, {} )
            const methodValid = Object
                .keys( lookUp )
                .includes( method )

            if( !methodValid ) {
                messages.push( &nbsp;Method: "${method}" is not valid&nbsp;)
            } else {
                Object
                .entries( args[ '0' ] )
                .forEach( a => {
                    const [ key, value ] = a

                    const validate = {
                        'key': null,
                        'value': false
                    }

                    validate['key'] = Object.hasOwn( lookUp[ method ], key )

                    if( validate['key'] ) {
                        const keyPath = lookUp[ method ][ key ]
                        const regexPath = this.#config['validations']['keyPaths'][ keyPath ]['validation']
                        const regex = keyPathToValue( { 'data': this.#config, 'keyPath':
    &nbsp;validations__regexs__${regexPath}&nbsp; } )
                        validate['value'] = value.match( regex['regex'] )
                        if( validate['value'] === null ) {
                            messages.push( &nbsp;"${key}": ${regex['message'][ language] }&nbsp; )
                        } else {
                            const keys = lookUp[ method ][ key ].split( '__' )
                            switch( keys.length ) {
                                case 1:
                                    this.#config[ keys[ 0 ] ] = value
                                    break
                                case 2:
                                    this.#config[ keys[ 0 ] ][ keys[ 1 ] ] = value
                                    break
                                case 3:
                                    this.#config[ keys[ 0 ] ][ keys[ 1 ] ][ keys[ 2 ] ] = value
                                    break
                                case 4:
                                    this.#config[ keys[ 0 ] ][ keys[ 1 ] ][ keys[ 2 ] ][ keys[ 3 ] ] = value
                                    break
                                case 5:
                                    this.#config[ keys[ 0 ] ][ keys[ 1 ] ][ keys[ 2 ] ][ keys[ 3 ] ][ keys[ 4 ] ] = value
                                    break
                                default:
                                    console.log( 'Key Length not found' )
                                    process.exit( 1 )
                                    break
                            }
                        }
                    } else {
                        const nearest = findClosestString( { 'input': key, 'keys': Object.keys( lookUp[ method ] ) } )
                        messages.push( &nbsp;"${key}": Key is unknown. Did you mean: ${nearest}?&nbsp; )
                    }
                } )
            }

            messages.forEach( ( msg, index, all ) => {
                index === 0 ? console.log( &nbsp;.${method}(): Input validation error&nbsp; ) : ''
                console.log( &nbsp;  - ${msg}&nbsp; )

                if( index === all.length -1 ) {

                    let url = ''
                    url += this.#config['docs']['url']
                    url += this.#config['docs'][ method ]
                    url += '#' + this.#config['docs']['options']

                    console.log( &nbsp;  For more information visit: ${url}&nbsp;)
                }
            } )

            if( messages.length === 0 ) {
                return true
            } else {
                process.exit( 1 )
            }
        }


        async #addEnvironmentWorkspace() {
            // console.log( '  Workspace' )

            const config = {
                'meta': { ...this.#config['meta'] },
                'environment': { ...this.#config['environment'] },
                'typescript': { ...this.#config['typescript'] },
                'validations': { ...this.#config['validations'] }
            }

            const workspace = new Workspace()
            workspace.init( { 'config': { ...config } } )
            await workspace.start()

            return true
        }


        #addEnvironmentCredentials() {
            // console.log( '  Credentials' )
            const credentials = new Credentials()

            credentials
```

```
        .init( { 'config': this.#config['environment']['addresses'] } )

    credentials
        .checkEnvironment()

    this.state['addresses'] = credentials
        .checkAccounts()

    return true
}


async #deployersValidate() {
    const availableDeployers = this.state['addresses']['deployers']
        .filter( a => a['name'] === this.#config['meta']['name'] )
        .filter( a => a.hasOwnProperty( 'unix' ) )
        .map( a => {
            a['unix'] = parseInt( a['unix'] )
            return a
        } )
        .sort( ( a, b ) => a['unix'] - b['unix'] )

    const deployerAccounts = await Promise.all(
        availableDeployers
            .map( async( item, index ) => {
                const { path } = item
                const account = await this.#initAccount( { 'mode': 'read', path } )
                return account
            } )
    )

    const valids = deployerAccounts
        .filter( a => a.state['valid'] )

    return valids
}


async #deployerAccounts( { valids } ) {
    const accounts = await Promise.all(
        valids
            .map( async( account ) => {
                await account.fetchAccount()
                return account
            } )
    )

    let now = moment()
    const list = accounts
        .map( ( a, index ) => {
            const result = {
                'index': index,
                'name': a.content['meta']['fileName'],
                'balance': a.state['balance']['balance'],
                'transactionsLeft': a.state['balance']['transactionsLeft'],
                'useable': null
            }

            result['useable'] = result['balance'] === null ? false : true
            result['faucets'] = a.content['data']['faucets']
                .map( ( b, index ) => {
                    const network = this.#config['network'][ this.#config['network']['use'] ]['faucet']['network']
                    const dateFromTimestamp = moment.unix( b['timestamp'] )
                    let now = moment()
                    let differenceInMinutes = now.diff(dateFromTimestamp, 'minutes' )
                    const result = {
                        'requestedInMinutes': differenceInMinutes,
                        'network': b['network'] === network
                    }

                    return result
                } )

            return result
        } )

    const readyToUse = list
        .filter( a => a['useable'] )
        .sort( ( a, b ) => b['transactionsLeft'] - a['transactionsLeft'] )
        .filter( a => a['transactionsLeft'] > 0 && a['useable'] )

    const faucetPending = list
        .filter( a => {
            const one = a['faucets']
                .some( b => b['network'] && ( b['requestedInMinutes'] < 11 ) && true )
            const two = !a['useable']
            return one && two
        } )

    return [ readyToUse, faucetPending, accounts ]
}


async #initAccount( { mode, path } ) {
    const secret = 'abc'
    const config = {
        'meta': { ...this.#config['meta'] },
        'network': { ...this.#config['network'] },
        'console': { ...this.#config['console'] },
        'graphQl': { ...this.#config['graphQl'] },
        'console': { ...this.#config['console'] },
        'print': { ...this.#config['print'] },
        'messages': { ...this.#config['messages'] },
        'environment': { ...this.#config['environment'] }
        }

        const newAccount = new Account()
        await newAccount.init(  { secret, config } )
        const currentPath = ( mode === 'new' ) ? await newAccount.createDeployer() : path
        newAccount.readDeployer( { 'path': currentPath } )

        return newAccount
    }


    async #deployerSelectAccount( { readyToUse, faucetPending, accounts } ) {
        let modes = [ 'known', 'pending', 'new' ]
        let status = null

        if( readyToUse.length > 0 ) {
            status = 'known'
        } else if( faucetPending.length > 0 ) {
            status = 'pending'
        } else {
            status = 'new'
        }

        let chooseAccount
        let newAccount
        let graphQl
        let transactionHash

        if( status === 'new' ) {
            // console.log( '      Create Account' )
            newAccount = await this.#initAccount( { 'mode': 'new' } )
        }

        switch( status ) {
            case 'known':
                // console.log( '      Use funded account' )
                chooseAccount = accounts[ readyToUse[ 0 ]['index'] ]
                break
            case 'pending':
                // console.log( '      Wait for pending faucet' )

                const faucet1 = accounts[ faucetPending[ 0 ]['index'] ]['content']['data']['faucets']
                    .find( a => a['network'] === this.#config['network'][ this.#config['network']['use'] ]['faucet']
['network'] )
                transactionHash = faucet1['transaction']
                chooseAccount = accounts[ faucetPending[ 0 ]['index'] ]
                break
            case 'new':
                const faucet2 = newAccount['content']['data']['faucets']
                    .find( a => a['network'] === this.#config['network'][ this.#config['network']['use'] ]['faucet']
['network'] )
```

```
            transactionHash = faucet2['transaction']
            chooseAccount = newAccount
            break
        default:
            console.log( 'Status not known' )
            process.exit( 1 )
            break
    }

/*
    if( status !== 'known' ) {
        const graphQl = new GraphQl()
        const config = {
            'network': { ...this.#config['network'] },
            'graphQl': { ...this.#config['graphQl'] },
            'messages': { ...this.#config['messages'] },
            'print': { ...this.#config['print'] }
        }
        graphQl.init( { config } )

        const cmd = 'transactionByHash'
        const vars = {
            'hash': transactionHash
        }

        await graphQl.waitForSignal( { cmd, vars } )

        if( status === 'new' ) {
            await newAccount.fetchAccount()
            chooseAccount = newAccount
        }
    }
*/

    return [ chooseAccount, status, transactionHash ]
  }


  async #addDeployers() {
    // process.stdout.write( '  Overall              ' )
    const valids = await this.#deployersValidate()
    const [ readyToUse, faucetPending, accounts ] = await
        this.#deployerAccounts( { valids } )

    const [ account, status, transactionHash ] = await this.#deployerSelectAccount( { readyToUse,
faucetPending, accounts } )
    this.account = account

    // process.stdout.write( '  Status               ' )

    console.log( '  Accounts ' )
    const m = [
        [ 'Funded', readyToUse.length, 'new' ],
        [ 'Pending', faucetPending.length, 'pending' ],
        [ 'Empty', ( valids.length - readyToUse.length ) - faucetPending.length, 'empty' ]
    ]
        .forEach( ( a, index, all ) => {
            let msg = ''
            if( index === all.length - 1 ) {
                msg += '  |     └──── '
            } else {
                msg += '  |     ├──── '
            }
            msg += &nbsp;${a[ 0 ]} (${a[ 1 ]})&nbsp;

            console.log( msg )
        } )

    console.log( &nbsp;  └──── ${this.account['content']['data']['address']['public']} (${status})&nbsp; )

    let use = this.#config['network']['use']
    let url = ''
    url += this.#config['network'][ use ]['explorer']['wallet']
    url += this.account['content']['data']['address']['public']

    const n = [
        [ 'File', &nbsp;${this.account.content['meta']['fileName']}&nbsp; ],
        [ 'Explorer', url ],
    ]

    transactionHash ? n.push( [ 'Transaction', transactionHash ] ) : ''

    n
        .forEach( ( a, index, all ) => {
            let msg = ''
            if( index === all.length - 1 ) {
                msg += '      └──── '
            } else {
                msg += '      ├──── '
            }
            msg += &nbsp;${a[ 1 ]}&nbsp;
            console.log( msg )
        } )

/*
    let use = this.#config['network']['use']
    let url = ''
    url += this.#config['network'][ use ]['explorer']['wallet']
    url += this.account['content']['data']['address']['public']



        .join( ', ' )

    console.log( &nbsp;     ${msg}&nbsp; )
```

```
*/

    //  process.stdout.write( '  Selection             ' )
/*
       this.#deployerSelectAccount( { readyToUse, faucetPending, accounts } )
*/


/*
       msg = ''
       msg += '  Choose                '
       msg += '      '
       msg += &nbsp;${this.account.content['meta']['fileName']}&nbsp;
       console.log( msg )

       let _public = ''
       _public += '  Public Key            '
       _public += '      '
       _public += this.account['content']['data']['address']['public']
       console.log( &nbsp;${_public}&nbsp; )

       let use = this.#config['network']['use']
       let url = ''
       url += '  Explorer              '
       url += '      '
       url += this.#config['network'][ use ]['explorer']['wallet']
       url += this.account['content']['data']['address']['public']
       console.log( &nbsp;${url}&nbsp; )

       let msg1 = ''
       msg1 += '  Balance               '
       msg1 += '      '
       msg1 += Object
           .entries( this.account.state['balance'] )
           .map( a => &nbsp;${a[ 0 ]}: ${a[ 1 ]}&nbsp; )
           .join( ', ' )

       console.log( msg1 )
*/

       return { status, transactionHash }
    }
}
```

</file>

<file>

## path: /src/accounts/Account.mjs

url: https://github.com/EasyMina/easyMina/blob/main/src/accounts/Account.mjs

```
/*
```

```
    Name
        Account.js

    Description
        Create a Mina Account and fetch account balance.

    Blocks
        {
            'console': { ...config['console'] },
            'environment': { ...config['environment'] }
            'graphQl': { ...config['graphQl'] },
            'messages': { ...config['messages'] },
            'meta': { ...config['meta'] },
            'network': { ...config['network'] },
            'print': { ...config['print'] },
        }

    Public:
        Variables
        Methods
            .init( { secret, config } ) (async)
            .readDeployer( { path } )
            .createDeployer()
            .fetchAccount()
*/


import fs from 'fs'
import { Cryption } from './Cryption.mjs'
import { Faucet } from './Faucet.mjs'


export class Account {
    #cryption
    #snarkyjs
    #validation


    constructor() {
        this.config = {}

        this.#cryption
        this.#snarkyjs
        this.#validation

        this.state
        this.content

        this.#cryption = new Cryption()
    }
```

```
async init( { secret, config } ) {
    this.state = {
        'path': null,
        'snarkyIsReady': false,
        'valid': false,
        'useable': false,
        'faucet': false,
        'balance': {}
    }

    this.config = { ...config }

    this.#cryption.init( { secret } )
    this.content = {}

    if( !this.state['snarkyIsReady'] ) {
        await this.#addSnarkyjs()
    }

    return true
}


readDeployer( { path } ) {
    this.state['path'] = path
    this.#checkValidation()

    return true
}


async createDeployer() {
    const struct = this.#newAccountStruct( { 'type': 'deployer' } )
    const receiver = struct['data']['address']['public']
    const faucetResponse = await this.#requestFaucet( { receiver } )

    struct['data']['faucets'].push( faucetResponse['faucet'] )
    this.content = struct

    let path = ''
    path += this.config['environment']['addresses']['deployers']['fullFolder']
    path += struct['meta']['fileName']

    if( !fs.existsSync( path ) ) {
        fs.writeFileSync(
            path,
            JSON.stringify( struct, null, 4 ),
            'utf-8'
        )
    } else {
        console.log( 'Deployer File already exists something went wrong!' )
        process.exit( 1 )
        }

        return path
    }


    async fetchAccount() {
        const status = await this.#accountStatus()
        await this.#accountBalance( { status } )
        return true
    }


    #checkValidation() {
        this.#validation = {
            'validJson': false,
            'encryptionActivated': false,
            'encryptionValid': true,
            'validStruct': false,
        }

        this.#validJson()
        this.#checkEncrypted()
        this.#checkStruct()

        this.state['valid'] = [
            'validJson',
            'encryptionValid',
            'validStruct'
        ]
            .map( key  => this.#validation[ key ] )
            .every( a => a )

        return true
    }


    #validJson() {
        try {
            const raw = fs.readFileSync(
                this.state['path'],
                'utf-8'
            )

            this.content = JSON.parse( raw )
            this.#validation['validJson'] = true
        } catch( e ) {
            console.log( 'e', e )
        }

        return true
    }
```

```
#checkEncrypted() {
    if( !this.#validation['validJson'] ) {
        return true
    }

    this.#validation['encryptionActivated'] = [ 'iv', 'content' ]
        .every( item => {
            const result = Object
                .keys( item )
                .every( key => {
                    const r = Object
                        .keys( this.content )
                        .some( obj => {
                            Object
                                .keys( obj )
                                .includes( key )
                        } )
                    return r
                } )
            return result
        } )

    if( this.#validation['encryptionActivated'] ) {
        try {
            const decrypt = this.#cryption
                .decrypt( { 'hash': this.content } )

            console.log( '>> Decrypt', decrypt )

            this.content = JSON.parse( decrypt['content'])

            console.log( '>>', this.#validation['encrypted'] )
        } catch( e ) {
            console.log( '>> Decrypt Error', e )
            this.#validation['encryptionValid'] = false
        }

    }

    return true
}


#checkStruct() {
    if( !this.#validation['validJson'] ) {
        return true
    }

    const checks = Object
        .entries( this.config['environment']['addresses']['structs']['deployer'] )
        .reduce( ( acc, a ) => {
            const [ cmd, tests ] = a
            const keys = cmd
                .split( this.config['environment']['addresses']['structs']['split'] )

            acc[ cmd ] = {
                'exists': false,
                'checks': keys.map( b => false )
            }

            let value = null
            switch( keys.length ) {
                case 1:
                    value = this.content['data'][ keys[ 0 ] ]
                    break
                case 2:
                    try {
                        value = this.content['data'][ keys[ 0 ] ][ keys[ 1 ] ]
                    } catch {}
                    break
                default:
                    /*
                        this.#consolePrintLine( {
                            'first' : &nbsp;"${key.length}" not defined&nbsp;
                        } )
                    */
                    break
            }

            value === undefined ? value = null : ''
            value !== null ? acc[ cmd ]['exists'] = true : ''

            if( acc[ cmd ]['exists'] ) {
                acc[ cmd ]['checks'] = tests
                    .map( mode => {
                        let check = false
                        switch( mode ) {
                            case 'String':
                                if( typeof value === 'string' || value instanceof String ) {
                                    check = true
                                }
                                break
                            case 'Int':
                                Number.isInteger( value ) ? check = true : ''
                                break
                            case 'MinaPublicAddress':
                                if( typeof value === 'string' || value instanceof String ) {
                                    const t = value.match( this.config['environment']['addresses']['structs']
['minaAddressRegex'] )
                                    if( t !== undefined && t !== null ) {
                                        check =  true
                                    }
                    }
                    break
                case 'Array':
                    Array.isArray( value ) ? check = true : ''
                    break
            }

            return check
        } )
    }

    return acc
}, {} )

this.#validation['validStruct'] = Object
    .entries( checks )
    .map( a => {
        const [ key, value ] = a
        return value['checks']
            .every( a => a )
    } )
    .every( a => a )
}


async #addSnarkyjs() {
    this.#snarkyjs = await import( 'snarkyjs' )

    const node = this.config['network'][ this.config['network']['use'] ]['nodeProxy']
    const Berkeley = this.#snarkyjs.Mina.BerkeleyQANet( node )
    this.#snarkyjs.Mina.setActiveInstance( Berkeley )
    this.state['snarkyIsReady'] = true

    return true
}


async #accountStatus() {
    const checks = {
        'fetch': false,
        'known': false
    }

    if( this.state['valid'] ) {
        if( !this.state['snarkyIsReady'] ) {
            await this.#addSnarkyjs()
        }
    }

    let account = {}
    try {
        account = await this.#snarkyjs.fetchAccount( {
            'publicKey': this.content['data']['address']['public']
        } )
        checks['fetch'] = true
    } catch( e ) {
        console.log( &nbsp;Error could not catch Account&nbsp; )
    }

    if( checks['fetch'] ) {
        if( account['account'] !== undefined ) {
            checks['known'] = true
        }
    }

    const result = {
        'valid': checks['fetch'],
        'known': checks['known'],
        'account': account
    }

    return result
}


async #accountBalance( { status } ) {
    let balance = {
        'balance': null,
        'transactionsLeft': null,
        'nonce': null
    }

    if( status['valid'] & status['known'] ) {
        balance = [ 'balance', 'nonce' ]
            .reduce( ( acc, key, index ) => {
                let value
                try {
                    switch( key ) {
                        case 'balance':
                            acc['transactionsLeft'] = null
                            value = status['account']['account']['balance']
                            acc[ key ] = parseInt( value )

                            const network = this.config['network']['use']
                            const transactionFee = this.config['network'][ network ]['transaction_fee']

                            if( acc[ key ] > transactionFee ) {
                                acc['transactionsLeft'] = Math.floor(
                                    acc[ key ] / transactionFee
                                )
                            } else {
                                acc['transactionsLeft'] = 0
                            }
                            break
                    case 'nonce':
                        value = status['account']['account']['nonce']['value']
                        acc[ key ] = parseInt( value )
                        break
                    default:
                        console.log( 'Something went wrong' )
                        break
                }
            } catch( e ) {
                acc[ key ] = null
            }

            return acc
        }, {} )
    }

    this.state['balance'] = balance

    if( this.state['balance']['transactionsLeft'] !== null ) {
        if( this.state['balance']['transactionsLeft'] !== 0 ) {
            this.state['useable'] = true
        }
    }

    return true
}


#newAccountStruct( { type } ) {
    const struct = {
        'meta': {
            'fileName': null,
            'easyMinaVersion': null
        },
        'data': {
            'name': this.config['meta']['name'],
            'type': type,
            'time': {
                'unix': this.config['meta']['unix'],
                'format': this.config['meta']['format']
            },
            'address': {
                'public': null,
                'private': null
            },
            'explorer': {},
            'faucets': [],
            'comment': this.config['console']['messages']['accountComment']
        }
    }

    struct['meta']['easyMinaVersion'] = this.config['meta']['easyMinaVersion']
```

```
        struct['data']['address']['private'] = this.#snarkyjs.PrivateKey
            .random()
            .toBase58()

        struct['data']['address']['public'] = this.#snarkyjs.PrivateKey
            .fromBase58( struct['data']['address']['private'] )
            .toPublicKey()
            .toBase58()

        const network = this.config['network']['use']
        struct['data']['explorer'][ network ] = ''
        struct['data']['explorer'][ network ] += this.config['network'][ network ]['explorer']['wallet']
        struct['data']['explorer'][ network ] += struct['data']['address']['public']

        struct['meta']['fileName'] = this.config['environment']['addresses']['deployers']['fileName']

        return struct
    }


    async #requestFaucet( { receiver } ) {
        const config = {
            'network': { ...this.config['network'] },
            'graphQl': { ...this.config['graphQl'] },
            'print': { ...this.config['print'] },
            'messages': { ...this.config['messages'] },
        }

        const faucet = new Faucet()
        faucet.init( { config } )
        const response = await faucet.start( { receiver } )

        return response
    }
}
```

</file>

<file>

## path: /src/accounts/Cryption.mjs

url: https://github.com/EasyMina/easyMina/blob/main/src/accounts/Cryption.mjs

```
/*
    Name
        Cryption.js
    Description
        Encrypts and decrypts user credentials
    Blocks
        {}
    Public:
        Variables
        Methods
            .init( { silent, secret } )
            .encrypt( { text } )
            .decrypt( { })
*/


import crypto from 'crypto'


export class Cryption {
    #config
    #state
    #silent

    constructor() {
        this.#config = {
            'algorithm': 'aes-256-cbc'
        }
        this.#state
        this.#silent
    }


    init( { silent=false, secret=null } ) {
        this.#silent = silent
        this.#state = {
            'secret': null
        }

        this.#addSecret( { secret } )

        return this
    }


    encrypt( { text } ) {
        const iv = crypto.randomBytes( 16 )

        const cipher = crypto.createCipheriv(
            this.#config['algorithm'],
            this.#state['secret'],
            iv
        )
        const encrypted = Buffer.concat( [
            cipher.update( text ),
            cipher.final()
        ] )

        const content = encrypted
            .toString( 'hex' )
        const result = {
            'iv': iv.toString('hex'),
            'content': content
        }

        return result
    }


    decrypt( { hash } ) {
        const decipher = crypto.createDecipheriv(
            this.#config['algorithm'],
            this.#state['secret'],
            Buffer.from( hash['iv'], 'hex' )
        )

        const decrypted = Buffer.concat( [
            decipher.update( Buffer.from( hash['content'], 'hex' ) ),
            decipher.final()
        ] )

        return decrypted.toString()
    }


    #addSecret( { secret } ) {
        if ( typeof secret === 'string' || secret instanceof String ) {
            const str = this.#hashString( { 'string': secret } )
            this.#state['secret'] = str

        } else {
            console.log( 'Secret is not a string.' )
            process.exit( 1 )
        }
    }


    #hashString( { string } ) {
        const hash = crypto.createHash( 'sha256' )
        hash.update( string )
        const bytes32 = hash.digest()

        return bytes32
    }
}
```

</file>

<file>

path: /src/accounts/Faucet.mjs

url: https://github.com/EasyMina/easyMina/blob/main/src/accounts/Faucet.mjs

```
/*
    Name
        Account.js
    Description
        Requests test tokens and displays the response in a structured format.
    Blocks
        {
            'graphQl': { config['graphQl'] },
            'messages': { config['messages'] },
            'network': { config['network'] },
            'print': { config['print'] },
        }
    Public:
        Variables
        Methods
            .init( { config } )
            .start( { receiver } )
*/


import axios from 'axios'
// import { GraphQl } from '../interactions/GraphQl.mjs'
import { PrintConsole } from './../helpers/PrintConsole.mjs'
import moment from 'moment'


export class Faucet {
    #config
    #graphql
    #printConsole

    constructor() {
        this.#config = {}
    }


    init( { config={} } ) {
        this.#config = { ...config }

        this.#printConsole = new PrintConsole()
        this.#printConsole.init( {
            'config': {
                'print': { ...this.#config['print'] },
                'messages': { ...this.#config['messages'] },
                'network': { ...this.#config['network'] }
            }
        })

        return true
    }
```

```
async start( { receiver } ) {
    process.stdout.write( '    Faucet               ' )
    const network = this.#config['network']['use']
    const response = await this.#request( {
        'url': this.#config['network'][ network ]['faucet']['api'],
        'address': receiver,
        'network': this.#config['network'][ network ]['faucet']['network'],
        'transaction': this.#config['network'][ network ]['explorer']['transaction']
    } )

    if( response['status'] ) {
        console.log( &nbsp;      ${response['faucet']['explorer']}&nbsp;)
    } else {
        console.log( &nbsp;      ${response['faucet']['explorer']}&nbsp; )
        process.exit( 1 )
    }

    return response
}


async #request( { url, address, network, transaction } ) {
    const result = {
        'status': false,
        'status_str': 'failed',
        'message': null,
        'faucet': {
            'transaction': '',
            'explorer': null,
            'network': network,
            'timestamp': moment().unix()
        }
    }

    try {
        const response = await axios.post(
            url,
            {
                'network': network,
                'address': address
            },
            {
                'headers': {
                    'Content-Type': 'application/json',
                    'Accept': '*/*'
                },
                'maxBodyLength': Infinity
            }
        )
        result['status'] = true
        result['status_str'] = 'finished'
        result['message'] = response.data.status
        result['faucet']['transaction'] = response.data.message.paymentID
        result['faucet']['explorer'] = &nbsp;${transaction}${result['faucet']['transaction']}&nbsp;
    } catch( error ) {
        try {
            const status = error['response']['data']['status']
            result['message'] = status
            if( result['message'] === 'rate-limit' ) {
                result['faucet']['transaction'] = 'manual'
            }
        } catch {
            result['message'] = 'unknwon'
        }
    }

    return result
  }
}
```

</file>

<file>

## path: /src/data/config.mjs

url: https://github.com/EasyMina/easyMina/blob/main/src/data/config.mjs

```
/*
  Name
    config.mjs
  Description
    This file contains almost all parameters, gathered in one place. This structuring is crucial to enable an
auto-generated documentation.
  Overview
    Blocks
      | NR | NAME        |
      |:--|:--          |
      | A | console      |
      | B | docs         |
      | C | environment |
      | D | graphQl      |
      | E | messages     |
      | F | meta         |
      | G | network      |
      | H | print        |
      | I | typescript  |
      | J | validations |

    In use:
      | NR | NAME           | A | B | C | D | E | F | G | H | I | J |
      |:--|:--            |:--|:--|:--|:--|:--|:--|:--|:--|:--|:--|
      | 1 | EasyMina.mjs      | X | X | X | X | X | X | X | X | X | X |
      | 2 | Account.mjs       | X |   | X | X | X | X | X | X |   |   |
      | 3 | Cryption.mjs      |   |   |   |   |   |   |   |   |   |   |
      | 4 | Faucet.mjs        |   |   |   | X | X |   | X | X |   |   |
      | 5 | Credential.mjs    |   |   | X |   |   |   |   |   |   |   |
      | 6 | SmartContract.mjs | X | X | X |   |   | X | X |   |   |   |
      | 7 | Workspace.mjs     |   |   | X |   |   | X |   |   | X | X |   |
      | 8 | PrintConsole.mjs  |   |   |   |   | X |   |   | X |   |   |
      | 9 | GraphQl.mjs       |   |   |   | X | X |   | X | X |   |   |
                                3   3   4   4   6   3   5   6   2   1
*/




import moment from 'moment'


export const configImported = {
  'meta': {
    'name': 'default',
    'unix': moment().unix(),
    'format': moment().format(),
    'easyMinaVersion': '0.01',
  },
  'docs': {
    'url': 'https://easymina.github.io/',
    'options': 'options',
    'setEnvironment': 'methods/setEnvironment.html',
    'fetchInformation': 'methods/fetchEnvironment.html',
    'deployContract': 'methods/deployContract.html'
  },
  'environment': {
    'addresses': {
      'splitter': '--',
      'root': '.mina/',
      'deployers': {
          'folder': 'deployers/',
          'fullFolder': null,
          'fileNameStruct': '{{name}}{{splitter}}{{unix}}.json',
          'fileName': null
      },
      'contracts': {
          'folder': 'contracts/',
          'fullFolder': null,
          'fileNameStruct': '{{name}}{{splitter}}{{unix}}.json'
      },
      'structs': {
          'split': '__',
          'minaAddressRegex': /^B62[a-km-zA-HJ-NP-Z1-9]{52}$/,
          'deployer': {
```

```
                'name': [ 'String' ],
                'type': [ 'String' ],
                'time__unix': [ 'Int' ],
                'time__format': [ 'String' ],
                'address__private': [ 'String' ],
                'address__public': [ 'MinaPublicAddress' ],
                'comment': [ 'String' ],
                'faucets': [ 'Array' ]
            }
        },
    },
    'workspace': {
        'contracts': {
            'root': 'workdir/',
            'typescript': {
                'folder': 'typescript/',
                'full': null,
                'fullRelative': null,
                'fileName': null
            },
            'build': {
                'folder': 'build/',
                'full': null,
                'fullRelative': null
            }
        },
        'gitignore': '.gitignore'
    },
    'template': {
        'regexs': {
            'gist':  'string__gistTemplate',
            'https': 'string__urlHttps',
            'plain': 'string__plainTemplate'
        },
        'source': {
            //'source': 'https://gist.github.com/a6b8/e6baafbad2e42e4a259c10b6a3dcc836'
            'content': "import {\n  Field,\n  SmartContract,\n  state,\n  State,\n  method,\n} from 'o1js';\nexport
class Main extends SmartContract {\n    events = { 'easyMina': Field };\n    @state(Field) num = State<Field>
();\n    \n    \n    init() {\n        super.init();\n        this.num.set( Field( 3 ) );\n        this.emitEvent( 'easyMina',
Field( 123456789 ) );\n    }\n    @method update( square: Field ) {\n        const currentState =
this.num.get();\n        this.num.assertEquals( currentState );\n        square.assertEquals( currentState.mul(
currentState ) );\n        this.num.set( square );\n    }\n}",
            'name': '{{name}}.ts',
            'className': 'Main'
        },
        'parse': {
            'gist': 'https://api.github.com/gists/{{three}}',
            'https': '{{one}}',
            'plain': '{{one}}'
        }
    }
},
'network': {
    'use': 'berkeley',
    'berkeley': {
        'explorer': {
            'transaction': 'https://berkeley.minaexplorer.com/transaction/',
            'wallet': 'https://berkeley.minaexplorer.com/wallet/'
        },
        'node': 'https://berkeley.graphql.minaexplorer.com',
        'nodeProxy': 'https://proxy.berkeley.minaexplorer.com/graphql',
        'graphQl': 'https://berkeley.graphql.minaexplorer.com',
        'faucet': {
            'api': 'https://faucet.minaprotocol.com/api/v1/faucet',
            'web': 'https://faucet.minaprotocol.com/?address={{address}}',
            'network': 'berkeley-qanet'
        },
        'transaction_fee': 100_000_000,
    }
},
'console': {
    'symbols': {
        'neutral': '🟥',
        'onProgress1': '    ',
        'onProgress2': '    ',
        'ok1': '    ',
        'ok2': '    ',
        'split': '',
        'failed': '    '
    },
    'space': 30,
    'messages': {
        'accountComment': 'Do not share this file with someone.'
    }
},
'graphQl': {
    'render': {
        'frameInterval': 1000,
        'delayBetweenRequests': 10000,
        'singleMaxInSeconds': 30
    },
    'presets': {
        'singleRequest': {
            'mode': 'maxTries',
            'maxTries': 1,
            'requestInterval': 0
        },
        'newTransaction': {
            'mode': 'maxMinutes',
            'maxMinutes': 3,
            'requestIntervalInSeconds': 0
        }
    },
    'transactionByHash': {
            'key': 'transaction',
            'type': 'hash',
            'cmd': {
                'query': "query q($hash: String!) {\n  transaction(query: {hash: $hash}) {\n    hash\n    dateTime\n
blockHeight\n    from\n    nonce\n    to\n    toAccount {\n      token\n    }\n  }\n}",
                'variables': {
                    'hash': '5Jv6t2eyPZgGNWxct5kkhRwmF5jkEYNZ7JCe1iq6DMusvXGmJwiD'
                },
                'operationName': 'q'
              }
        },
        'latestBlockHeight': {
            'key': 'block',
            'type': 'hash',
            'cmd': {
                'query': "query q($blockHeight_lt: Int) {\n  block(query: {blockHeight_lt: $blockHeight_lt}) {\n
blockHeight\n    dateTime\n  }\n}",
                'variables': {
                    'blockHeight_lt': 999999999
                },
                'operationName': 'q'
            }
        },
        'latestBlockHeights': {
            'key': 'blocks',
            'type': 'array',
            'cmd': {
                'query': "query q($limit: Int) {\n  blocks(limit: $limit, sortBy: BLOCKHEIGHT_DESC) {\n
blockHeight\n    protocolState {\n      consensusState {\n        slotSinceGenesis\n        slot\n      }\n    }\n
dateTime\n    receivedTime\n  }\n}",
                'variables': {
                    'limit': 10
                },
                'operationName': 'q'
              }
        },
        'latestEventsFromContract': {
            'key': 'events',
            'type': 'array',
            'cmd': {
                'query': "query q($limit: Int!, $blockHeight_lt: Int!, $creator: String!) {\n events(query:
{blockHeight_lt: $blockHeight_lt, blockStateHash: {creator: $creator}}, sortBy: BLOCKHEIGHT_DESC, limit:
$limit) {\n blockHeight\n dateTime\n event\n blockStateHash {\n creatorAccount {\n publicKey\n }\n }\n }\n}",
                'variables': {
                    'limit': 10,
                    'blockHeight_lt': 999999999,
                    'creator': 'B62qnLVz8wM7MfJsuYbjFf4UWbwrUBEL5ZdawExxxFhnGXB6siqokyM'
                },
                'operationName': 'q'
            }
        },
        'latestEventsFromContractByBlockHeight': {
            'key': 'events',
            'type': 'array',
            'cmd': {
                'query': "query q($limit: Int!, $blockHeight: Int!, $publicKey: String!) {\n events(limit: $limit, query: {blockHeight: $blockHeight, blockStateHash: {creatorAccount: {publicKey: $publicKey}}}) {\n blockHeight\n dateTime\n event\n blockStateHash {\n creatorAccount {\n publicKey\n }\n }\n }\n}",
                'variables': {
                    'limit': 10,
                    'blockHeight': 2785,
                    'publicKey': 'B62qnLVz8wM7MfJsuYbjFf4UWbwrUBEL5ZdawExxxFhnGXB6siqokyM'
                },
                'operationName': 'q'
            }
        }
    },
    'print': {
        'split': '__',
        'insertManual': '<<insert>>',
        'regexs': {
            'format': /{{[a-zA-Z0-9_]+}}/g,
            'minaAddress': /^B62[a-km-zA-HJ-NP-Z1-9]{52}$/
        },
        'levels': [ '', '  {{enumerations__1}}.   ', '  {{enumerations__1}}.{{enumerations__2}}. ', '  ' ],
        'spaces': {
            'standard': 24,
            'extended': 40
        },
        'enumerations': {
            '1': [ 'A', 'B', 'C', 'D', 'E', 'F', 'G', 'H', 'I', 'J', 'K', 'L', 'M', 'N', 'O', 'P', 'Q', 'R', 'S', 'T', 'U', 'V', 'W', 'X', 'Y', 'Z' ],
            '2': [ '1', '2', '3', '4', '5', '6', '7', '8', '9', '10', '11', '12', '13', '14', '15', '16', '17', '18', '19', '20', '21', '22', '23', '24', '25', '26' ],
            'notFound': ''
        },
        'userInteractions': {
            'yesNo': {
                'validatons': [
                    {
                        'validation': 'messages__yes',
                        'finished': { 'format': '{{messages__validationIsSetTo}}', 'output': true },
                    },
                    {
                        'validation': 'messages__no',
                        'finished': { 'format': '{{messages__validationIsSetTo}}', 'output': true },
                    }
                ],
                'failed': {
                    'format': '{{messages__validationIsSetTo}}',
                    'output': null
                }
            },
            'minaAddress': {
                'validations': [
                    {
                        'validation': 'regexs__minaAddress',
                        'finished': {
                            'format': '{{levels}}{{messages__transaction}}{{spaces__standard}}{{status__standard}}{{messages__validationSuccess}}',
                            'output': 'userInteractions__input'
                        }
                    }
                ],
                'failed': {
                    'format': '{{levels}}{{messages__transaction}}{{spaces__standard}}{{status__standard}}{{messages__validationFailed}}',
                    'output': null
                }
            }
        },
        'status': {
            'standard': {
                'finished': [ '    ' ],
                'failed': [ '    ' ],
                'progress': [ '    ' ]
            },
            'loading': {
                'finished': [],
                'failed': [],
                'progress': [
                    "▰▱▱",
                    "▰▰▱",
                    "▰▰▰",
                    "▱▰▰",
                    "▱▱▰",
                    "▱▱▱",
                ]
                /*
                'progress': [
                    "▰▱▱▱▱▱▱",
                    "▰▰▱▱▱▱▱",
                    "▰▰▰▱▱▱▱",
                    "▰▰▰▰▱▱▱",
                    "▰▰▰▰▰▱▱",
                    "▰▰▰▰▰▰▱",
                    "▰▰▰▰▰▰▰",
                    "▰▰▰▰▰▰▱",
                    "▰▰▰▰▰▱▱",
                    "▰▰▰▰▱▱▱",
                    "▰▰▰▱▱▱▱",
                    "▰▰▱▱▱▱▱"
                ]
                */
            }
        },
        'structs': {
            'newLine': {
                'question':  null,
                'format': {
                    'finished': ''
                },
                'cmds': null
            },
            'title': {
                'question':  null,
                'format': {
                    'finished': '{{levels}}{{external__title}}'
                },
                'cmds': null
            },
            'standard': {
                'question': null,
                'format': {
                    'progress': '{{levels}}{{messages__standardFront}} {{spaces__standard}}{{status__standard}}
    {{status__loading}} {{messages__standardProgress}} {{messages__didYouKnow}}',
                    'finished': '{{levels}}{{messages__standardFront}} {{spaces__standard}}{{status__standard}}
    {{messages__standardFinished}}',
                    'failed': '{{levels}}{{messages__standardFront}} {{spaces__standard}}{{status__standard}}
    {{messages__standardFailed}}'
                },
                'cmds': null
            },
            'addDeployers': {
                'question': null,
                'format': {
                    'progress': '{{levels}}{{messages__addDeployersFront}} {{spaces__standard}}
    {{status__standard}} {{status__loading}} {{messages__addDeployersProgress}}',
                    'finished': '{{levels}}{{messages__addDeployersFront}} {{spaces__standard}}
    {{status__standard}} {{messages__addDeployersSuccess}}',
                    'failed': '{{levels}}{{messages__addDeployersFront}} {{spaces__standard}}{{status__standard}}
    {{messages__addDeployersFailed}}'
                },
                'cmds': null
            },
            'addEnvironmentAddresses': {
                'question': null,
                'format': {
                    'progress': '{{levels}}{{messages__addEnvironmentAddressesFront}} {{spaces__standard}}
    {{status__standard}} {{status__loading}} {{messages__addEnvironmentAddressesProgress}}',
                    'finished': '{{levels}}{{messages__addEnvironmentAddressesFront}} {{spaces__standard}}
    {{status__standard}} {{messages__addEnvironmentAddressesSuccess}}',
                    'failed': '{{levels}}{{messages__addEnvironmentAddressesFront}} {{spaces__standard}}
    {{status__standard}} {{messages__addEnvironmentAddressesFailed}}'
                },
                'cmds': null
            },
            'getFaucet': {
                'question': null,
                'format': {
                    'progress': '{{levels}}{{messages__getFaucetFront}} {{spaces__standard}}{{status__standard}}
    {{status__loading}} {{messages__getFaucetProgress}}',
                    'finished': '{{levels}}{{messages__getFaucetFront}} {{spaces__standard}}{{status__standard}}
    {{messages__getFaucetSuccess}}',
                    'failed': '{{levels}}{{messages__getFaucetFront}} {{spaces__standard}}{{status__standard}}
    {{messages__getFaucetFailed}}'
                },
                'cmds': null
            },
            'transactionByHash': {
                'question': null,
                'format': {
                    'progress': '{{levels}}{{messages__transactionByHashFront}} {{spaces__standard}}
    {{status__standard}} {{status__loading}} {{messages__transactionByHashProgress}}',
                    'finished': '{{levels}}{{messages__transactionByHashFront}} {{spaces__standard}}
    {{status__standard}} {{messages__transactionByHashSuccess}}',
                    'failed': '{{levels}}{{messages__transactionByHashFront}} {{spaces__standard}}
    {{status__standard}} {{messages__transactionByHashFailed}}'
                },
                'cmds': null
            },
            'latestBlockHeight': {
                'question': null,
                'format': {
                    'progress': '{{levels}}{{messages__latestBlockHeightFront}} {{spaces__standard}}
    {{status__standard}} {{messages__latestBlockHeightProgress}}',
                    'finished': '{{levels}}{{messages__latestBlockHeightFront}} {{spaces__standard}}
    {{status__standard}} {{messages__latestBlockHeightSuccess}}',
                    'failed': '{{levels}}{{messages__latestBlockHeightFront}} {{spaces__standard}}
    {{status__standard}} {{messages__latestBlockHeightFailed}}'
                },
                'cmds': null
            },
            'latestBlockHeights': {
                'question': null,
                'format': {
                    'progress': '{{levels}}{{messages__latestBlockHeightsFront}} {{spaces__standard}}
    {{status__standard}} {{messages__latestBlockHeightsProgress}}',
                    'finished': '{{levels}}{{messages__latestBlockHeightsFront}} {{spaces__standard}}
    {{status__standard}} {{messages__latestBlockHeightsSuccess}}',
                    'failed': '{{levels}}{{messages__latestBlockHeightsFront}} {{spaces__standard}}
    {{status__standard}} {{messages__latestBlockHeightsFailed}}'
                },
                'cmds': null
            },
            'latestEventsFromContract': {
                'question': null,
                'format': {
                    'progress': '{{levels}}{{messages__latestEventsFromContractFront}} {{spaces__standard}}
    {{status__standard}} {{messages__latestEventsFromContractProgress}}',
                    'finished': '{{levels}}{{messages__latestEventsFromContractFront}} {{spaces__standard}}
{{status__standard}} {{messages__latestEventsFromContractSuccess}}',
                    'failed': '{{levels}}{{messages__latestEventsFromContractFront}} {{spaces__standard}}
{{status__standard}} {{messages__latestEventsFromContractFailed}}'
                },
                'cmds': null
            },
            'latestEventsFromContractByBlockHeight': {
                'question': null,
                'format': {
                    'progress': '{{levels}}{{messages__latestEventsFromContractByBlockHeightFront}}
{{spaces__standard}}{{status__standard}}
{{messages__latestEventsFromContractByBlockHeightProgress}}',
                    'finished': '{{levels}}{{messages__latestEventsFromContractByBlockHeightFront}}
{{spaces__standard}}{{status__standard}}
{{messages__latestEventsFromContractByBlockHeightSuccess}}',
                    'failed': '{{levels}}{{messages__latestEventsFromContractByBlockHeightFront}}
{{spaces__standard}}{{status__standard}} {{messages__latestEventsFromContractByBlockHeightFailed}}'
                },
                'cmds': null
            }
        }
    },
    'messages': {
        'use': 'en',
        'en': {
            'transactionByHashFront': [ 'Get Transaction by Hash' ],
            'transactionByHashProgress': [ 'Waiting for Transaction ID: {{external__hash}}' ],
            'transactionByHashSuccess': [ 'Success! {{custom__networkExplorerTransaction}}
{{external__hash}}' ],
            'transactionByHashFailed': [ 'Transaction not found ({{external__hash}})' ],

            'latestBlockHeightFront': [ 'Get Latest Block Height' ],
            'latestBlockHeightProgress': [ 'Wait for response.' ],
            'latestBlockHeightSuccess': [ 'Success! Current Block Height is: {{external__blockHeight}}' ],
            'latestBlockHeightFailed': [ 'Unable to fetch results.' ],

            'latestBlockHeightsFront': [ 'Get Latest Blocks' ],
            'latestBlockHeightsProgress': [ 'Waiting for the latest Blocks.' ],
            'latestBlockHeightsSuccess': [ 'Success! Received {{external__count}} blocks.' ],
            'latestBlockHeightsFailed': [ 'Unable to fetch results.' ],

            'latestEventsFromContractFront': [ 'Get Events from Contract' ],
            'latestEventsFromContractProgress': [ 'Waiting for Events from ({{external__creator}})' ],
            'latestEventsFromContractSuccess': [ 'Success! Received {{external__count}} Events.' ],
            'latestEventsFromContractFailed': [ 'Unable to fetch results.' ],

            'latestEventsFromContractByBlockHeightFront': [ 'Get Events by Contract/Block Height' ],
            'latestEventsFromContractByBlockHeightProgress': [ 'Waiting for confirmation, {{external__receiver}}'
],
            'latestEventsFromContractByBlockHeightSuccess': [ 'Success! Received {{external__count}} Events,
({{external__publicKey}}, {{external__blockHeight}})' ],
            'latestEventsFromContractByBlockHeightFailed': [ 'Failed! Creator: {{external__publicKey}}, Block
```

```
Height {{external__blockHeight}}' ],

            'getFaucetFront': [ 'Get Faucet' ],
            'getFaucetProgress': [ 'Waiting for confirmation! {{external__receiver}}' ],
            'getFaucetSuccess': [ 'Success! {{external__explorer}}' ],
            'getFaucetFailed': [ 'Failed!' ],

            'addEnvironmentAddressesFront': [ 'Scan Environment' ],
            'addEnvironmentAddressesProgress': [ 'Scan folders!' ],
            'addEnvironmentAddressesSuccess': [ 'Found {{external__deployers}} Deployers,
{{external__contracts}} Contracts' ],
            'addEnvironmentAddressesFailed': [ 'Failed!' ],

            'addDeployersFront': [ 'Deployers' ],
            'addDeployersProgress': [ 'Scanning!' ],
            'addDeployersSuccess': [ 'Ready to use: {{external__readyToUse}}, Pending: {{external__pending}}.'
],
            'addDeployersFailed': [ 'Failed!' ],

            'errorKeyNotFound': [ 'Key: "<<insert>>" not found' ],
            'errorTypeNotFound': [ 'Type: "<<insert>>" not found' ],

            'standardFront': [ '{{external__front}}' ],
            'standardProgress': [ 'Next slot: {{external__progress}}.' ],
            'standardFinished': [ '{{external__progress}}.' ],
            'standardSuccess': [ '{{external__success}}' ],
            'standardFailed': [ '{{external__failed}}' ],

            'didYouKnow': [ '' ]
        }
    },
    'typescript': {
        'template': {
            'compilerOptions': {
                'target': 'ES2019',
                'module': 'es2022',
                'lib': [ 'dom', 'esnext' ],
                'outDir': null,
                'rootDir': null,
                'strict': true,
                'strictPropertyInitialization': false, // to enable generic constructors, e.g. on CircuitValue
                'skipLibCheck': true,
                'forceConsistentCasingInFileNames': true,
                'esModuleInterop': true,
                'resolveJsonModule': true,
                'moduleResolution': 'node',
                'experimentalDecorators': true,
                'emitDecoratorMetadata': true,
                'allowJs': true,
                'declaration': false,
                'sourceMap': false,
                'noFallthroughCasesInSwitch': true,
                'allowSyntheticDefaultImports': true,
                'isolatedModules': true,
                // 'outFile': null
            },
            'include': null,
            'exclude': []
        },
        'fileName': 'tsconfig.json'
    },
    "validations": {
        "keyPaths": {
            "meta__name": {
                "userPath": {
                    "en": "projectName"
                },
                "validation": "string__hyphenAsSpace",
                "category": "general",
                "methods": [ 'setEnvironment', 'deployContract' ]
            },
            "environment__addresses__splitter": {
                "userPath": {
                    "en": "fileNameSplitter"
                },
                "validation": "string__splitters",
                "category": "account",
                "methods": [ 'setEnvironment' ]
            },
            "environment__addresses__root": {
                "userPath": {
                    "en": "credentialsRootFolderName"
                },
                "validation": "string__folderNameInvisible",
                "category": "account",
                "methods": [ 'setEnvironment' ]
            },
            "environment__addresses__deployers__folder": {
                "userPath": {
                    "en": "accountsFolderName"
                },
                "validation": "string__folderName",
                "category": "account",
                "methods": [ 'setEnvironment' ]
            },
            "environment__addresses__deployers__fileName": {
                "userPath": {
                    "en": "deployerFileName"
                },
                "validation": "string__fileNameCredentials",
                "category": "account",
                "methods": []
            },
            "environment__addresses__contracts__folder": {
```

```
        "userPath": {
            "en": "contractsFolderName"
        },
        "validation": "string__folderName",
        "category": "account",
        "methods": [ 'setEnvironment' ]
    },
    "environment__workspace__contracts__root": {
        "userPath": {
            "en": "workspaceRootFolderName"
        },
        "validation": "string__folderName",
        "category": "workspace",
        "methods": [ 'setEnvironment' ]
    },
    "environment__workspace__contracts__typescript__folder": {
        "userPath": {
            "en": "workspaceTypescriptFolderName"
        },
        "validation": "string__folderName",
        "category": "workspace",
        "methods": [ 'setEnvironment' ]
    },
    "environment__workspace__contracts__typescript__fileName": {
        "userPath": {
            "en": "smartContractFileName"
        },
        "validation": "string__typescriptFileName",
        "category": "workspace",
        "methods": [ 'deployContract' ]
    },
    "environment__workspace__contracts__build__folder": {
        "userPath": {
            "en": "workspaceBuildFolderName"
        },
        "validation": "string__folderName",
        "category": "workspace",
        "methods": [ 'setEnvironment' ]
    },
    "environment__template__source__content": {
        "userPath": {
            "en": "smartContractContentDefault"
        },
        "validation": "string__plainTemplate",
        "category": "workspace",
        "methods": [ 'setEnvironment' ]
    },
    "environment__template__source__name": {
        "userPath": {
            "en": "smartContractNameDefault"
        },
        "validation": "string__typescriptFileNamePlaceholder",
        "category": "workspace",
        "methods": [ 'setEnvironment' ]
    },
    "network__use": {
        "userPath": {
            "en": "networkName"
        },
        "validation": "string__networkName",
        "category": "network",
        "methods": [ 'setEnvironment', 'deployContract' ]
    },
    "network__berkeley__explorer__transaction": {
        "userPath": {
            "en": "transactionExplorer"
        },
        "validation": "string__urlHttps",
        "category": "network",
        "methods": [ 'setEnvironment', 'deployContract' ]
    },
    "network__berkeley__explorer__wallet": {
        "userPath": {
            "en": "walletExplorer"
        },
        "validation": "string__urlHttps",
        "category": "network",
        "methods": [ 'setEnvironment', 'deployContract' ]
    },
    "network__berkeley__node": {
        "userPath": {
            "en": "berkeleyNode"
        },
        "validation": "string__urlHttps",
        "category": "network",
        "methods": [ 'setEnvironment', 'deployContract' ]
    },
    "network__berkeley__nodeProxy": {
        "userPath": {
            "en": "berkeleyNodeProxy"
        },
        "validation": "string__urlHttps",
        "category": "network",
        "methods": [ 'setEnvironment', 'deployContract' ]
    },
    "network__berkeley__graphQl": {
        "userPath": {
            "en": "berkeleyGraphQl"
        },
        "validation": "string__urlHttps",
        "category": "network",
        "methods": [ 'setEnvironment', 'deployContract' ]
    },
    "network__berkeley__transaction_fee": {
        "userPath": {
            "en": "transactionFee"
        },
        "validation": "string__urlHttps",
        "category": "network",
        "methods": [ 'setEnvironment', 'deployContract' ]
    },
    "console__messages__accountComment": {
        "userPath": {
            "en": "accountMessage"
        },
        "validation": "string__default",
        "category": "general",
        "methods": [ 'setEnvironment', 'deployContract' ]
    },
    "print__spaces__standard": {
        "userPath": {
            "en": "consoleSpacesStandard"
        },
        "validation": "integer__positiveNumber",
        "category": "general",
        "methods": [ 'setEnvironment', 'deployContract' ]
    },
    "print__spaces__extended": {
        "userPath": {
            "en": "consoleSpacesStandard"
        },
        "validation": "integer__positiveNumber",
        "category": "general",
        "methods": [ 'setEnvironment', 'deployContract' ]
    },
    "messages__use": {
        "userPath": {
            "en": "consoleLanguage"
        },
        "validation": "string__language",
        "category": "general",
        "methods": [ 'setEnvironment', 'deployContract' ]
    }
},
"regexs": {
    "string": {
        "default": {
            "description": {
                "en": "Allow A-Z, a-b and 0-9 as value."
            },
            "regex": "^[a-zA-Z0-9\\s.-]*{{content}}quot;,
            "message": {
                "en": "Please use only letters (A-Z, a-z), numbers (0-9), spaces, periods, or hyphens."
            }
        },
        "hyphenAsSpace": {
          "description": {
            "en": "Allow A-Z, a-b and 0-9 as value, use '-' for space."
          },
          "regex": "^[a-zA-Z0-9-]*{{content}}quot;,
          "message": {
            "en": "You can use only letters (A-Z, a-z), numbers (0-9), and hyphens. Replace spaces with
hyphens."
          }
        },
        "fileNameCredentials": {
          "description": {
            "en": "Allows file names with the structure \"{{name}}--{{unix}}.json\". {{name}} can contain
alphanumerics and hyphens, and {{unix}} is a Unix timestamp."
          },
          "regex": "^[a-zA-Z0-9-]+--\\d+\\.json{{content}}quot;,
          "message": {
            "en": "The file name should follow the pattern \"{{name}}--{{unix}}.json\", where {{name}}
contains alphanumerics and hyphens, and {{unix}} is a Unix timestamp."
          }
        },
        "splitters": {
          "description": {
            "en": "Matches strings containing either \"__\" without consecutive occurrences or \"--\"
without consecutive occurrences"
          },
          "regex": "^(__(?!--)|--(?!__))*{{content}}quot;,
          "message": {
            "en": "The string can contain either \"__\" or \"--\" without consecutive occurrences."
          }
        },
        "folderName": {
          "description": {
            "en": "The first character must be from 'a' to 'z', followed by a combination of 'a' to 'z', '0' to '9'
and '-' or '_' . The last character must be a '/'."
          },
          "regex": "^[a-z](?!.*[-_]{2})[a-z0-9-_]*[a-z0-9]\\/{{content}}quot;,
          "message": {
            "en": "The folder name must start with a lowercase letter, followed by letters (a-z), numbers
(0-9), hyphens, or underscores. It must end with a '/'."
          }
        },
        "folderNameInvisible": {
          "description": {
            "en": "start with a \".\", followed by a lowercase letter, and may contain alphanumeric
characters, hyphens, and underscores, ending with an alphanumeric character and a \"\/""
          },
          "regex": "^[.][a-z](?!.*[-_]{2})[a-z0-9-_]*[a-z0-9]\\/{{content}}quot;,
          "message": {
            "en": "The folder name must start with a \".\" and a lowercase letter, followed by letters (a-z),
numbers (0-9), hyphens, or underscores. It must end with a '/'."
          }
        },
        "networkName": {
          "description": {
            "en": "Currently only \"berkeley\" is valid"
          },
          "regex": "^(berkeley|none){{content}}quot;,
          "message": {
            "en": "The network name can only be \"berkeley\" or \"none\"."
          }
        },
        "language": {
          "description": {
            "en": "Currently only \"en\" is valid"
          },
          "regex": "^(en|none){{content}}quot;,
          "message": {
            "en": "The language can only be \"en\" or \"none\"."
          }
        },
        "urlHttps": {
          "description": {
            "en": "URLs with optional \"http://\" or \"https://\", followed by a domain name, a top-level
domain (TLD) of at least two letters, and an optional path segment at the end"
          },
          "regex": "(https?:\\/\\/)?([a-zA-Z0-9.-]+)\\.([a-zA-Z]{2,})(\\/\\S*)?{{content}}quot;,
          "message": {
            "en": "Please enter a valid URL with optional \"http://\" or \"https://\" and a valid domain name
followed by a top-level domain (TLD) of at least two letters."
          }
        },
        "gistTemplate": {
          "description": {
            "en": "Matches strings with two words separated by '::'"
          },
          "regex": "^(\w+)::(\w+){{content}}quot;,
          "message": {
            "en": "The input should consist of two words separated by '::'."
          }
        },
        "plainTemplate": {
          "description": {
            "en": "Matches any sequence of characters, including newlines, in a single line"
          },
          "regex": "[\s\S]*",
          "message": {
            "en": "The input should contain any sequence of characters, including whitespace and non-
whitespace characters, and can be of any length, including an empty string"
          }
        },
        "typescriptFileName": {
          "description": {
            "en": "Matches file names starting with a lowercase letter, potentially containing multiple
occurrences of alphanumerics, hyphens, or underscores, with an optional period, and ending with the
extension \".ts\"."
                    },
                    "regex": "^[a-z](?:[a-z0-9_-]*\\.?)+\\.ts",
                    "message": {
                        "en": "The file name must start with a lowercase letter and can include multiple occurrences
of alphanumerics, hyphens, or underscores, with an optional period before ending with the extension \".ts\"."
                    }
                },
                "typescriptFileNamePlaceholder": {
                    "description": {
                        "en": "match strings starting with a lowercase letter, potentially containing multiple
occurrences of regular characters or \"{{a-z}}\" input followed by an optional period, and ending with the
extension.ts"
                    },
                    "regex": "^[a-z](?:([a-z0-9_-]*|\{\{[a-z]+\}\})*\.?)+\.ts",
                    "message": {
                        "en": "The filename must start with a lowercase letter and can include multiple occurrences of
regular characters or \"{{a-z}}\" input, with an optional period before ending with the extension.ts"
                    }
                }
            },
            "integer": {
                "positiveNumber": {
                    "description": {
                        "en": "This regular expression matches any positive integer with one or more digits."
                    },
                    "regex": "^[1-9]\\d*{{content}}quot;,
                    "message": {
                        "en": "Please enter a positive integer greater than zero."
                    }
                }
            }
        }
    }
}
```

</file>

<file>

## path: /src/environment/Credentials.mjs

url: https://github.com/EasyMina/easyMina/blob/main/src/environment/Credentials.mjs

```
/*
    Name
        Credentials.js
    Description
        Verifies the validity of stored accounts and whether the private key matches the public key.
    Blocks
        {
            'environment': { ...config['environment'] }
        }
    Public
        Variables
        Methods
            .init( { config } )
            .checkEnvironment()
            .checkAccounts()
*/


import fs from 'fs'


export class Credentials {
    #config
    #state
    #locations

    constructor() {}


    init( { config={} } ) {
        this.#config = { ...config }
        this.#state = {
            'folders': {}
        }

        this.#addLocations()
        return true
    }


    checkEnvironment() {
        // process.stdout.write( '    Folder              ' )

        this.#state['folders'] = this.#scanFolder()
        !this.#state['folders']['valid'] ? this.#createFolder() : ''


        let msg = ''
        msg += '   ├──── '
        msg += &nbsp;${this.#config['root']}&nbsp;
        msg += this.#state['folders']['valid'] ? '' : '*'

        console.log( msg )

        return true
    }


    checkAccounts() {
        // process.stdout.write( '    Accounts            ' )
        const result = this.#locations['accounts']
            .reduce( ( acc, a, index ) => {
                const [ key, value ] = a
                acc[ key ] = this.#scanAccounts( { 'type': key } )
                return acc
            }, {} )

        this.#state = {
            'folders': this.#state['folders'],
            ...result
        }

        const found = Object
            .entries( result )
            .map( ( a, index ) => {
                const key = this.#config[ a[ 0 ] ]['folder']
                const str = &nbsp;   |      ├─── ${key} (${a[ 1 ].length})&nbsp;
                console.log( str )
                return str
            } )

        /*
            .join( ', ' )

        let msg = ''
        msg += '      '
        // msg += 'Found! '
        msg += found
        console.log( msg )
        */

        return result
    }


    #addLocations() {
        this.#locations = {
            'root': [ 'mina', this.#config['root'] ],
            'accounts': [
                [ 'deployers', this.#config['deployers']['fullFolder'] ],
                [ 'contracts', this.#config['contracts']['fullFolder'] ]
            ],
            'both': null
        }

        this.#locations['both'] = Object
            .entries( this.#locations )
            .reduce( ( acc, a, index ) => {
                const [ key, values ] = a
                if( index === 0 ) {
                    acc.push( values )
                } else if( index === 1 ) {
                    values.forEach( b => acc.push( b ) )
                }
                return acc
            }, [] )

        return true
    }


    #scanFolder() {
        const result = this.#locations['both']
            .reduce( ( acc, a, index, all ) => {
                const [ key, value ] = a
                try {
                    acc[ key ] = fs.existsSync( value )
                } catch {
                    acc[ key ] = false
                }

                if( index === all.length-1 ) {
                    acc['valid'] = Object
                        .entries( acc )
                        .map( a => a[ 1 ] )
                        .every( a => a )
                }

                return acc
            }, {} )

        return result
    }


    #scanAccounts( { type } ) {

        if( !this.#state['folders']['valid'] ) {
            return []
        }

        const splitter = this.#config['splitter']
        const fileNameStructKeys = this.#config[ type ]['fileNameStruct']
            .replace( '.json', '' )
            .split( '{{splitter}}' )
            .map( a => {
                return a
                    .replace( '{{', '' )
                    .replace( '}}', '' )
            } )

        const valid = fs
            .readdirSync( this.#config[ type ]['fullFolder'] )
            .filter( a => a.endsWith( '.json' ) )
            .map( ( a, index ) => {
                const struct = a
                    .split( splitter )
                    .map( b => b.split( '.json')[ 0 ] )
                    .reduce( ( acc, b, index ) => {
                        acc[ fileNameStructKeys[ index ] ] = b
                        return acc
                    }, {} )

                struct['path'] = ''
                struct['path'] += this.#config[ type ]['fullFolder']
                struct['path'] += a

                return struct
            } )

        return valid
    }


    #createFolder() {
        const n = this.#locations['both']
            .forEach( a => {
                const [ key, value ] = a
                if( !this.#state['folders'][ key ] ) {
                    fs.mkdirSync( value )
                }
            } )

        return true
    }
    }
```

</file>

<file>

## path: /src/environment/SmartContract.mjs

url: https://github.com/EasyMina/easyMina/blob/main/src/environment/SmartContract.mjs

```
/*
    Name
        SmartContract.js
    Description
        This class contains methods through which one can interact with the Smart Contract.
    Blocks
        {
            'console': { ...config['console'] },
            'docs': { ...config['docs'] },
            'environment': { ...config['environment'] },
            'meta': { ...config['meta'] },
            'network': { ...config['network'] }
        }
    Public
        Variables
        Methods
            .init( { config, smartContractPath, smartContractClassName } ) (async)
            .deploy( { accountPath } ) (async)
*/


import fs from 'fs'
import moment from 'moment'


export class SmartContract {
    #config
    #contract
    #snarkyjs
    #state

    constructor() {}


    async init( { config={}, smartContractPath, smartContractClassName } ) {
        this.#config = { ...config }
        this.#state = {
            'contract': {
                'path': smartContractPath,
                'className': smartContractClassName
            }
        }


        await this.#addSnarkyjs()
        await this.#addContract()

        return  true
    }


    async deploy( { accountPath } ) {
        let struct

        console.log( &nbsp;  Smart Contract            ${this.#state['contract']['path']}&nbsp; )
        console.log( &nbsp;  Class Name                ${this.#state['contract']['className']}&nbsp; )
        struct = await this.#deployPrepare( { accountPath } )

        process.stdout.write( &nbsp;  Verification Key    &nbsp; )
        const start = new Date()
        await this.#addVerificationKey()
        const end = Math.round( ( new Date() - start ) / 1000 )
        console.log( '', &nbsp;      ${end} seconds&nbsp;)

        process.stdout.write( &nbsp;  Send Transaction      &nbsp; )
        struct = await this.#deploySend( { struct } )
        console.log( &nbsp;      ${struct['transaction']['explorer']}&nbsp; )

        process.stdout.write( &nbsp;  Save Contract         &nbsp; )
        const data = this.#deploySavePrepare( { struct } )
        const path = this.#deploySave( { data } )
        console.log( &nbsp;      ${path}&nbsp; )

        return true
    }


    async #addSnarkyjs() {
        this.#snarkyjs = await import( 'o1js' )

        return true
    }


    async #addContract() {
        const messages = []

        const struct = {
            'source': null,
            'className': null,
            'content': null,
            'verificationKey': null
        }

        struct['source'] = this.#state['contract']['path']
        struct['className'] = this.#state['contract']['className']

        struct['content'] = fs.readFileSync(
            struct['source'],
            'utf8'
        )

        const raw = await import( struct['source'] )
        struct['class'] = raw[ struct['className'] ]
        struct['methods'] = [ struct['className'], '_methods' ]
            .reduce( ( acc, key, index, all ) => {
                try {
                    acc = acc[ key ]
                    if( index === all.length - 1 ) {
                        acc = acc.map( b  => b['methodName'] )
                    }
                } catch( e ) {
                    console.log( e )
                    acc = []
                }
                return acc
            }, raw )

        if( struct['class'] === null II struct['class'] === undefined ) {
            messages.push( &nbsp;smartContractClass "${smartContractClassName}": is not found&nbsp; )
        }

        if( messages.length !== 0 ) {
            messages
                .forEach( ( msg, index, all ) => {
                    index === 0 ? console.log( &nbsp;Load Smart Contract: Input error&nbsp; ) : ''
                    console.log( &nbsp;  - ${msg}&nbsp; )

                    if( index === all.length -1 ) {
                        let url = ''
                        url += this.#config['docs']['url']
                        url += this.#config['docs']['deployContract']
                        url += '#' + this.#config['docs']['options']

                        console.log( &nbsp;  For more information visit: ${url}&nbsp;)
                    }
                } )
            process.exit( 1 )
        } else {
            this.#contract = struct
        }

        return true
    }


    #deploySave( { data } ) {
        let path = ''
        path += this.#config['environment']['addresses']['contracts']['fullFolder']
        path += data['meta']['fileName']

        if( !fs.existsSync( path ) ) {
            fs.writeFileSync(
                path,
                JSON.stringify( data, null, 4 ),
                'utf-8'
            )
        }

        return path
    }


    #deploySavePrepare( { struct } ) {
        const result = {
            'meta': {
```

```
        'fileName': null,
        'easyMinaVersion': null
    },
    'data': {
        'name': '',
        'type': 'contract',
        'chain': null,
        'time': {
            'unix': null,
            'format': null
        },
        'feePayer': {
            'name': '',
            'public': null,
            'explorer': null
        },
        'address': {
            'public': null,
            'private': null,
            'explorer': null
        },
        'verificationKey': null,
        'smartContract': {
            'className': null,
            'content': null
        },
        'transaction': {
            'hash': null,
            'explorer': null
        }
    },
    'comment': null
}

result['meta']['fileName'] = [
    [ '{{name}}', this.#config['meta']['name'] ],
    [ '{{splitter}}', this.#config['environment']['addresses']['splitter'] ],
    [ '{{unix}}', this.#config['meta']['unix'] ]
]
    .reduce( ( acc, a, index ) => {
        ( index === 0 ) ? acc = this.#config['environment']['addresses']['contracts']['fileNameStruct'] : ''
        acc = acc.replace( a[ 0 ], a[ 1 ] )
        return acc
    }, '' )

const network = this.#config['network']['use']
const n = [
    [ 'meta__easyMinaVersion', this.#config['meta']['easyMinaVersion'] ],
    [ 'data__chain', this.#config['network']['use'] ],
    [ 'data__name', this.#config['meta']['name'] ],
    [ 'data__address__public', struct['destination']['public'] ],
    [ 'data__address__private', struct['destination']['private'] ],
            [ 'data__address__explorer', &nbsp;${this.#config['network'][ network ]['explorer']
['wallet']}${struct['destination']['public']}&nbsp; ],
            [ 'data__feePayer__name', struct['deployer']['name'] ],
            [ 'data__feePayer__public', struct['deployer']['public'] ],
            [ 'data__feePayer__explorer', &nbsp;${this.#config['network'][ network ]['explorer']
['wallet']}${struct['deployer']['public']}&nbsp; ],
            [ 'comment', this.#config['console']['messages']['accountComment'] ],
            [ 'data__time__unix', moment().unix() ],
            [ 'data__time__format', moment().format() ],
            [ 'data__verificationKey', this.#contract['verificationKey'] ],
            [ 'data__smartContract__className', this.#contract['className'] ],
            [ 'data__smartContract__methods', this.#contract['methods'] ],
            [ 'data__smartContract__content', this.#contract['content'] ],
            [ 'data__transaction__hash', struct['transaction']['hash'] ],
            [ 'data__transaction__explorer', struct['transaction']['explorer'] ],
        ]
            .forEach( a => {
                const [ key, value ] = a
                const keys = key.split( '__' )

                switch( keys.length ) {
                    case 1:
                        result[ keys[ 0 ] ] = value ? value : ''
                        break
                    case 2:
                        result[ keys[ 0 ] ][ keys[ 1 ] ] = value ? value : ''
                        break
                    case 3:
                        result[ keys[ 0 ] ][ keys[ 1 ] ][ keys[ 2 ] ] = value ? value : ''
                        break
                    default:
                        break
                }
            } )

        return result
    }


    async #deployPrepare( { accountPath, smartContractPath } ) {

        const struct = {
            'deployer': {
                'source': null,
                'name': null,
                'private': null,
                'public': null,
                'encodedPrivate': null
            },
            'destination': {
                'source': null,
                'private': null,
        'public': null,
        'encodedPrivate': null
    },
    'transaction': {
        'status': null,
        'fee': null,
        'hash': null,
        'explorer': null
    }
}

const m = [
    [ 'deployer', 'deployers' ],
    [ 'destination', 'deployers' ]
]
    .forEach( keys => {
        const [ one, two ] = keys
        struct[ one ]['source'] = accountPath
    } )

struct['transaction']['fee'] =
    this.#config['network'][ this.#config['network']['use'] ]['transaction_fee']

const n = [
    'deployer',
    'destination'
]
    .forEach( key => {
        let _private
        switch( key ) {
            case 'deployer':
                const tmp = fs.readFileSync( struct['deployer']['source'], 'utf-8' )
                const json = JSON.parse( tmp )
                _private = json['data']['address']['private']
                struct['deployer']['name'] = json['meta']['fileName']
                break
            case 'destination':
                _private = this.#snarkyjs.PrivateKey
                    .random()
                    .toBase58()
                break
            default:
                console.log( &nbsp;Key: ${key} not found&nbsp; )
                process.exit( 1 )
                break
        }

        struct[ key ]['private'] = _private

        struct[ key ]['encodedPrivate'] = this.#snarkyjs.PrivateKey
            .fromBase58( struct[ key ]['private'] )
            struct[ key ]['public'] = struct[ key ]['encodedPrivate']
                .toPublicKey()
                .toBase58()
        } )

    return struct
}


async #addVerificationKey() {
    const compiled = await this.#contract['class'].compile()
    this.#contract['verificationKey'] = compiled['verificationKey']
    return true
}


async #deploySend( { struct } ) {
    struct['transaction']['status'] = 'failed'
    try {
        const zkApp = new this.#contract['class'](
            struct['destination']['encodedPrivate'].toPublicKey()
        )

        const deployTxn = await this.#snarkyjs.Mina.transaction(
            {
                'feePayerKey': struct['deployer']['encodedPrivate'],
                'fee': struct['transaction']['fee']
            },
            () => {
                this.#snarkyjs.AccountUpdate
                    .fundNewAccount( struct['deployer']['encodedPrivate'] )

                zkApp.deploy( {
                    'zkappKey': struct['destination']['encodedPrivate'],
                    'verificationKey': this.#contract['verificationKey'],
                    'zkAppUri': 'hello-world'
                } )

                zkApp.init(
                    struct['destination']['encodedPrivate']
                )
            }
        )

        const response =  await deployTxn
            .sign( [
                struct['deployer']['encodedPrivate'],
                struct['destination']['encodedPrivate']
            ] )
            .send()

        struct['transaction']['status'] = 'success'
```

```
            struct['transaction']['hash'] = await response.hash()

            struct['transaction']['explorer'] = ''
            struct['transaction']['explorer'] +=
                this.#config['network'][ this.#config['network']['use'] ]['explorer']['transaction']
            struct['transaction']['explorer'] += struct['transaction']['hash']
        } catch( e ) {
            console.log( &nbsp;Error: Deploy Send ${e}&nbsp; )
            process.exit( 1 )
        }

        return struct
    }
}
```

</file>

<file>

## path: /src/environment/Workspace.mjs

url: https://github.com/EasyMina/easyMina/blob/main/src/environment/Workspace.mjs

```
/*
    Name
        Workspace.mjs
    Description
        This class creates the workspace folders.
    Blocks
        {
            'environment': { ...config['environment'] },
            'meta': { ...config['meta'] },
            'typescript': { ...config['typescript'] },
            'validations': { ...config['validations'] }
        }
    Public
        Variables
        Methods
            .init( { config } )
            .start() (async)
*/


import fs from 'fs'
import axios from 'axios'
import { keyPathToValue } from './../helpers/mixed.mjs'


export class Workspace {
    #config

    constructor() {}
```

```
  init( { config={} } ) {
      this.#config = { ...config }

      return true
  }


  async start() {
      this.#addWorkDir()

      await this.#checkWorkFiles()
      this.#addGitIgnore()
      this.#addConfig()
      // await this.#addTemplate()

      return true
  }


  #addGitIgnore() {
      // process.stdout.write( '  Gitignore             ' )

      let exists = true
      let correct = false

      const path = this.#config['environment']['workspace']['gitignore']
      if( !fs.existsSync( path ) ) {
          exists = false
          fs.writeFileSync(
              path,
              &nbsp;${this.#config['environment']['addresses']['root']}&nbsp;,
              'utf-8'
          )
      } else {
          const raw = fs.readFileSync( path, 'utf-8' )
          const test = raw
              .split( "\n" )
              .map( a => a.trim() )
              .map( a => a === this.#config['environment']['addresses']['root'] )
              .some( a => a )

          if( !test ) {
              console.log( &nbsp;Please insert in ${path} following line: ${this.#config['environment']['addresses']
['root']}&nbsp;)
              process.exit( 1 )
          } else {
              correct = true
          }
      }
```

```
    let msg = ''
    msg += '   ├─── .gitignore'
    msg += exists ? '' : '*'
    msg += correct ? &nbsp; (${this.#config['environment']['addresses']['root']} included)&nbsp; : ''
    console.log( msg )

    return true
}


#addConfig() {
    // console.log( '  Typescript            ')
    // process.stdout.write( '    Config              ' )

    const path = this.#config['typescript']['fileName']
    let exists = true

    if( !fs.existsSync( path ) ) {
        exists = false
        fs.writeFileSync(
            path,
            JSON.stringify( this.#config['typescript']['template'], null, 4 ),
            'utf-8'
        )
    }

    let msg = ''
    msg += '   └─── tsconfig.json'
    msg += exists ? '' : '*'

    console.log( msg )

    return true
}


#addWorkDir() {
    // process.stdout.write( '    Folder              ' )

    let exists = true

    const result = [
        'typescript', 'build'
    ]
        .reduce( ( acc, key, index ) => {
            const dir = this.#config['environment']['workspace']['contracts'][ key ]['full']
            if( !fs.existsSync( dir ) ) {
                exists = false
                fs.mkdirSync( dir, { 'recursive': true } )
            }

            return acc
            }, {} )


        let msg = ''
        msg += '   ├── '
        msg += &nbsp;${this.#config['environment']['workspace']['contracts']['root']}&nbsp;
        msg += exists ? '' : '*'
        console.log( msg )

        return true
    }


    async #checkWorkFiles() {
        const keys = [ 'typescript', 'build' ]
        for( const key of keys ) {
            const dir = this.#config['environment']['workspace']['contracts'][ key ]['full']
            const files = fs
                .readdirSync( dir )
                .filter( ( file ) => !file.startsWith( '.' ) )
            const folder = this.#config['environment']['workspace']['contracts'][ key ]['folder']
            const str = &nbsp;   |     ├── ${folder} (${files.length})&nbsp;
            console.log( str )
            if( key === 'typescript' ) {
                await this.#addTemplate()
            }
        }
/*
            .join( ', ' )

        let msg = ''
        msg += '      '
        // msg += result !== '' ? 'Found! ' : ''
        msg += result
        console.log( msg )
*/
        return true
    }



    async #addTemplate() {
        // process.stdout.write( '    Template            ' )

        const cmd = this.#addTemplatePrepare()

        let path = ''
        path += this.#config['environment']['workspace']['contracts']['typescript']['full']
        path += cmd['fileName']

        let exists = true
        if( !fs.existsSync( path ) ) {
```

```
        exists = false
        const content = await this.#setTemplateContent( { cmd } )
        await this.#addTemplateStore( { path, content } )
    }

    let msg = ''
    msg += '   |     |     └──── '
    msg += exists ? '' : '*'
    msg += &nbsp;${cmd['fileName']}&nbsp;
    console.log( msg )

    return true
}


#addTemplatePrepare() {
    const keys = [ '{{one}}', '{{two}}', '{{three}}', '{{four}}' ]
    const result = {
        'type': null,
        'url': null,
        'content': null,
        'fileName': null
    }

    const vars = Object
        .entries( this.#config['environment']['template']['regexs'] )
        .reduce( ( acc, a, index ) => {
            const [ key, value ] = a

            const regex = keyPathToValue( {
                'data': this.#config['validations']['regexs'],
                'keyPath': value
            } )

            const content = this.#config['environment']['template']['source']['content']
            const matches = content.match( regex['regex'] )

            if( matches ) {
                result['type'] = key
                matches
                    .forEach( ( match, rindex ) =>
                        acc[ keys[ rindex ] ] = match
                    )
            }
            return acc
        }, {} )

    result['fileName'] = this.#config['environment']['workspace']['contracts']['typescript']['fileName']

    result['url'] = Object
        .entries( vars )
        .reduce( ( acc, a, index ) => {
            const [ key, value ] = a
            index === 0 ? acc = this.#config['environment']['template']['parse'][ result['type'] ] : ''
            acc = acc.replace( key, value )
            return acc
        }, '' )

    return result
}


async #setTemplateContent( { cmd } ) {
    let fileContent = ''
    try {
        switch( cmd['type'] ) {
            case 'gist':
                const raw1 = await this.#request( { 'url': cmd['url'] } )
                const fileKey1 = Object.keys( raw1.data.files )[ 0 ]
                fileContent = raw1.data.files[ fileKey1 ].content
                break
            case 'https':
                const raw2 = await this.#request( { 'url': cmd['url'] } )
                const fileKey2 = Object.keys( raw2.data.files )[ 0 ]
                fileContent = raw2.data.files[ fileKey2 ].content
                break
            case 'plain':
                fileContent = this.#config['environment']['template']['source']['content']
                break
            default:
                console.log( &nbsp;Add Template: ${cmd} someting went wrong&nbsp;)
                break
        }
    } catch( e ) {
        console.error( 'Error parsings the file:', e )
    }

    return fileContent
}


#addTemplateStore( { path, content } ) {
    fs.writeFileSync( path, content, 'utf-8' )

    return true
}


async #request( { url } ) {
    let fileContent = ''
    let response = null
    try {
      response = await axios.get( url )
    } catch ( e ) {
            console.error( 'Error fetching the file:', e.message )
        }

        return response
    }
}
```

</file>

<file>

## path: /src/helpers/PrintConsole.mjs

url: https://github.com/EasyMina/easyMina/blob/main/src/helpers/PrintConsole.mjs

```
/*
    Name
        PrintConsole.mjs
    Description
        Here, internationalized messages can be output to the console. Additionally, a variety of formatting
options have been implemented.
    Blocks
        {
            'print': { ...config['print'] },
            'messages': { ...config['messages'] },
            // 'network': { ...config['network'] } optional
        }
    Public
        Variables
        Methods
            .init( { config } )
            .print( { vars={}, key, status=null } ) (async)
            .printQuestionExperimental() (async)
*/


import readline from 'readline'


export class PrintConsole {
    #state
    #config
    #readline

    constructor() {
        this.#config = {}
        return true
    }


    init( { config } ) {
        this.#config = { ...config }
    this.#state = {
        'index': 0,
        'levels': {
            '0': 0,
            '1': 0,
            '2': 0,
            '3': 0
        },
        'userInteractions': {
            'input': null,
            'output': null
        },
        'status': 'progress',
        'question': {
            'use': false,
            'pointer': null
        },
        'lastLevel': 0
    }

    this.#readline = readline.createInterface( {
        'input': process.stdin,
        'output': process.stdout
    } )

    this.#setStructCmds()

    return true
}


async print( { vars={}, key, status=null } ) {
    if( !Object.hasOwn( this.#config['print']['structs'], key ) ) {
        const language = this.#config['messages']['use']

        const msg = this.#config['messages'][ language ]['errorKeyNotFound'][ 0 ]
            .replace( this.#config['print']['insertManual'], key )

        console.log( msg )
        console.log()
        process.exit( 1 )
        return true
    }

    !Object.hasOwn( vars, 'levels' ) ? vars['levels'] = 0 : ''
    key !== 'newLine' ? this.#updateLevels( { vars, status } ) : ''

    this.#setStatus( { status, key } )


    const struct = this.#messageStruct( { vars, key } )
    this.#messagePrintStandard( { vars, struct } )
    status === 'progress' ? this.#state['index']++ : this.#state['index'] = 0
    return true
}


async printQuestionExperimental() {
    this.#setQuestion( { key } )
    const r = await this.#messagePrintQuestion( { vars, struct } )

    const str = r
        .filter( a => a['status'] === 'finished' )[ 0 ]['str']

    console.log( str )
    console.log()

    return true
}


#setQuestion( { key } ) {
    if( this.#config['print']['structs'][ key ]['question'] !== null ) {
        this.#state['question']['use'] = true
        this.#state['question']['pointer'] = this.#config['print']['structs'][ key ]['question']
    } else {
        this.#state['question']['use'] = false
    }

    return true
}


#setStatus( { status, key } ) {
    if( this.#config['print']['structs'][ key ]['question'] !== null ) {
        this.#state['status'] = 'progress'
    } else {
        if( status !== null ) {
            this.#state['status'] = status
        }
    }

    return true
}


#updateLevels( { vars } ) {
    const status = this.#state['status']
    if( Object.hasOwn( vars, 'levels' ) && ( status === 'finished' || status === 'failed' ) ) {
        let reset = []
        let plusOne = []

        switch( vars['levels'] ) {
            case 0:
                plusOne = [ '0' ]
                break
            case 1:
                if( this.#state['lastLevel'] > vars['levels'] ) {
                    plusOne = [ '1' ]
                } else if( this.#state['lastLevel'] < vars['levels'] ) {
                }

                reset = [ '2', '3' ]
                break
            case 2:
                if( this.#state['lastLevel'] === vars['levels'] ) {
                    plusOne = [ '2' ]
                }

                break
            case 3:
                break
            default:
                console.log( &nbsp;Level: ${levels} is not usable&nbsp; )
                break
        }


        this.#state['lastLevel'] = vars['levels']

        plusOne.forEach( key => this.#state['levels'][ key ]++ )
        reset.forEach( key => this.#state['levels'][ key ] = 0 )
    }

    return true
 }


#questionPrint( { question } ) {
    return new Promise( ( resolve, reject ) => {
        this.#readline.question(
            question,
            ( answer ) => {
                resolve( answer )
            }
        )
    } )
}


async #questionValidate( { vars, str } ) {
    const [ type, key ] = this.#getTypeKey( {
        'str': this.#state['question']['pointer']
    } )
```

```
this.#state['userInteractions']['input'] = str + ''
let selection = {
    'status': 'failed',
    ...this.#config['print'][ type ][ key ]['failed']
}

this.#config['print'][ type ][ key ]['validations']
    .forEach( ( a, index ) => {
        const [ _type, _key ] = this.#getTypeKey( {
            'str': a['validation']
        } )

        switch( _type ) {
            case 'regexs':
                const regex = this.#config['print'][ _type ][ _key ]
                const check = this.#state['userInteractions']['input'].match( regex )
                if( check !== null ) {
                    if( check.length === 1 ) {
                        selection = {
                            'status': 'finished',
                            ...a['finished']
                        }
                    }
                }
                break
            case 'messages':
                break
        }

        return true
    } )

this.#state['status'] = selection['status']
switch( this.#state['status'] ) {
    case 'progress':
        break
    case 'finished':
        const [ _type, _key ] =  this.#getTypeKey( {
            'str': selection['output']
        } )

        this.#state['userInteractions']['output'] = this.#getVariables( {
            vars,
            'type': _type,
            'key': _key
        } )
        break
    case 'failed':
        this.#state['userInteractions']['output'] = selection['output']
        break
}
```

```
    const format = selection['format']
    const cmds = this.#structCreateCmds( { format } )
    const result = this.#messageStruct( { vars, key, cmds } )
        .join( '' )

    return {
        'status':  this.#state['status'],
        'str': result
    }
}


#messagePrintStandard( { vars, struct } ) {
    if( this.#state['index'] > 0 ) {
        this.#clearLine()
    }

    const str = struct.join( '' )
    let result = ''
    console.log( str )

    return true
}


async #messagePrintQuestion( { vars, struct, acc=[] } ) {
    const str = struct.join( '' )
    const input = await this.#questionPrint( { 'question': str } )

    this.#clearLine()
    let result = await this.#questionValidate( { vars, 'str': input } )
    acc.push( result )

    if( this.#state['status'] === 'failed' ) {
        this.#state['status'] = 'progress'
        await this.#messagePrintQuestion( { vars, struct, acc } )
    }

    return acc
}


#clearLine() {
    process.stdout.moveCursor( 0, -1 )
    process.stdout.clearLine( 1 )
}


#setStructCmds() {
    const language = this.#config['messages']['language']

    this.#config['print']['structs'] = Object
```

```
        .entries( this.#config['print']['structs'] )
        .reduce( ( acc, a, index ) => {
            const [ key, value ] = a
            acc[ key ] = {}
            acc[ key ]['question'] = value['question']
            acc[ key ]['format'] = value['format']
            acc[ key ]['cmds'] = Object
                .entries( value['format'] )
                .reduce( ( abb, b, rindex ) => {
                    const [ _key, format ] = b
                    abb[ _key ] = this.#structCreateCmds( { format } )
                    return abb
                }, {} )
            return acc
        }, {} )

    return true
}


#structCreateCmds( { format } ) {
    format = format + ''
    let matches = format
        .match( this.#config['print']['regexs']['format'] )

    matches === null ? matches = [ '' ] : ''

    const results = matches
        .reduce( ( abb, tag, rindex, all ) => {
            const tmp = {
                'before': '',
                'tag': tag,
                'after': ''
            }

            if( rindex === 0 ) {
                tmp['before'] = format.substring( 0, format.indexOf( tmp['tag'] ) )
                if( all.length === 1 ) {
                    const start2 = format.indexOf( tmp['tag'] ) + tmp['tag'].length
                    tmp['after'] = format.substring( start2, format.length )
                }
            } else if( rindex === all.length - 1 ) {
                const start2 = format.indexOf( tmp['tag'] ) + tmp['tag'].length
                const start = format.indexOf( all[ rindex - 1 ] ) +  all[ rindex - 1 ].length
                tmp['before'] = format.substring( start, format.indexOf( tmp['tag'] ) )
                tmp['after'] = format.substring( start2, format.length )
            } else {
                const start = format.indexOf( all[ rindex - 1 ] ) +  all[ rindex - 1 ].length
                tmp['before'] = format.substring( start, format.indexOf( tmp['tag'] ) )
            }

            Object
```

```
                .entries( tmp )
                .forEach( c => {
                    const [ key, value ] = c
                    const struct = {
                        'type': key === 'tag' ? 'cmd': 'str',
                        'value': value
                    }

                    struct['value'] !== '' ? abb.push( struct ) : ''
                } )

            return abb
        }, [] )

    return results
}


#messageStruct( { vars, key, cmds=null } ) {
    const status = this.#state['status']
    cmds === null ? cmds = this.#config['print']['structs'][ key ]['cmds'][ status ] : ''
    let acc = ''
    const transforms = cmds
        .map( ( a, index, all ) => {
            let str
            switch( a['type'] ) {
                case 'cmd':
                    const [ _type, _key ] = this.#getTypeKey( {
                        'str': a['value']
                    } )

                    const transform = this.#getVariables( {
                        vars,
                        'type': _type,
                        'key': _key,
                        index,
                        acc
                    } )

                    str = transform
                    break
                case 'str':
                    str = a['value']
                    break
            }
            acc += str
            return str
        } )

    return transforms
}
```

```
#getVariables( { vars, type, key, acc, recursive=0 } ) {
    const status = this.#state['status']
    const language = this.#config['messages']['use']
    let result
    switch( type ) {
        case 'spaces':
            const tmp = this.#config['print']['spaces'][ key ] - acc.length
            const spaces = tmp < 0 ? 0 : tmp

            result = new Array( spaces )
                .fill( '' )
                .join( ' ' )
            break
        case 'status':
            result = this.#config['print']['status'][ key ][ status ][
                this.#state['index'] % this.#config['print']['status'][ key ][ status ].length
            ]
            break
        case 'levels':
            const level = vars['levels']
            const format = this.#config['print']['levels'][ vars['levels'] ]
            result = this.#structCreateCmds( { 'format': format } )
                .map( a => {
                    if( a['type'] === 'cmd' ) {
                        const [ _type, _key ] = this.#getTypeKey( {
                            'str': a['value']
                        } )

                        let str
                        if( this.#state['levels'][ _key ] > this.#config['print']['enumerations'][ _key ].length - 1 ) {
                            str = this.#config['print']['enumerations']['notFound']
                        } else {
                            str = this.#config['print']['enumerations'][ _key ][ this.#state['levels'][ _key ] ]
                        }

                        return str
                    } else {
                        return a['value']
                    }
                } )
                .join( '' )
            break
        case 'messages':
            let message = ''
            try{
                message = this.#config['messages'][ language ][ key ][
                    this.#state['index'] % this.#config['messages'][ language ][ key ].length
                ]
            } catch( e ) {
                console.log( &nbsp;messages__${key} not found&nbsp; )
                console.log( )
        }

        if( recursive === 0 ) {
            result = this
                .#structCreateCmds( { 'format': message } )
                .map( a => {
                    let modified = ''
                    switch( a['type'] ) {
                        case 'str':
                            modified = a['value']
                            break
                        case 'cmd':
                            const [ _type, _key ] = this.#getTypeKey( {
                                'str': a['value']
                            } )

                            modified = this.#getVariables( {
                                vars,
                                'type': _type,
                                'key': _key,
                                'recursive': 1
                            } )
                            break
                        default:
                            break
                    }
                    return modified
                } )
                .join( '' )
        } else {
            result = 'n/a'
        }

        break
    case 'external':
        if( vars[ key ] === undefined ) {
            const msg = this.#config['messages'][ language ]['errorKeyNotFound'][ 0 ]
                .replace( this.#config['print']['insertManual'], key )

            console.log( msg )
            console.log()
            result = ''
        } else {
            result = vars[ key ]
        }
        break
    case 'custom':
        switch( key ) {
            case 'networkExplorerTransaction':
                const network = this.#config['network']['use']
                result = this.#config['network'][ network ]['explorer']['transaction']
                break
                default:
                    const msg = this.#config['messages'][ language ]['errorKeyNotFound'][ 0 ]
                        .replace( this.#config['print']['insertManual'], key )

                    console.log( msg )
                    console.log()
                    break
            }
            break
        case 'userInteractions':
            result = this.#state[ type ][ key ]
            break
        default:
            const msg = this.#config['messages'][ language ]['errorTypeNotFound'][ 0 ]
                .replace( this.#config['print']['insertManual'], type )

            console.log( 'abc>>', msg )
            console.log()
            break
    }

    return result
}


#getTypeKey( { str } ) {
    const result = str
        .replace( '{{', '' )
        .replace( '}}', '' )
        .split( this.#config['print']['split'] )

    return result
}
}
```

</file>

<file>

## path: /src/helpers/mixed.mjs

url: https://github.com/EasyMina/easyMina/blob/main/src/helpers/mixed.mjs

```
/*
    Name
        mixed.mjs
    Description
        Here, various additional functionalities are stored.
    Public
        findClosestString( { input, keys } )
        keyPathToValue( { data, keyPath, separator='__' } )
```

```
*/

function findClosestString( { input, keys } ) {
    function distance( a, b ) {
        let dp = Array( a.length + 1 )
                .fill( null )
                .map( () => Array( b.length + 1 )
                .fill( 0 )
            )
            .map( ( z, index, all ) => {
                index === 0 ? z = z.map( ( y, rindex ) => rindex ) : ''
                z[ 0 ] = index
                return z
            } )

        dp = dp
            .map( ( z, i ) => {
                return z.map( ( y, j ) => {
                    if( i > 0 && j > 0 ) {
                        if( a[ i - 1 ] === b[ j - 1 ] ) {
                            y = dp[ i - 1 ][ j - 1 ]
                        } else {
                            const min = Math.min(
                                dp[ i - 1 ][ j ],
                                dp[ i ][ j - 1 ],
                                dp[ i - 1 ][ j - 1 ]
                            )
                            y = 1 + min
                        }
                    }
                    return y
                } )
            } )

        return dp[ a.length ][ b.length ]
    }

    const result = keys
        .reduce( ( acc, key, index ) => {
            const currentDistance = distance( input, key )
            if( index === 0 ) {
                acc = {
                    'closestKey': key,
                    'closestDistance': currentDistance
                }
            }

            if( currentDistance < acc['closestDistance'] ) {
                acc['closestKey'] = key;
                acc['closestDistance'] = currentDistance;
            }
        return acc
    }, {} )

  return result['closestKey']
}


function keyPathToValue( { data, keyPath, separator='__' } ) {
  if( typeof keyPath !== 'string' ) {
    return undefined
  }

  const result = keyPath
    .split( separator )
    .reduce( ( acc, key, index ) => {
      if( !acc ) return undefined
      if( !acc.hasOwnProperty( key ) ) return undefined
      acc = acc[ key ]
      return acc
    }, data )

  return result
}


export { findClosestString, keyPathToValue }
```

</file>

<file>

## path: /src/interactions/GraphQl.mjs

url: https://github.com/EasyMina/easyMina/blob/main/src/interactions/GraphQl.mjs

```
/*
  Name
    GraphQl.mjs
  Description
    This class establishes a connection to GraphQL and simplifies data interpretation.
  Blocks
    {
      'graphQl': { ...config['graphQl'] },
      'messages': { ...config['messages'] },
      'network': { ...config['network'] },
      'print': { ...config['print'] }
    }
  Public
    Variables
    Methods
      .init( { config } )
```

```
          .payload( { cmd, vars } )
          .request( { payload } )
          .waitForSignal( { cmd, vars } )
*/


import axios from 'axios'
import { PrintConsole } from '../helpers/PrintConsole.mjs'
import moment from 'moment'


export class GraphQl {
    #config
    #template
    #state

    constructor() {}


    init( { config={} } ) {
        this.#config = { ...config }

        this.printConsole = new PrintConsole()
        this.printConsole.init( {
            'config': {
                'print': { ...this.#config['print'] },
                'messages': { ...this.#config['messages'] }
            }
        } )

        this.#state = {
            'latestBlock': null
        }

        return true
    }


    payload( { cmd, vars } ) {
        const network = this.#config['network']['use']
        const url = this.#config['network'][ network ]['graphQl']
        const data = { ...this.#config['graphQl'][ cmd ]['cmd'] }

        Object
            .entries( vars )
            .forEach( a => {
                const [ _key, _value ] = a
                data['variables'][ _key ] = _value
            } )

        const struct = {
            'cmd': cmd,
            'key': this.#config['graphQl'][ cmd ]['key'],
            'type': this.#config['graphQl'][ cmd ]['type'],
            'vars': data['variables'],
            'axios': {
                'method': 'post',
                'maxBodyLength': Infinity,
                'url': url,
                'headers': {
                    'Content-Type': 'application/json',
                    'Accept': 'application/json'
                },
                'data': JSON.stringify( data )
            }
        }

        return struct
    }


    async request( { payload } ) {
        const result = {
            'status': false,
            'message': null,
            'response': null
        }

        try {
            const response = await axios.request( payload['axios'] )
            result['status'] = true
            result['message'] = 'success'
            result['response'] = response['data']
        } catch( e ) {
            result['message'] = 'error'
            console.log( 'Error!', e )
        }

        return result
    }


    async waitForSignal( { cmd, vars } ) {
        const before = this.payload( { 'cmd': 'latestBlockHeights', 'vars': { 'limit': 1 } } )
        this.#state['latestBlock'] = await this.request( { 'payload': before } )

        const payload = this.payload( { cmd, vars } )

        let msg = ' '
        msg += this.#config['network'][ this.#config['network']['use'] ]['explorer']['transaction']
        msg += vars['hash']


        let tmp = null
        let loop = true
        let result = null

        let first = true
        let lastRequest = new Date()
        while( loop ) {
            const endTime = new Date()
            const diff = endTime - lastRequest

            this.printConsole.print(
                {
                    'status': 'progress',
                    'vars': { 'levels': 3, 'front': 'Waiting', 'progress': this.#predictNextSlot() },
                    'key': 'standard'
                }
            )

            if( first || diff > this.#config['graphQl']['render']['delayBetweenRequests'] ) {
                first = false

                this.printConsole.print(
                    {
                        'status': 'progress',
                        'vars': { 'levels': 3, 'front': 'Waiting', 'progress': this.#predictNextSlot() },
                        'key': 'standard'
                    }
                )

                lastRequest = new Date()
                result = await this.#waitForSingleRequest( { payload } )
                result['found'] ? loop = false : ''
            }

            await new Promise( resolve =>
                setTimeout( resolve, this.#config['graphQl']['render']['frameInterval'] )
            )
        }

        this.printConsole.print(
            {
                'status': 'finished',
                'vars': { 'levels': 3, 'front': 'Transaction', 'progress': msg  },
                'key': 'standard'
            }
        )

        return result
    }


    async #waitForSingleRequest( { payload } ) {
        return new Promise( ( outerResolve, outerReject ) => {
```

```
let promiseCompleted = false
let result
let status = 'failed'
let loop = true

const promise = new Promise(
    async ( resolve, reject ) => {
        try {
            result = await this.#requestParse( { payload } )
            result['found'] ? status = 'finished' : ''
            promiseCompleted = true
            // process.stdout.write( &nbsp;${result['response']}&nbsp; )

            this.printConsole.print(
                {
                    'status': 'progress',
                    'vars': { 'levels': 3, 'front': 'Waiting', 'progress': this.#predictNextSlot() },
                    'key': 'standard'
                }
            )

            resolve( 'Promise completed!' )
        } catch( error ) {
            reject( error )
        }
    })

const interval = setInterval(
    async () => {
        if ( promiseCompleted ) {
            clearInterval( interval )

            this.printConsole.print(
                {
                    'status': 'progress',
                    'vars': { 'levels': 3, 'front': 'Waiting', 'progress': this.#predictNextSlot() },
                    'key': 'standard'
                }
            )

            loop = false
            outerResolve( result )
        } else {
            this.printConsole.print(
                {
                    'status': 'progress',
                    'vars': { 'levels': 3, 'front': 'Waiting', 'progress': this.#predictNextSlot() },
                    'key': 'standard'
                }
            )
        }
    },
            this.#config['graphQl']['render']['frameInterval']
        )

    } )
}


async #requestParse( { payload } ) {
    let result = {
        'error': false,
        'found': false,
        'response': null,
        'vars': { ...payload['vars'] }
    }

    try {
        const raw = await this.request( { payload } )
        const response = raw['response']['data']
        const key = payload['key']
        switch( payload['type'] ) {
            case 'hash':
                if( response[ key ] !== null ) {
                    result['response'] = response

                    Object
                        .entries( response[ key ] )
                        .forEach( a => {
                            const [ key, value ] = a
                            result['vars'][ key ] = value
                        } )

                    result['found'] = true
                    break
                }
                break
            case 'array':
                if( response[ key ].length !== 0 ) {
                    result['response'] = response
                    result['found'] = true
                }

                result['vars']['count'] = response[ key ].length
                break
            default:
                console.log( 'type not found' )
                break
        }

    } catch( e ) {
        result['error'] = true
    }
```

```
        return result
    }


    #predictNextSlot() {
        const response = this.#state['latestBlock']
        const result = {
            'lastBlock': null,
            'offsetSlot': null,
            'slotNumber': null,
            'nextSlot': null,
            'timeUntilNextSlot': null
        }

        try {
            result['lastBlock'] = response['response']['data']['blocks'][ 0 ]['dateTime']
            result['slotNumber'] = response['response']['data']['blocks'][ 0 ]['protocolState']['consensusState']['slot']
        } catch( e ) {
            return result
        }

        const slotStartTime = moment( result['lastBlock'] )
        const now = moment()
        const slotIntervalMilliseconds = 3 * 60 * 1000
        const timeElapsedMilliseconds = now.diff( slotStartTime )

        let slotNumberOffset = Math.floor( timeElapsedMilliseconds / slotIntervalMilliseconds ) + 1
        slotNumberOffset = ( slotNumberOffset <= 0 ) ? 1 : slotNumberOffset

        const remainingTimeMilliseconds = slotIntervalMilliseconds - ( timeElapsedMilliseconds %
slotIntervalMilliseconds )
        const remainingTimeInSeconds = Math.round( remainingTimeMilliseconds / 1000 )

        const formattedRemainingTime = &nbsp;${String( Math.floor( remainingTimeInSeconds / 60 )
).padStart( 2, '0' )}:${String( remainingTimeInSeconds % 60 ).padStart( 2, '0' )}&nbsp;

        result['offsetSlot'] = slotNumberOffset
        result['nextSlot'] = result['slotNumber'] + slotNumberOffset + 1
        result['timeUntilNextSlot'] = formattedRemainingTime

        const format = &nbsp;${result['timeUntilNextSlot']} (${result['nextSlot']})&nbsp;

        return format
    }
}
```

</file>

<file>

## path: /test/circle-ci.mjs

url: https://github.com/EasyMina/easyMina/blob/main/test/circle-ci.mjs

```
import { EasyMina } from './../src/EasyMina.mjs'

const easyMina = new EasyMina()
const test = easyMina.health()

console.log( 'Test', test )
if( test !== null || test !== undefined ) {
    process.exit( 0 )
} else {
    process.exit( 1 )
}
```

</file>

</repository>

# o1js

## overview

repository: https://github.com/o1-labs/o1js/
userName: o1-labs
repository: o1js
branch: main
date: 2023-11-10T14:03:44+01:00 (1699621424)

<file>

## path: /.eslintignore

url: https://github.com/o1-labs/o1js/blob/main/.eslintignore

```
{
  "ignorePatterns": [
    "src/bindings/compiled/web_bindings",
    "src/bindings/compiled/node_bindings",
    "node_modules",
    "dist"
  ]
}
```

</file>

<file>

## path: /.eslintrc.json

url: https://github.com/o1-labs/o1js/blob/main/.eslintrc.json

```
{
  "env": {
    "browser": true,
    "es2021": true,
    "node": true
  },
  "extends": "eslint:recommended",
  "parserOptions": {
    "ecmaVersion": 13,
    "sourceType": "module"
  },
  "rules": {
    "no-unused-vars": "warn",
    "no-constant-condition": "off",
    "no-empty": "off",
    "no-unused-labels": "off"
  }
}
```

</file>

<file>

## path: /.prettierignore

url: https://github.com/o1-labs/o1js/blob/main/.prettierignore

```
src/bindings/compiled/web_bindings/**/plonk_wasm*
src/bindings/compiled/web_bindings/**/*.bc.js
src/bindings/compiled/node_bindings/*
dist/**/*
src/bindings/kimchi/js/**/*.js
```

</file>

<file>

## path: /.prettierrc.cjs

url: https://github.com/o1-labs/o1js/blob/main/.prettierrc.cjs

```
module.exports = {
  trailingComma: 'es5',
  tabWidth: 2,
  semi: true,
  singleQuote: true,
};
```

</file>

<file>

## path: /CHANGELOG.md

url: https://github.com/o1-labs/o1js/blob/main/CHANGELOG.md

# Changelog

All notable changes to this project are documented in this file.

The format is based on [Keep a Changelog (https://keepachangelog.com/en/1.0.0/)](https://keepachangelog.com/en/1.0.0/).
This project adheres to [Semantic Versioning (https://semver.org/spec/v2.0.0.html)](https://semver.org/spec/v2.0.0.html).

<!--
Possible subsections:
*Added* for new features.
*Changed* for changes in existing functionality.
*Deprecated* for soon-to-be removed features.
*Removed* for now removed features.
*Fixed* for any bug fixes.
*Security* in case of vulnerabilities.

-->

## [Unreleased (https://github.com/o1-labs/o1js/compare/26363465d...HEAD)](https://github.com/o1-labs/o1js/compare/26363465d...HEAD)

> No unreleased changes yet

## [0.14.1 (https://github.com/o1-labs/o1js/compare/e8e7510e1...26363465d)](https://github.com/o1-labs/o1js/compare/e8e7510e1...26363465d)

### Added

- Gadgets.not(), new provable method to support bitwise shifting for native field elements. https://github.com/o1-labs/o1js/pull/1198
- Gadgets.leftShift() / Gadgets.rightShift(), new provable method to support bitwise shifting for native field elements. https://github.com/o1-labs/o1js/pull/1194
- Gadgets.and(), new provable method to support bitwise and for native field elements. https://github.com/o1-labs/o1js/pull/1193
- Gadgets.multiRangeCheck() and Gadgets.compactMultiRangeCheck(), two building blocks for non-native arithmetic with bigints of size up to 264 bits. https://github.com/o1-labs/o1js/pull/1216

## [0.14.0 (https://github.com/o1-labs/o1js/compare/045faa7...e8e7510e1)](https://github.com/o1-labs/o1js/compare/045faa7...e8e7510e1)

### Breaking changes

- Constraint optimizations in Field methods and core crypto changes break all verification keys https://github.com/o1-labs/o1js/pull/1171 https://github.com/o1-labs/o1js/pull/1178

### Changed

- ZkProgram has moved out of the Experimental namespace and is now available as a top-level import directly. Experimental.ZkProgram has been deprecated.
- ZkProgram gets a new input argument name: string which is required in the non-experimental API. The name is used to identify a ZkProgram when caching prover keys. https://github.com/o1-labs/o1js/pull/1200

### Added

- Lightnet namespace to interact with the account manager provided by the lightnet Mina network (https://hub.docker.com/r/o1labs/mina-local-network). https://github.com/o1-labs/o1js/pull/1167
- Internal support for several custom gates (range check, bitwise operations, foreign field operations) and lookup tables https://github.com/o1-labs/o1js/pull/1176
- Gadgets.rangeCheck64(), new provable method to do efficient 64-bit range checks using lookup tables https://github.com/o1-labs/o1js/pull/1181
- Gadgets.rotate(), new provable method to support bitwise rotation for native field elements. https://github.com/o1-labs/o1js/pull/1182
- Gadgets.xor(), new provable method to support bitwise xor for native field elements. https://github.com/o1-labs/o1js/pull/1177
- Proof.dummy() to create dummy proofs https://github.com/o1-labs/o1js/pull/1188
  - You can use this to write ZkPrograms that handle the base case and the inductive case in the same method.

### Changed

- Use cached prover keys in compile() when running in Node.js https://github.com/o1-labs/o1js/pull/1187
  - Caching is configurable by passing a custom Cache (new export) to compile()
  - By default, prover keys are stored in an OS-dependent cache directory; ~/.cache/pickles on Mac and Linux
- Use cached setup points (SRS and Lagrange bases) when running in Node.js https://github.com/o1-labs/o1js/pull/1197
  - Also, speed up SRS generation by using multiple threads
  - Together with caching of prover keys, this speeds up compilation time by roughly
    - 86% when everything is cached
    - 34% when nothing is cached

## 0.13.1 (https://github.com/o1-labs/o1js/compare/c2f392fe5...045faa7)

### Breaking changes

- Changes to some verification keys caused by changing the way Struct orders object properties. https://github.com/o1-labs/o1js/pull/1124 @Comdex (https://github.com/Comdex)
  - To recover existing verification keys and behavior, change the order of properties in your Struct definitions to be alphabetical

    - The customObjectKeys option is removed from Struct

### Changed

- Improve prover performance by ~25% https://github.com/o1-labs/o1js/pull/1092
    - Change internal representation of field elements to be JS bigint instead of Uint8Array
- Consolidate internal framework for testing equivalence of two implementations

## 0.13.0 (https://github.com/o1-labs/o1js/compare/fbd4b2717...c2f392fe5)

### Breaking changes

- Changes to verification keys caused by updates to the proof system. This breaks all deployed contracts https://github.com/o1-labs/o1js/pull/1016

## 0.12.2 (https://github.com/o1-labs/o1js/compare/b1d8d5910...fbd4b2717)

### Changed

- Renamed SnarkyJS to o1js https://github.com/o1-labs/o1js/pull/1104
- Reduce loading time of the library by 3-4x https://github.com/o1-labs/o1js/pull/1073
- Improve error when forgetting transaction.prove() https://github.com/o1-labs/o1js/pull/1095

## 0.12.1 (https://github.com/o1-labs/o1js/compare/161b69d602...b1d8d5910)

### Added

- Added a method createTestNullifier to the Nullifier class for testing purposes. It is recommended to use mina-signer to create Nullifiers in production, since it does not leak the private key of the user. The Nullifier.createTestNullifier method requires the private key as an input *outside of the users wallet*. https://github.com/o1-labs/o1js/pull/1026
- Added field.isEven to check if a Field element is odd or even. https://github.com/o1-labs/o1js/pull/1026

### Fixed

- Revert verification key hash change from previous release to stay compatible with the current testnet https://github.com/o1-labs/o1js/pull/1032

## 0.12.0 (https://github.com/o1-labs/o1js/compare/eaa39dca0...161b69d602)

### Breaking Changes

- Fix the default verification key hash that was generated for AccountUpdates. This change adopts the default mechanism provided by Mina Protocol https://github.com/o1-labs/o1js/pull/1021

    - Please be aware that this alteration results in a breaking change affecting the verification key of already deployed contracts.

## 0.11.4 (https://github.com/o1-labs/o1js/compare/544489609...eaa39dca0)

### Fixed

- NodeJS error caused by invalid import https://github.com/o1-labs/o1js/issues/1012

## 0.11.3 (https://github.com/o1-labs/o1js/compare/2d2af219c...544489609)

### Fixed

- Fix commonJS version of o1js, again https://github.com/o1-labs/o1js/pull/1006

## 0.11.2 (https://github.com/o1-labs/o1js/compare/c549e02fa...2d2af219c)

### Fixed

- Fix commonJS version of o1js https://github.com/o1-labs/o1js/pull/1005

## 0.11.1 (https://github.com/o1-labs/o1js/compare/3fbd9678e...c549e02fa)

### Breaking changes

- Group operations now generate a different set of constraints. This breaks deployed contracts, because the circuit changed. https://github.com/o1-labs/o1js/pull/967

### Added

- Implemented Nullifier as a new primitive https://github.com/o1-labs/o1js/pull/882
    - mina-signer can now be used to generate a Nullifier, which can be consumed by zkApps using the newly added Nullifier Struct

### Changed

- Improve error message Can't evaluate prover code outside an as_prover block https://github.com/o1-labs/o1js/pull/998

### Fixed

- Fix unsupported use of window when running o1js in workers https://github.com/o1-labs/o1js/pull/1002

## 0.11.0 (https://github.com/o1-labs/o1js/compare/a632313a...3fbd9678e)

### Breaking changes

- Rewrite of Provable.if() causes breaking changes to all deployed contracts https://github.com/o1-labs/o1js/pull/889
- Remove all deprecated methods and properties on Field https://github.com/o1-labs/o1js/pull/902
- The Field(x) constructor and other Field methods no longer accept a boolean as input. Instead, you can now pass in a bigint to all Field methods. https://github.com/o1-labs/o1js/pull/902
- Remove redundant signFeePayer() method https://github.com/o1-labs/o1js/pull/935

### Added

- Add field.assertNotEquals() to assert that a field element does not equal some value https://github.com/o1-labs/o1js/pull/902
    - More efficient than field.equals(x).assertFalse()
- Add scalar.toConstant(), scalar.toBigInt(), Scalar.from(), privateKey.toBigInt(), PrivateKey.fromBigInt() https://github.com/o1-labs/o1js/pull/935
- Poseidon.hashToGroup enables hashing to a group https://github.com/o1-labs/o1js/pull/887

### Changed

- Make stack traces more readable https://github.com/o1-labs/o1js/pull/890
    - Stack traces thrown from o1js are cleaned up by filtering out unnecessary lines and other noisy details
- Remove optional zkappKey argument in smartContract.init(), and instead assert that provedState is false when init() is called https://github.com/o1-labs/o1js/pull/908
- Improve assertion error messages on Field methods https://github.com/o1-labs/o1js/issues/743 https://github.com/o1-labs/o1js/pull/902
- Publicly expose the internal details of the Field type https://github.com/o1-labs/o1js/pull/902

### Deprecated

- Utility methods on Circuit are deprecated in favor of the same methods on Provable https://github.com/o1-labs/o1js/pull/889
    - Circuit.if(), Circuit.witness(), Circuit.log() and others replaced by Provable.if(), Provable.witness(), Provable.log()
    - Under the hood, some of these methods were rewritten in TypeScript
- Deprecate field.isZero() https://github.com/o1-labs/o1js/pull/902

### Fixed

- Fix running o1js in Node.js on Windows https://github.com/o1-labs/o1js-bindings/pull/19 @wizicer (https://github.com/wizicer)
- Fix error reporting from GraphQL requests https://github.com/o1-labs/o1js/pull/919
- Resolved an Out of Memory error experienced on iOS devices (iPhones and iPads) during the initialization of the WASM memory https://github.com/o1-labs/o1js-bindings/pull/26
- Fix field.greaterThan() and other comparison methods outside provable code https://github.com/o1-labs/o1js/issues/858 https://github.com/o1-labs/o1js/pull/902
- Fix field.assertBool() https://github.com/o1-labs/o1js/issues/469 https://github.com/o1-labs/o1js/pull/902
- Fix Field(bigint) where bigint is larger than the field modulus https://github.com/o1-

labs/o1js/issues/432 https://github.com/o1-labs/o1js/pull/902

  - The new behaviour is to use the modular residual of the input

- No longer fail on missing signature in tx.send(). This fixes the flow of deploying a zkApp from a UI via a wallet https://github.com/o1-labs/o1js/pull/931 [@marekyggdrasil (https://github.com/marekyggdrasil)](https://github.com/marekyggdrasil)

## [0.10.1 (https://github.com/o1-labs/o1js/compare/bcc666f2...a632313a)](https://github.com/o1-labs/o1js/compare/bcc666f2...a632313a)

### Changed

- Allow ZkPrograms to return their public output https://github.com/o1-labs/o1js/pull/874 https://github.com/o1-labs/o1js/pull/876
  - new option ZkProgram({ publicOutput?: Provable<any>, ... }); publicOutput has to match the *return type* of all ZkProgram methods.
  - the publicInput option becomes optional; if not provided, methods no longer expect the public input as first argument
  - full usage example: https://github.com/o1-labs/o1js/blob/f95cf2903e97292df9e703b74ee1fc3825df826d/src/examples/program.ts

## [0.10.0 (https://github.com/o1-labs/o1js/compare/97e393ed...bcc666f2)](https://github.com/o1-labs/o1js/compare/97e393ed...bcc666f2)

### Breaking Changes

- All references to actionsHash are renamed to actionState to better mirror what is used in Mina protocol APIs https://github.com/o1-labs/o1js/pull/833

  - This change affects function parameters and returned object keys throughout the API

- No longer make MayUseToken.InheritFromParent the default mayUseToken value on the caller if one zkApp method calls another one; this removes the need to manually override mayUseToken in several known cases https://github.com/o1-labs/o1js/pull/863

  - Causes a breaking change to the verification key of deployed contracts that use zkApp composability

### Added

- this.state.getAndAssertEquals() as a shortcut for let x = this.state.get(); this.state.assertEquals(x); https://github.com/o1-labs/o1js/pull/863

  - also added .getAndAssertEquals() on this.account and this.network fields

- Support for fallback endpoints when making network requests, allowing users to provide an array of endpoints for GraphQL network requests. https://github.com/o1-labs/o1js/pull/871

  - Endpoints are fetched two at a time, and the result returned from the faster response

- reducer.forEach(actions, ...) as a shortcut for reducer.reduce() when you don't need a state https://github.com/o1-labs/o1js/pull/863
- New export TokenId which supersedes Token.Id; TokenId.deriveId() replaces Token.Id.getId() https://github.com/o1-labs/o1js/pull/863

- Add Permissions.allImpossible() for the set of permissions where nothing is allowed (more convenient than Permissions.default() when you want to make most actions impossible) https://github.com/o1-labs/o1js/pull/863

### Changed

- Massive improvement of memory consumption, thanks to a refactor of o1js' worker usage https://github.com/o1-labs/o1js/pull/872
  - Memory reduced by up to 10x; see [the PR (https://github.com/o1-labs/o1js/pull/872)](https://github.com/o1-labs/o1js/pull/872) for details
  - Side effect: Circuit API becomes async, for example MyCircuit.prove(...) becomes await MyCircuit.prove(...)
- Token APIs this.token.{send,burn,mint}() now accept an AccountUpdate or SmartContract as from / to input https://github.com/o1-labs/o1js/pull/863
- Improve Transaction.toPretty() output by adding account update labels in most methods that create account updates https://github.com/o1-labs/o1js/pull/863
- Raises the limit of actions/events per transaction from 16 to 100, providing users with the ability to submit a larger number of events/actions in a single transaction. https://github.com/o1-labs/o1js/pull/883.

### Deprecated

- Deprecate both shutdown() and await isReady, which are no longer needed https://github.com/o1-labs/o1js/pull/872

### Fixed

- SmartContract.deploy() now throws an error when no verification key is found https://github.com/o1-labs/o1js/pull/885
  - The old, confusing behaviour was to silently not update the verification key (but still update some permissions to "proof", breaking the zkApp)

## [0.9.8 (https://github.com/o1-labs/o1js/compare/1a984089...97e393ed)](https://github.com/o1-labs/o1js/compare/1a984089...97e393ed)

### Fixed

- Fix fetching the access permission on accounts https://github.com/o1-labs/o1js/pull/851
- Fix fetchActions https://github.com/o1-labs/o1js/pull/844 https://github.com/o1-labs/o1js/pull/854 [@Comdex (https://github.com/Comdex)](https://github.com/Comdex)
- Updated Mina.TransactionId.isSuccess to accurately verify zkApp transaction status after using Mina.TransactionId.wait(). https://github.com/o1-labs/o1js/pull/826
  - This change ensures that the function correctly checks for transaction completion and provides the expected result.

## [0.9.7 (https://github.com/o1-labs/o1js/compare/0b7a9ad...1a984089)](https://github.com/o1-labs/o1js/compare/0b7a9ad...1a984089)

### Added

- smartContract.fetchActions() and Mina.fetchActions(), asynchronous methods to fetch actions directly from an archive node https://github.com/o1-labs/o1js/pull/843 [@Comdex (https://github.com/Comdex)](https://github.com/Comdex)

### Changed

- Circuit.runAndCheck() now uses snarky to create a constraint system and witnesses, and check constraints. It closely matches behavior during proving and can be used to test provable code without having to create an expensive proof https://github.com/o1-labs/o1js/pull/840

### Fixed

- Fixes two issues that were temporarily reintroduced in the 0.9.6 release https://github.com/o1-labs/o1js/issues/799 https://github.com/o1-labs/o1js/issues/530

## [0.9.6 (https://github.com/o1-labs/o1js/compare/21de489...0b7a9ad)](https://github.com/o1-labs/o1js/compare/21de489...0b7a9ad)

### Breaking changes

- Circuits changed due to an internal rename of "sequence events" to "actions" which included a change to some hash prefixes; this breaks all deployed contracts.
- Temporarily reintroduces 2 known issues as a result of reverting a fix necessary for network redeployment:
  - https://github.com/o1-labs/o1js/issues/799
  - https://github.com/o1-labs/o1js/issues/530
  - Please note that we plan to address these issues in a future release. In the meantime, to work around this breaking change, you can try calling fetchAccount for each account involved in a transaction before executing the Mina.transaction block.
- Improve number of constraints needed for Merkle tree hashing https://github.com/o1-labs/o1js/pull/820
  - This breaks deployed zkApps which use MerkleWitness.calculateRoot(), because the circuit is changed
  - You can make your existing contracts compatible again by switching to MerkleWitness.calculateRootSlow(), which has the old circuit
- Renamed function parameters: The getAction function now accepts a new object structure for its parameters. https://github.com/o1-labs/o1js/pull/828
  - The previous object keys, fromActionHash and endActionHash, have been replaced by fromActionState and endActionState.

### Added

- zkProgram.analyzeMethods() to obtain metadata about a ZkProgram's methods https://github.com/o1-labs/o1js/pull/829 [@maht0rz (https://github.com/maht0rz)](https://github.com/maht0rz)

### Fixed

- Improved Event Handling in o1js https://github.com/o1-labs/o1js/pull/825
  - Updated the internal event type to better handle events emitted in different zkApp transactions and when multiple zkApp transactions are present within a block.

    - The internal event type now includes event data and transaction information as separate objects, allowing for more accurate information about each event and its associated transaction.

- Removed multiple best tip blocks when fetching action data https://github.com/o1-labs/o1js/pull/817
    - Implemented a temporary fix that filters out multiple best tip blocks, if they exist, while fetching actions. This fix will be removed once the related issue in the Archive-Node-API repository (https://github.com/o1-labs/Archive-Node-API/issues/7) is resolved.

- New fromActionState and endActionState parameters for fetchActions function in o1js https://github.com/o1-labs/o1js/pull/828
    - Allows fetching only necessary actions to compute the latest actions state
    - Eliminates the need to retrieve the entire actions history of a zkApp
    - Utilizes actionStateTwo field returned by Archive Node API as a safe starting point for deriving the most recent action hash

## 0.9.5 (https://github.com/o1-labs/o1js/compare/21de489...4573252d)

- Update the zkApp verification key from within one of its own methods, via proof https://github.com/o1-labs/o1js/pull/812

### Breaking changes

- Change type of verification key returned by SmartContract.compile() to match VerificationKey https://github.com/o1-labs/o1js/pull/812

### Fixed

- Failing Mina.transaction on Berkeley because of unsatisfied constraints caused by dummy data before we fetched account state https://github.com/o1-labs/o1js/pull/807
    - Previously, you could work around this by calling fetchAccount() for every account invovled in a transaction. This is not necessary anymore.
- Update the zkApp verification key from within one of its own methods, via proof https://github.com/o1-labs/o1js/pull/812

## 0.9.4 (https://github.com/o1-labs/o1js/compare/9acec55...21de489)

### Fixed

- getActions to handle multiple actions with multiple Account Updates https://github.com/o1-labs/o1js/pull/801

## 0.9.3 (https://github.com/o1-labs/o1js/compare/1abdfb70...9acec55)

### Added

- Use fetchEvents() to fetch events for a specified zkApp from a GraphQL endpoint that implements this schema (https://github.com/o1-labs/Archive-Node-API/blob/efebc9fd3cfc028f536ae2125e0d2676e2b86cd2/src/schema.ts#L1). Mina.Network accepts an additional endpoint which points to a GraphQL server. https://github.com/o1-labs/o1js/pull/749

  - Use the mina property for the Mina node.
  - Use archive for the archive node.

- Use getActions to fetch actions for a specified zkApp from a GraphQL endpoint GraphQL endpoint that implements the same schema as fetchEvents. https://github.com/o1-labs/o1js/pull/788

### Fixed

- Added the missing export of Mina.TransactionId https://github.com/o1-labs/o1js/pull/785
- Added an option to specify tokenId as Field in fetchAccount() https://github.com/o1-labs/o1js/pull/787 [@rpanic (https://github.com/rpanic)](https://github.com/rpanic)

## [0.9.2 (https://github.com/o1-labs/o1js/compare/9c44b9c2...1abdfb70)](https://github.com/o1-labs/o1js/compare/9c44b9c2...1abdfb70)

### Added

- this.network.timestamp is added back and is implemented on top of this.network.globalSlotSinceGenesis https://github.com/o1-labs/o1js/pull/755

### Changed

- On-chain value globalSlot is replaced by the clearer currentSlot https://github.com/o1-labs/o1js/pull/755
  - currentSlot refers to the slot at which the transaction *will be included in a block*.
  - the only supported method is currentSlot.assertBetween() because currentSlot.get() is impossible to implement since the value is determined in the future and currentSlot.assertEquals() is error-prone

### Fixed

- Incorrect counting of limit on events and actions https://github.com/o1-labs/o1js/pull/758
- Type error when using Circuit.array in on-chain state or events https://github.com/o1-labs/o1js/pull/758
- Bug when using Circuit.witness outside the prover https://github.com/o1-labs/o1js/pull/774

## [0.9.1 (https://github.com/o1-labs/o1js/compare/71b6132b...9c44b9c2)](https://github.com/o1-labs/o1js/compare/71b6132b...9c44b9c2)

### Fixed

- Bug when using this.<state>.get() outside a transaction https://github.com/o1-labs/o1js/pull/754

## [0.9.0 (https://github.com/o1-labs/o1js/compare/c5a36207...71b6132b)](https://github.com/o1-labs/o1js/compare/c5a36207...71b6132b)

### Added

- Transaction.fromJSON to recover transaction object from JSON https://github.com/o1-labs/o1js/pull/705
- New precondition: provedState, a boolean which is true if the entire on-chain state of this account was last modified by a proof https://github.com/o1-labs/o1js/pull/741

- Same API as all preconditions: this.account.provedState.assertEquals(Bool(true))
    - Can be used to assert that the state wasn't tampered with by the zkApp developer using non-contract logic, for example, before deploying the zkApp

- New on-chain value globalSlot, to make assertions about the current time https://github.com/o1-labs/o1js/pull/649

    - example: this.globalSlot.get(), this.globalSlot.assertBetween(lower, upper)
    - Replaces network.timestamp, network.globalSlotSinceGenesis and network.globalSlotSinceHardFork. https://github.com/o1-labs/o1js/pull/560

- New permissions:

    - access to control whether account updates for this account can be used at all https://github.com/o1-labs/o1js/pull/500
    - setTiming to control who can update the account's timing field https://github.com/o1-labs/o1js/pull/685
    - Example: this.permissions.set({ ...Permissions.default(), access: Permissions.proofOrSignature() })

- Expose low-level view into the PLONK gates created by a smart contract method https://github.com/o1-labs/o1js/pull/687

    - MyContract.analyzeMethods().<method name>.gates

## Changed

- BREAKING CHANGE: Modify signature algorithm used by Signature.{create,verify} to be compatible with mina-signer https://github.com/o1-labs/o1js/pull/710

    - Signatures created with mina-signer's client.signFields() can now be verified inside a SNARK!
    - Breaks existing deployed smart contracts which use Signature.verify()

- BREAKING CHANGE: Circuits changed due to core protocol and cryptography changes; this breaks all deployed contracts.
- BREAKING CHANGE: Change structure of Account type which is returned by Mina.getAccount() https://github.com/o1-labs/o1js/pull/741

    - for example, account.appState -> account.zkapp.appState
    - full new type (exported as Types.Account): https://github.com/o1-labs/o1js/blob/0be70cb8ceb423976f348980e9d6238820758cc0/src/provable/gen/transactic

- Test accounts hard-coded in LocalBlockchain now have default permissions, not permissions allowing everything. Fixes some unintuitive behaviour in tests, like requiring no signature when using these accounts to send MINA https://github.com/o1-labs/o1js/issues/638

## Removed

- Preconditions timestamp and globalSlotSinceHardFork https://github.com/o1-labs/o1js/pull/560

    - timestamp is expected to come back as a wrapper for the new globalSlot

## [0.8.0 (https://github.com/o1-labs/o1js/compare/d880bd6e...c5a36207)](https://github.com/o1-labs/o1js/compare/d880bd6e...c5a36207)

## Added

- this.account.<field>.set() as a unified API to update fields on the account https://github.com/o1-labs/o1js/pull/643
    - covers permissions, verificationKey, zkappUri, tokenSymbol, delegate, votingFor
    - exists on SmartContract.account and AccountUpdate.account
- this.sender to get the public key of the transaction's sender https://github.com/o1-labs/o1js/pull/652
    - To get the sender outside a smart contract, there's now Mina.sender()
- tx.wait() is now implemented. It waits for the transactions inclusion in a block https://github.com/o1-labs/o1js/pull/645
    - wait() also now takes an optional options parameter to specify the polling interval or maximum attempts. wait(options?: { maxAttempts?: number; interval?: number }): Promise<void>;
- Circuit.constraintSystemFromKeypair(keypair) to inspect the circuit at a low level https://github.com/o1-labs/o1js/pull/529
    - Works with a keypair (prover + verifier key) generated with the Circuit API
- Mina.faucet() can now be used to programmatically fund an address on the testnet, using the faucet provided by faucet.minaprotocol.com https://github.com/o1-labs/o1js/pull/693

## Changed

- BREAKING CHANGE: Constraint changes in sign(), requireSignature() and createSigned() on AccountUpdate / SmartContract. *This means that smart contracts using these methods in their proofs won't be able to create valid proofs against old deployed verification keys.* https://github.com/o1-labs/o1js/pull/637
- Mina.transaction now takes a *public key* as the fee payer argument (passing in a private key is deprecated) https://github.com/o1-labs/o1js/pull/652
    - Before: Mina.transaction(privateKey, ...). Now: Mina.transaction(publicKey, ...)
    - AccountUpdate.fundNewAccount() now enables funding multiple accounts at once, and deprecates the initialBalance argument
- New option enforceTransactionLimits for LocalBlockchain (default value: true), to disable the enforcement of protocol transaction limits (maximum events, maximum sequence events and enforcing certain layout of AccountUpdates depending on their authorization) https://github.com/o1-labs/o1js/pull/620
- Change the default send permissions (for sending MINA or tokens) that get set when deploying a zkApp, from signature() to proof() https://github.com/o1-labs/o1js/pull/648
- Functions for making assertions and comparisons have been renamed to their long form, instead of the initial abbreviation. Old function names have been deprecated https://github.com/o1-labs/o1js/pull/681
    - .lt -> .lessThan
    - .lte -> .lessThanOrEqual
    - .gt -> .greaterThan
    - .gte -> greaterThanOrEqual
    - .assertLt -> .assertLessThan

    - .assertLte -> .assertLessThanOrEqual
    - .assertGt -> .assertGreaterThan
    - .assertGte -> assertGreaterThanOrEqual
    - .assertBoolean -> .assertBool

### Deprecated

- this.setPermissions() in favor of this.account.permissions.set() https://github.com/o1-labs/o1js/pull/643
    - this.tokenSymbol.set() in favor of this.account.tokenSymbol.set()
    - this.setValue() in favor of this.account.<field>.set()
- Mina.transaction(privateKey: PrivateKey, ...) in favor of new signature Mina.transaction(publicKey: PublicKey, ...)
- AccountUpdate.createSigned(privateKey: PrivateKey) in favor of new signature AccountUpdate.createSigned(publicKey: PublicKey) https://github.com/o1-labs/o1js/pull/637
- .lt, .lte, gt, gte, .assertLt, .assertLte, .assertGt, .assertGte have been deprecated. https://github.com/o1-labs/o1js/pull/681

### Fixed

- Fixed Apple silicon performance issue https://github.com/o1-labs/o1js/issues/491
- Type inference for Structs with instance methods https://github.com/o1-labs/o1js/pull/567
    - also fixes Struct.fromJSON
- SmartContract.fetchEvents fixed when multiple event types existed https://github.com/o1-labs/o1js/issues/627
- Error when using reduce with a Struct as state type https://github.com/o1-labs/o1js/pull/689
- Fix use of stale cached accounts in Mina.transaction https://github.com/o1-labs/o1js/issues/430

## 0.7.3 (https://github.com/o1-labs/o1js/compare/5f20f496...d880bd6e)

### Fixed

- Bug in deploy() when initializing a contract that already exists https://github.com/o1-labs/o1js/pull/588

### Deprecated

- Mina.BerkeleyQANet in favor of the clearer-named Mina.Network https://github.com/o1-labs/o1js/pull/588

## 0.7.2 (https://github.com/o1-labs/o1js/compare/705f58d3...5f20f496)

### Added

- MerkleMap and MerkleMapWitness https://github.com/o1-labs/o1js/pull/546
- Lots of doc comments! https://github.com/o1-labs/o1js/pull/580

### Fixed

- Bug in Circuit.log printing account updates https://github.com/o1-labs/o1js/pull/578

## [0.7.1 (https://github.com/o1-labs/o1js/compare/f0837188...705f58d3)](https://github.com/o1-labs/o1js/compare/f0837188...705f58d3)

### Fixed

- Testnet-incompatible signatures in v0.7.0 https://github.com/o1-labs/o1js/pull/565

## [0.7.0 (https://github.com/o1-labs/o1js/compare/f0837188...9a94231c)](https://github.com/o1-labs/o1js/compare/f0837188...9a94231c)

### Added

- Added an optional string parameter to certain assert methods https://github.com/o1-labs/o1js/pull/470
- Struct, a new primitive for declaring composite, SNARK-compatible types https://github.com/o1-labs/o1js/pull/416
    - With this, we also added a way to include auxiliary, non-field element data in composite types
    - Added VerificationKey, which is a Struct with auxiliary data, to pass verification keys to a @method
    - BREAKING CHANGE: Change names related to circuit types: AsFieldsAndAux<T> -> Provable<T>, AsFieldElement<T> -> ProvablePure<T>, circuitValue -> provable
    - BREAKING CHANGE: Change all ofFields and ofBits methods on circuit types to fromFields and fromBits
- New option proofsEnabled for LocalBlockchain (default value: true), to quickly test transaction logic with proofs disabled https://github.com/o1-labs/o1js/pull/462
    - with proofsEnabled: true, proofs now get verified locally https://github.com/o1-labs/o1js/pull/423
- SmartContract.approve() to approve a tree of child account updates https://github.com/o1-labs/o1js/pull/428 https://github.com/o1-labs/o1js/pull/534
    - AccountUpdates are now valid @method arguments, and approve() is intended to be used on them when passed to a method
    - Also replaces Experimental.accountUpdateFromCallback()
- Circuit.log() to easily log Fields and other provable types inside a method, with the same API as console.log() https://github.com/o1-labs/o1js/pull/484
- SmartContract.init() is a new method on the base SmartContract that will be called only during the first deploy (not if you re-deploy later to upgrade the contract) https://github.com/o1-labs/o1js/pull/543
    - Overriding init() is the new recommended way to add custom state initialization logic.
- transaction.toPretty() and accountUpdate.toPretty() for debugging transactions by printing only the pieces that differ from default account updates https://github.com/o1-labs/o1js/pull/428
- AccountUpdate.attachToTransaction() for explicitly adding an account update to the current transaction. This replaces some previous behaviour where an account update got attached implicitly https://github.com/o1-labs/o1js/pull/484
- SmartContract.requireSignature() and AccountUpdate.requireSignature() as a simpler, better-

named replacement for .sign() https://github.com/o1-labs/o1js/pull/558

### Changed

- BREAKING CHANGE: tx.send() is now asynchronous: old: send(): TransactionId new: send(): Promise<TransactionId> and tx.send() now directly waits for the network response, as opposed to tx.send().wait() https://github.com/o1-labs/o1js/pull/423
- Sending transactions to LocalBlockchain now involves
- Circuit.witness can now be called outside circuits, where it will just directly return the callback result https://github.com/o1-labs/o1js/pull/484
- The FeePayerSpec, which is used to specify properties of the transaction via Mina.transaction(), now has another optional parameter to specify the nonce manually. Mina.transaction({ feePayerKey: feePayer, nonce: 1 }, () => {}) https://github.com/o1-labs/o1js/pull/497
- BREAKING CHANGE: Static methods of type .fromString(), .fromNumber() and .fromBigInt() on Field, UInt64, UInt32 and Int64 are no longer supported https://github.com/o1-labs/o1js/pull/519
    - use Field(number | string | bigint) and UInt64.from(number | string | bigint)
- Move several features out of 'experimental' https://github.com/o1-labs/o1js/pull/555
    - Reducer replaces Experimental.Reducer
    - MerkleTree and MerkleWitness replace Experimental.{MerkleTree,MerkleWitness}
    - In a SmartContract, this.token replaces this.experimental.token

### Deprecated

- CircuitValue deprecated in favor of Struct https://github.com/o1-labs/o1js/pull/416
- Static props Field.zero, Field.one, Field.minusOne deprecated in favor of Field(number) https://github.com/o1-labs/o1js/pull/524
- SmartContract.sign() and AccountUpdate.sign() in favor of .requireSignature() https://github.com/o1-labs/o1js/pull/558

### Fixed

- Uint comparisons and division fixed inside the prover https://github.com/o1-labs/o1js/pull/503
- Callback arguments are properly passed into method invocations https://github.com/o1-labs/o1js/pull/516
- Removed internal type JSONValue from public interfaces https://github.com/o1-labs/o1js/pull/536
- Returning values from a zkApp https://github.com/o1-labs/o1js/pull/461

### Fixed

- Callback arguments are properly passed into method invocations https://github.com/o1-labs/o1js/pull/516

## 0.6.1 (https://github.com/o1-labs/o1js/compare/ba688523...f0837188)

### Fixed

- Proof verification on the web version https://github.com/o1-labs/o1js/pull/476

## 0.6.0 (https://github.com/o1-labs/o1js/compare/f2ad423...ba688523)

### Added

- reducer.getActions partially implemented for local testing https://github.com/o1-labs/o1js/pull/327
- gte and assertGte methods on UInt32, UInt64 https://github.com/o1-labs/o1js/pull/349
- Return sent transaction hash for RemoteBlockchain https://github.com/o1-labs/o1js/pull/399

### Changed

- BREAKING CHANGE: Rename the Party class to AccountUpdate. Also, rename other occurrences of "party" to "account update". https://github.com/o1-labs/o1js/pull/393
- BREAKING CHANGE: Don't require the account address as input to SmartContract.compile(), SmartContract.digest() and SmartContract.analyzeMethods() https://github.com/o1-labs/o1js/pull/406
    - This works because the address / public key is now a variable in the method circuit; it used to be a constant
- BREAKING CHANGE: Move ZkProgram to Experimental.ZkProgram

## 0.5.4 (https://github.com/o1-labs/o1js/compare/3461333...f2ad423)

### Fixed

- Running o1js inside a web worker https://github.com/o1-labs/o1js/issues/378

## 0.5.3 (https://github.com/o1-labs/o1js/compare/4f0dd40...3461333)

### Fixed

- Infinite loop when compiling in web version https://github.com/o1-labs/o1js/issues/379, by @mahtOrz (https://github.com/mahtOrz)

## 0.5.2 (https://github.com/o1-labs/o1js/compare/55c8ea0...4f0dd40)

### Fixed

- Crash of the web version introduced in 0.5.0
- Issue with Experimental.MerkleWitness https://github.com/o1-labs/o1js/pull/368

## 0.5.1 (https://github.com/o1-labs/o1js/compare/e0192f7...55c8ea0)

### Fixed

- fetchAccount https://github.com/o1-labs/o1js/pull/350

## 0.5.0 (https://github.com/o1-labs/o1js/compare/2375f08...e0192f7)

### Added

- Recursive proofs. RFC: https://github.com/o1-labs/o1js/issues/89, PRs: https://github.com/o1-labs/o1js/pull/245 https://github.com/o1-labs/o1js/pull/250 https://github.com/o1-labs/o1js/pull/261

  - Enable smart contract methods to take previous proofs as arguments, and verify them in the circuit
  - Add ZkProgram, a new primitive which represents a collection of circuits that produce instances of the same proof. So, it's a more general version of SmartContract, without any of the Mina-related API.
    ZkProgram is suitable for rollup-type systems and offchain usage of Pickles + Kimchi.

- zkApp composability -- calling other zkApps from inside zkApps. RFC: https://github.com/o1-labs/o1js/issues/303, PRs: https://github.com/o1-labs/o1js/pull/285, https://github.com/o1-labs/o1js/pull/296, https://github.com/o1-labs/o1js/pull/294, https://github.com/o1-labs/o1js/pull/297
- Events support via SmartContract.events, this.emitEvent. RFC: https://github.com/o1-labs/o1js/issues/248, PR: https://github.com/o1-labs/o1js/pull/272

  - fetchEvents partially implemented for local testing: https://github.com/o1-labs/o1js/pull/323

- Payments: this.send({ to, amount }) as an easier API for sending Mina from smart contracts https://github.com/o1-labs/o1js/pull/325

  - Party.send() to transfer Mina between any accounts, for example, from users to smart contracts

- SmartContract.digest() to quickly compute a hash of the contract's circuit. This is [used by the zkApp CLI (https://github.com/o1-labs/zkapp-cli/pull/233)](https://github.com/o1-labs/zkapp-cli/pull/233) to figure out whether compile should be re-run or a cached verification key can be used. https://github.com/o1-labs/o1js/pull/268
- Circuit.constraintSystem() for creating a circuit from a function, counting the number of constraints and computing a digest of the circuit https://github.com/o1-labs/o1js/pull/279
- this.account.isNew to assert that an account did not (or did) exist before the transaction https://github.com/MinaProtocol/mina/pull/11524
- LocalBlockchain.setTimestamp and other setters for network state, to test network preconditions locally https://github.com/o1-labs/o1js/pull/329
- Experimental APIs are now collected under the Experimental import, or on this.experimental in a smart contract.
- Custom tokens (*experimental*), via this.token. RFC: https://github.com/o1-labs/o1js/issues/233, PR: https://github.com/o1-labs/o1js/pull/273,
- Actions / sequence events support (*experimental*), via Experimental.Reducer. RFC: https://github.com/o1-labs/o1js/issues/265, PR: https://github.com/o1-labs/o1js/pull/274
- Merkle tree implementation (*experimental*) via Experimental.MerkleTree https://github.com/o1-labs/o1js/pull/343

### Changed

- BREAKING CHANGE: Make on-chain state consistent with other preconditions - throw an error when state is not explicitly constrained https://github.com/o1-labs/o1js/pull/267
- CircuitValue improvements https://github.com/o1-labs/o1js/pull/269, https://github.com/o1-labs/o1js/pull/306, https://github.com/o1-labs/o1js/pull/341

  - Added a base constructor, so overriding the constructor on classes that extend CircuitValue is now *optional*. When overriding, the base constructor can be called without arguments, as previously: super(). When not overriding, the expected arguments are all the @props on the class, in the order they were defined in: new MyCircuitValue(prop1, prop2).
  - CircuitValue.fromObject({ prop1, prop2 }) is a new, better-typed alternative for using the base constructor.

    - Fixed: the overridden constructor is now free to have any argument structure -- previously, arguments had to be the props in their declared order. I.e., the behaviour that's now used by the base constructor used to be forced on all constructors, which is no longer the case.

- Mina.transaction improvements

    - Support zkApp proofs when there are other account updates in the same transaction block https://github.com/o1-labs/o1js/pull/280
    - Support multiple independent zkApp proofs in one transaction block https://github.com/o1-labs/o1js/pull/296

- Add previously unimplemented preconditions, like this.network.timestamp https://github.com/o1-labs/o1js/pull/324 https://github.com/MinaProtocol/mina/pull/11577
- Improve error messages thrown from Wasm, by making Rust's panic log to the JS console https://github.com/MinaProtocol/mina/pull/11644
- Not user-facing, but essential: Smart contracts fully constrain the account updates they create, inside the circuit https://github.com/o1-labs/o1js/pull/278

### Fixed

- Fix comparisons on UInt32 and UInt64 (UInt32.lt, UInt32.gt, etc) https://github.com/o1-labs/o1js/issues/174, https://github.com/o1-labs/o1js/issues/101. PR: https://github.com/o1-labs/o1js/pull/307

## 0.4.3 (https://github.com/o1-labs/o1js/compare/e66f08d...2375f08)

### Added

- Implement the precondition RFC (https://github.com/o1-labs/o1js/issues/179#issuecomment-1139413831):

    - new fields this.account and this.network on both SmartContract and Party
    - this.<account|network>.<property>.get() to use on-chain values in a circuit, e.g. account balance or block height
    - this.<account|network>.<property>.{assertEqual, assertBetween, assertNothing}() to constrain what values to allow for these

- CircuitString, a snark-compatible string type with methods like .append() https://github.com/o1-labs/o1js/pull/155
- bool.assertTrue(), bool.assertFalse() as convenient aliases for existing functionality
- Ledger.verifyPartyProof which can check if a proof on a transaction is valid https://github.com/o1-labs/o1js/pull/208
- Memo field in APIs like Mina.transaction to attach arbitrary messages https://github.com/o1-labs/o1js/pull/244
- This changelog

### Changed

- Huge snark performance improvements (2-10x) for most zkApps https://github.com/MinaProtocol/mina/pull/11053
- Performance improvements in node with > 4 CPUs, for all snarks https://github.com/MinaProtocol/mina/pull/11292
- Substantial reduction of o1js' size https://github.com/MinaProtocol/mina/pull/11166

### Removed

- Unused functions call and callUnproved, which were embryonic versions of what is now the transaction API to call smart contract methods
- Some unimplemented fields on SmartContract

### Fixed

- zkApp proving on web https://github.com/o1-labs/o1js/issues/226
  </file>

<file>

## path: /CODEOWNERS

url: https://github.com/o1-labs/o1js/blob/main/CODEOWNERS

```
/src/lib/gadgets @o1-labs/crypto-eng-reviewers @mitschabaude
```

</file>

<file>

## path: /CONTRIBUTING.md

url: https://github.com/o1-labs/o1js/blob/main/CONTRIBUTING.md

# Contributing guidelines

Thank you for investing your time in contributing to our project.

We also welcome contributions to zkApps Developer (https://docs.minaprotocol.com/zkapps) documentation.

There are two ways to contribute:

1. Preferred: Maintain your own package that can be installed from npm (https://www.npmjs.com/) and used alongside o1js. See Creating high-quality community packages.
2. Directly contribute to this repo. See Contributing to o1js.

If you maintain your own package, we strongly encourage you to add it to our official list of community packages (./README.md#community-packages).

For information that is helpful for o1js core contributors, see README-dev (README-dev.md).

### Creating high-quality community packages

To ensure consistency within the o1js ecosystem and ease review and use by our team and other o1js devs, we encourage community packages to follow these standards:

- The package is published to npm (https://www.npmjs.com/).
    - npm install <your-package> works and is all that is needed to use the package.

  - o1js must be listed as a peer dependency (https://docs.npmjs.com/cli/v9/configuring-npm/package-json#peerdependencies).
  - If applicable, the package must work both on the web and in NodeJS.

- The package is created using the zkApp CLI (https://github.com/o1-labs/zkapp-cli) (recommended). If you did not create the package using the zkApp CLI, follow these guidelines for code consistency:

  - Use TypeScript, and export types from d.ts files. We suggest that you base your tsconfig on the tsconfig.json (./tsconfig.json) that o1js uses.
  - Code must be auto-formatted with prettier (https://prettier.io/). We encourage you to use .prettierrc.cjs (./.prettierrc.cjs), the same prettier config as o1js.

- The package includes tests.

  - If applicable, tests must demonstrate that the package's methods can successfully run as provable code. For example, when the package is used in a SmartContract or ZkProgram that is compiled and proven.
  - Ideally, your tests are easy to use, modify, and port to other projects by developers in the ecosystem. This is achieved by using Jest (see jest.config.js (./jest.config.js) for an example config) or by structuring your tests as plain node scripts, like this example (./src/lib/circuit_value.unit-test.ts).

- Public API must be documented and JSDoc (https://jsdoc.app/) comments must be present on exported methods and globals.
- Include README and LICENSE files.
- Comments and README must be in English and preferably use American spelling.

## Contributing to o1js

The main branch contains the development version of the code.

To contribute directly to this project repo, follow these steps to get your changes in the main branch as quickly as possible.

1. Create a new issue for your proposed changes or use an existing issue if a relevant one exists.

2. Write a request for comment (RFC) to outline your proposed changes and motivation, like this example RFC (https://github.com/o1-labs/o1js/issues/233). Describe your objective and why the change is useful, how it works, and so on.

   Note: If you are proposing a smaller change your RFC will be smaller, and that's ok! :)

3. One of the codebase maintainers reviews your RFC and works with you until it is approved.

4. After your RFC proposal is approved, fork the repository and implement your changes.

5. Submit a pull request and wait for code review. :)

Our goal is to include functionality within o1js when the change is likely to be widely useful for developers. For more esoteric functionality, it makes more sense to provide high-quality community packages that can be used alongside o1js.

We appreciate your contribution!
</file>

<file>

## path: /README-dev.md

url: https://github.com/o1-labs/o1js/blob/main/README-dev.md

# How to contribute to the o1js codebase

This README includes information that is helpful for o1js core contributors.

## Run examples using Node.js

```
npm install
npm run build

./run src/examples/api_exploration.ts
```

## Run examples in the browser

```
npm install
npm run build:web

./run-in-browser.js src/examples/api_exploration.ts
```

To see the test running in a web browser, go to http://localhost:8000/.

Note: Some of our examples don't work on the web because they use Node.js APIs.

## Run tests

- Unit tests

  ```
  npm run test
  npm run test:unit
  ```

- Integration tests

  ```
  npm run test:integration
  ```

- E2E tests

  ```
  npm install
  npm run e2e:install
  npm run build:web

  npm run e2e:prepare-server
  npm run test:e2e
  npm run e2e:show-report
  ```

## Branch Compatibility

o1js is mostly used to write Mina Smart Contracts and must be compatible with the latest Berkeley Testnet, or soon Mainnet.

The OCaml code is in the o1js-bindings repository, not directly in o1js.

To maintain compatibility between the repositories and build o1js from the Mina repository (https://github.com/MinaProtocol/mina), make changes to its core, such as the OCaml-bindings in the o1js-bindings repository (https://github.com/o1-labs/o1js-bindings), you must follow a certain branch compatibility pattern:

The following branches are compatible:

| | |
|---|---|
| repository | mina -> o1js -> o1js-bindings |
| branches | o1js-main -> main -> main |
| | berkeley -> berkeley -> berkeley |
| | develop -> develop -> develop |

## Run the GitHub actions locally

<!-- The test example should stay in sync with a real value set in .github/workflows/build-actions.yml -->

You can execute the CI locally by using act (https://github.com/nektos/act). First generate a GitHub token and use:

```
act -j Build-And-Test-Server --matrix test_type:"Simple integration tests" -s $GITHUB_TOKEN
```

to execute the job "Build-And-Test-Server for the test type Simple integration tests.
</file>

<file>

## path: /README.md

url: https://github.com/o1-labs/o1js/blob/main/README.md

# o1js npm v0.14.1 (https://www.npmjs.com/package/o1js) PRs welcome (https://github.com/o1-labs/o1js/blob/main/CONTRIBUTING.md)

o1js is an evolution of SnarkyJS (https://www.npmjs.com/package/snarkyjs) which saw:
49 updated versions over 2 years of development with 43,141 downloads

This name change reflects the evolution of our vision for the premiere toolkit used by developers to build zero knowledge-enabled applications, while paying homage to our technology's recursive proof generation capabilities.

Your favorite functionality stays the same and transitioning to o1js is a quick and easy process:

- To update zkApp-cli, run the following command:
  npm i -g zkapp-cli@latest
- To remove the now-deprecated SnarkyJs package and install o1js, run the following command:
  npm remove snarkyjs && npm install o1js
- For existing zkApps, make sure to update your imports from snarkyjs to o1js
- No need to redeploy, you are good to go!

## o1js

o1js helps developers build apps powered by zero-knowledge (zk) cryptography.

The easiest way to write zk programs is using o1js.

o1js is a TypeScript library for zk-SNARKs (https://minaprotocol.com/blog/what-are-zk-snarks) and zkApps. You can use o1js to write zk smart contracts based on zero-knowledge proofs for the Mina Protocol.

o1js is automatically included when you create a project using the Mina zkApp CLI (https://github.com/o1-labs/zkapp-cli).

## Learn More

- To learn more about developing zkApps, see the zkApp Developers (https://docs.minaprotocol.com/zkapps) docs.

- For guided steps building and using zkApps, see the zkApp Developers Tutorials (https://docs.minaprotocol.com/zkapps/tutorials/hello-world).

- To meet other developers building zkApps with o1js, participate in the #zkapps-developers (https://discord.com/channels/484437221055922177/915745847692636181) channel on Mina Protocol Discord.

- For a list of changes between versions, see the CHANGELOG (https://github.com/o1-labs/o1js/blob/main/CHANGELOG.md).

- To stay up to date with o1js, see the O(1) Labs Blog (https://blog.o1labs.org/tagged/o1js).

## Contributing

o1js is an open source project. We appreciate all community contributions to o1js!

See the Contributing guidelines (https://github.com/o1-labs/o1js/blob/main/CONTRIBUTING.md) for ways you can contribute.

## Community Packages

High-quality community packages from open source developers are available for your project.

- o1js-elgamal A partially homomorphic encryption library for o1js based on Elgamal encryption: GitHub (https://github.com/Trivo25/o1js-elgamal) and npm (https://www.npmjs.com/package/o1js-elgamal)
- o1js-pack A library for o1js that allows a zkApp developer to pack extra data into a single Field. GitHub (https://github.com/45930/o1js-pack) and npm (https://www.npmjs.com/package/o1js-pack)

To include your package, see [Creating high-quality community packages (https://github.com/o1-labs/o1js/blob/main/CONTRIBUTING.md#creating-high-quality-community-packages)](https://github.com/o1-labs/o1js/blob/main/CONTRIBUTING.md#creating-high-quality-community-packages).
</file>

<file>

## path: /generate-keys.js

url: https://github.com/o1-labs/o1js/blob/main/generate-keys.js

```
#!/usr/bin/env node
import Client from './dist/node/mina-signer/MinaSigner.js';

let client = new Client({ network: 'testnet' });

console.log(client.genKeys());
```

</file>

<file>

## path: /jest

url: https://github.com/o1-labs/o1js/blob/main/jest

```
NODE_OPTIONS=--experimental-vm-modules npx jest $@
```

</file>

<file>

## path: /jest.config.js

url: https://github.com/o1-labs/o1js/blob/main/jest.config.js

```
export default {
  preset: 'ts-jest/presets/js-with-ts',
  testEnvironment: 'node',
  extensionsToTreatAsEsm: ['.ts'],
  transformIgnorePatterns: ['node_modules/', 'dist/node/'],
  globals: {
    'ts-jest': {
      useESM: true,
    },
  },
  testTimeout: 1_000_000,
};
```

</file>

<file>

## path: /package.json

url: https://github.com/o1-labs/o1js/blob/main/package.json

```json
{
  "name": "o1js",
  "description": "TypeScript framework for zk-SNARKs and zkApps",
  "version": "0.14.1",
  "license": "Apache-2.0",
  "homepage": "https://github.com/o1-labs/o1js/",
  "keywords": [
    "mina",
    "zkapp",
    "zk",
    "smart contract",
    "cryptography",
    "blockchain",
    "web3",
    "zk-snark",
    "zero knowledge"
  ],
  "type": "module",
  "main": "./dist/web/index.js",
  "exports": {
    "types": "./dist/node/index.d.ts",
    "browser": "./dist/web/index.js",
    "node": {
      "import": "./dist/node/index.js",
      "require": "./dist/node/index.cjs"
    },
    "default": "./dist/web/index.js"
  },
  "types": "./dist/node/index.d.ts",
  "files": [
    "src/build",
    "dist",
    "src/**/*.ts",
    "src/**/*.d.ts",
    "dist/**/*.map",
    "src/**/*.map"
  ],
  "bin": {
    "snarky-run": "src/build/run.js"
  },
  "engines": {
    "node": ">=16.4.0"
  },
  "scripts": {
    "dev": "npx tsc -p tsconfig.node.json && node src/build/copy-to-dist.js",
    "make": "make -C ../../.. snarkyjs",
    "make:no-types": "npm run clean && make -C ../../.. snarkyjs_no_types",
    "wasm": "./src/bindings/scripts/update-wasm-and-types.sh",
    "bindings": "cd ../../.. && ./scripts/update-snarkyjs-bindings.sh && cd src/lib/snarkyjs",
    "build": "node src/build/copy-artifacts.js && rimraf ./dist/node && npm run dev && node src/build/buildNode.js",
    "build:test": "npx tsc -p tsconfig.test.json && cp src/snarky.d.ts dist/node/snarky.d.ts",
    "build:node": "npm run build",
    "build:web": "rimraf ./dist/web && node src/build/buildWeb.js",
    "build:examples": "rimraf ./dist/examples && npx tsc -p tsconfig.examples.json || exit 0",
    "build:docs": "npx typedoc --tsconfig ./tsconfig.web.json",
    "prepublish:web": "NODE_ENV=production node src/build/buildWeb.js",
    "prepublish:node": "npm run build && NODE_ENV=production node src/build/buildNode.js",
    "prepublishOnly": "npm run prepublish:web && npm run prepublish:node",
    "dump-vks": "./run tests/vk-regression/vk-regression.ts --bundle --dump",
    "format": "prettier --write --ignore-unknown **/*",
    "clean": "rimraf ./dist && rimraf ./src/bindings/compiled/_node_bindings",
    "clean-all": "npm run clean && rimraf ./tests/report && rimraf ./tests/test-artifacts",
    "test": "./run-jest-tests.sh",
    "test:integration": "./run-integration-tests.sh",
    "test:unit": "./run-unit-tests.sh",
    "test:e2e": "rimraf ./tests/report && rimraf ./tests/test-artifacts && npx playwright test",
    "e2e:prepare-server": "npm run build:examples && (cp -rf dist/examples dist/web || :) && node src/build/e2eTestsBuildHelper.js && cp -rf src/examples/plain-html/index.html src/examples/plain-html/server.js tests/artifacts/html/*.html tests/artifacts/javascript/*.js dist/web",
    "e2e:run-server": "node dist/web/server.js",
    "e2e:install": "npx playwright install --with-deps",
    "e2e:show-report": "npx playwright show-report tests/report"
  },
  "author": "O(1) Labs",
  "devDependencies": {
    "@playwright/test": "^1.25.2",
    "@types/isomorphic-fetch": "^0.0.36",
    "@types/jest": "^27.0.0",
    "@types/node": "^18.14.2",
    "@typescript-eslint/eslint-plugin": "^5.0.0",
    "esbuild": "^0.19.2",
    "eslint": "^8.0.0",
    "expect": "^29.0.1",
    "fs-extra": "^10.0.0",
    "glob": "^8.0.3",
    "howslow": "^0.1.0",
    "jest": "^28.1.3",
    "minimist": "^1.2.7",
    "prettier": "^2.8.4",
    "replace-in-file": "^6.3.5",
    "rimraf": "^3.0.2",
    "ts-jest": "^28.0.8",
    "typedoc": "^0.24.8",
    "typedoc-plugin-markdown": "^3.15.3",
    "typedoc-plugin-merge-modules": "^5.0.1",
    "typescript": "5.1"
  },
  "dependencies": {
    "blakejs": "1.2.1",
    "cachedir": "^2.4.0",
    "detect-gpu": "^5.0.5",
    "isomorphic-fetch": "^3.0.0",
    "js-sha256": "^0.9.0",
    "reflect-metadata": "^0.1.13",
    "tslib": "^2.3.0"
  }
}
```

</file>

<file>

## path: /playwright.config.ts

url: https://github.com/o1-labs/o1js/blob/main/playwright.config.ts

```
import type { PlaywrightTestConfig } from '@playwright/test';
import { devices } from '@playwright/test';

/**
 * Read environment variables from file.
 * https://github.com/motdotla/dotenv
 */
// require('dotenv').config();

/**
 * See https://playwright.dev/docs/test-configuration.
 */
const config: PlaywrightTestConfig = {
  testDir: './tests',
  outputDir: './tests/test-artifacts',
  /* Maximum time one test can run for. */
  timeout: 25 * 60 * 1000,
  expect: {
    /**
     * Maximum time expect() should wait for the condition to be met.
     * For example in &nbsp;await expect(locator).toHaveText();&nbsp;
     */
    timeout: 5 * 60 * 1000,
  },
  /* Run tests in files in parallel */
  fullyParallel: true,
  /* Fail the build on CI if you accidentally left test.only in the source code. */
  forbidOnly: !!process.env.CI,
  /* Retry on CI only */
  retries: process.env.CI ? 2 : 0,
  /* Opt out of parallel tests (on CI and locally for now). */
  workers: process.env.CI ? 1 : 1,
  /* Reporter to use. See https://playwright.dev/docs/test-reporters */
  reporter: [
    ['list'],
    ['html', { outputFolder: 'tests/report', open: 'never' }],
  ],
  /* Shared settings for all the projects below. See https://playwright.dev/docs/api/class-testoptions. */
  use: {
    browserName: 'chromium',
    actionTimeout: 0,
    baseURL: 'http://localhost:8000',
    headless: process.env.CI ? true : false,
    viewport: { width: 1280, height: 720 },
    ignoreHTTPSErrors: true,
    screenshot: 'only-on-failure',
    video: 'on-first-retry',
    trace: 'retain-on-failure',
    storageState: './tests/artifacts/config/storageState.json',
  },
  testIgnore: ['*.js'],
  /* Configure projects for major browsers */
  projects: [
    {
      name: 'chromium-desktop',
      use: {
        browserName: 'chromium',
        ...devices['Desktop Chrome'],
      },
    },
    // {
    //   name: 'firefox-desktop',
    //   use: {
    //     browserName: 'firefox',
    //     ...devices['Desktop Firefox'],
    //   },
    // },
  ],
  /* Run your local dev server before starting the tests */
  webServer: {
    command: 'npm run e2e:run-server',
    url: 'http://localhost:8000',
    timeout: 3 * 60 * 1000,
  },
};

export default config;
```

</file>

<file>

## path: /run

url: https://github.com/o1-labs/o1js/blob/main/run

```
node --enable-source-maps --stack-trace-limit=1000 src/build/run.js $@
```

</file>

<file>

## path: /run-ci-tests.sh

url: https://github.com/o1-labs/o1js/blob/main/run-ci-tests.sh

```
#!/bin/bash
set -e

case $TEST_TYPE in
"Simple integration tests")
  echo "Running basic integration tests"
  ./run src/examples/zkapps/hello_world/run.ts --bundle
  ./run src/examples/simple_zkapp.ts --bundle
  ./run src/examples/zkapps/reducer/reducer_composite.ts --bundle
  ./run src/examples/zkapps/composability.ts --bundle
  ;;

"Voting integration tests")
  echo "Running voting integration tests"
  ./run src/examples/zkapps/voting/run.ts --bundle
  ;;

"DEX integration tests")
  echo "Running DEX integration tests"
  ./run src/examples/zkapps/dex/run.ts --bundle
  ./run src/examples/zkapps/dex/upgradability.ts --bundle
  ;;

"DEX integration test with proofs")
  echo "Running DEX integration test with proofs"
  ./run src/examples/zkapps/dex/happy-path-with-proofs.ts --bundle
  ;;

"Live integration tests")
  echo "Running integration tests against real Mina network"
  ./run src/examples/zkapps/hello_world/run_live.ts --bundle
  ;;

"Unit tests")
  echo "Running unit tests"
  cd src/mina-signer
  npm run build
  cd ../..
  npm run test:unit
  npm run test
  ;;

"Verification Key Regression Check")
  echo "Running Regression checks"
  ./run ./tests/vk-regression/vk-regression.ts --bundle
  ;;

"CommonJS test")
  echo "Testing CommonJS version"
  node src/examples/commonjs.cjs
  ;;

*)
  echo "ERROR: Invalid enviroment variable, not clear what tests to run! $CI_NODE_INDEX"
  exit 1
  ;;
esac
```

</file>

<file>

## path: /run-in-browser.js

url: https://github.com/o1-labs/o1js/blob/main/run-in-browser.js

```
#!/usr/bin/env node
import fs from 'node:fs/promises';
import path from 'node:path';
import http from 'node:http';
import minimist from 'minimist';
import { build } from './src/build/buildExample.js';

let {
  _: [filePath],
} = minimist(process.argv.slice(2));

if (!filePath) {
  console.log(&nbsp;Usage:
./run-in-browser [file]&nbsp;);
  process.exit(0);
}

// copy file into /dist/web
let targetDir = &nbsp;./dist/web&nbsp;;

let absPath = await build(filePath, true);
let fileName = path.basename(absPath);
let newPath = &nbsp;${targetDir}/${fileName}&nbsp;;
await fs.copyFile(absPath, newPath);
await fs.unlink(absPath);
console.log(&nbsp;running in the browser: ${newPath}&nbsp;);

const indexHtml = &nbsp;
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <link rel="icon" href="data:," />
    <title>o1js</title>
    <script type="module" src="./${fileName}">
    </script>
  </head>
  <body>
    <div>Check out the console (F12)</div>
  </body>
</html>

&nbsp;;

const port = 8000;
const defaultHeaders = {
  'content-type': 'text/html',
  'Cross-Origin-Embedder-Policy': 'require-corp',
  'Cross-Origin-Opener-Policy': 'same-origin',
};

const server = http.createServer(async (req, res) => {
  let file = '.' + req.url;
  if (file === './') file = './index.html';
  // console.log('serving', file);

  let content;
  if (file === './index.html') content = indexHtml;
  else {
    try {
      content = await fs.readFile(path.resolve('./dist/web', file), 'utf8');
    } catch (err) {
      res.writeHead(404, defaultHeaders);
      res.write('<html><body>404</body><html>');
      res.end();
      return;
    }
  }

  const extension = path.basename(file).split('.').pop();
  const contentType = {
    html: 'text/html',
    js: 'application/javascript',
   map: 'application/json',
  }[extension];
  const headers = { ...defaultHeaders, 'content-type': contentType };

  res.writeHead(200, headers);
  res.write(content);
  res.end();
});

server.listen(port, () => {
  console.log(&nbsp;Server is running on: http://localhost:${port}&nbsp;);
});
```

</file>

<file>

## path: /run-integration-tests.sh

url: https://github.com/o1-labs/o1js/blob/main/run-integration-tests.sh

```
#!/bin/bash
set -e

./run src/examples/zkapps/hello_world/run.ts --bundle
./run src/examples/zkapps/voting/run.ts --bundle
./run src/examples/simple_zkapp.ts --bundle
./run src/examples/zkapps/reducer/reducer_composite.ts --bundle
./run src/examples/zkapps/composability.ts --bundle
./run src/examples/zkapps/dex/run.ts --bundle
./run src/examples/zkapps/dex/happy-path-with-actions.ts --bundle
./run src/examples/zkapps/dex/upgradability.ts --bundle
```

</file>

<file>

## path: /run-jest-tests.sh

url: https://github.com/o1-labs/o1js/blob/main/run-jest-tests.sh

```
#!/bin/bash
set -e
shopt -s globstar # to expand '**' into nested directories

for f in ./src/**/*.test.ts; do
  NODE_OPTIONS=--experimental-vm-modules npx jest $f;
done
```

</file>

<file>

## path: /run-mina-integration-tests.sh

url: https://github.com/o1-labs/o1js/blob/main/run-mina-integration-tests.sh

```
#!/bin/bash
set -e

node tests/integration/simple-zkapp-mock-apply.js
node tests/integration/inductive-proofs.js
```

</file>

<file>

## path: /run-minimal-mina-tests.sh

url: https://github.com/o1-labs/o1js/blob/main/run-minimal-mina-tests.sh

```
#!/bin/bash
set -e

./run src/tests/inductive-proofs-small.ts --bundle
```

</file>

<file>

## path: /run-unit-tests.sh

url: https://github.com/o1-labs/o1js/blob/main/run-unit-tests.sh

```
#!/bin/bash
set -e
shopt -s globstar # to expand '**' into nested directories./

# run the build:test
npm run build:test

# find all unit tests in dist/node and run them
# TODO it would be nice to make this work on Mac
# here is a first attempt which has the problem of not failing if one of the tests fails
# find ./dist/node -name "*.unit-test.js" -print -exec node {} \;
for f in ./dist/node/**/*.unit-test.js; do
  echo "Running $f"
  node --enable-source-maps --stack-trace-limit=1000 $f;
done
```

</file>

<file>

## path: /src/build/buildExample.js

url: https://github.com/o1-labs/o1js/blob/main/src/build/buildExample.js

```
import fs from 'fs/promises';
import path from 'path';
import ts from 'typescript';
import esbuild from 'esbuild';

export { buildAndImport, build, buildOne };

async function buildAndImport(srcPath, { keepFile = false }) {
  let absPath = await build(srcPath);
  let importedModule;
  try {
    importedModule = await import(absPath);
  } finally {
    if (!keepFile) await fs.unlink(absPath);
  }
  return importedModule;
}

async function build(srcPath, isWeb = false) {
  let tsConfig = findTsConfig() ?? defaultTsConfig;
  // TODO hack because ts.transpileModule doesn't treat module = 'nodenext' correctly
  // but &nbsp;tsc&nbsp; demands it to be &nbsp;nodenext&nbsp;
  tsConfig.compilerOptions.module = 'esnext';

  let outfile = srcPath.replace('.ts', '.tmp.js');

  await esbuild.build({
    entryPoints: [srcPath],
    bundle: true,
    format: 'esm',
    platform: isWeb ? 'node' : 'browser',
    outfile,
    target: 'esnext',
    resolveExtensions: ['.node.js', '.ts', '.js'],
    logLevel: 'error',
    plugins: isWeb
      ? [typescriptPlugin(tsConfig)]
      : [
          typescriptPlugin(tsConfig),
          makeNodeModulesExternal(),
          makeJsooExternal(),
        ],
    dropLabels: ['CJS'],
  });
  let absPath = path.resolve('.', outfile);
  return absPath;
}

async function buildOne(srcPath) {
  let tsConfig = findTsConfig() ?? defaultTsConfig;
  // TODO hack because ts.transpileModule doesn't treat module = 'nodenext' correctly
  // but &nbsp;tsc&nbsp; demands it to be &nbsp;nodenext&nbsp;
  tsConfig.compilerOptions.module = 'esnext';

  let outfile = path.resolve(
    './dist/node',
    srcPath.replace('.ts', '.js').replace('src', '.')
  );

  await esbuild.build({
    entryPoints: [srcPath],
    format: 'esm',
    platform: 'node',
    outfile,
    target: 'esnext',
    resolveExtensions: ['.node.js', '.ts', '.js'],
    logLevel: 'error',
    plugins: [typescriptPlugin(tsConfig)],
  });

  let absPath = path.resolve('.', outfile);
  return absPath;
}

const defaultTsConfig = {
  compilerOptions: {
    module: 'esnext',
    lib: ['dom', 'esnext'],
    target: 'esnext',
    importHelpers: true,
    strict: true,
    moduleResolution: 'nodenext',
    esModuleInterop: true,
    skipLibCheck: true,
    forceConsistentCasingInFileNames: true,
    experimentalDecorators: true,
    emitDecoratorMetadata: true,
    allowSyntheticDefaultImports: true,
  },
};

function typescriptPlugin(tsConfig) {
  return {
    name: 'plugin-typescript',
    setup(build) {
      build.onLoad({ filter: /\.tsx?$/ }, async (args) => {
        let src = await fs.readFile(args.path, { encoding: 'utf8' });
        let { outputText: contents } = ts.transpileModule(src, tsConfig);
        return { contents };
      });
    },
  };
}

function makeNodeModulesExternal() {
  let isNodeModule = /^[^./]|^\.[^./]|^\.\.[^/]/;
  return {
    name: 'plugin-external',
    setup(build) {
      build.onResolve({ filter: isNodeModule }, ({ path }) => ({
        path,
        external: true,
      }));
    },
  };
}

function makeJsooExternal() {
  let isJsoo = /(bc.cjs|plonk_wasm.cjs)$/;
  return {
    name: 'plugin-external',
    setup(build) {
      build.onResolve({ filter: isJsoo }, ({ path: filePath, resolveDir }) => {
        return {
          path: path.resolve(resolveDir, filePath),
          external: true,
        };
      });
    },
  };
}

function findTsConfig() {
  let tsConfigPath = ts.findConfigFile(process.cwd(), ts.sys.fileExists);
  if (tsConfigPath === undefined) return;
  let text = ts.sys.readFile(tsConfigPath);
  if (text === undefined) throw new Error(&nbsp;failed to read '${tsConfigPath}'&nbsp;);
  return ts.parseConfigFileTextToJson(tsConfigPath, text).config;
}
```

</file>

<file>

## path: /src/build/buildNode.js

url: https://github.com/o1-labs/o1js/blob/main/src/build/buildNode.js

```
import path from 'node:path';
import { fileURLToPath } from 'node:url';
import esbuild from 'esbuild';
import minimist from 'minimist';

let { bindings = './src/bindings/compiled/node_bindings/' } = minimist(
  process.argv.slice(2)
);

export { buildNode };

const entry = './src/index.ts';
const target = 'es2021';

let nodePath = path.resolve(process.argv[1]);
let modulePath = path.resolve(fileURLToPath(import.meta.url));
let isMain = nodePath === modulePath;

if (isMain) {
  console.log('building cjs version of', entry);
  console.log('using bindings from', bindings);
  await buildNode({ production: process.env.NODE_ENV === 'production' });
  console.log('finished build');
}

async function buildNode({ production }) {
  // bundle the index.js file with esbuild and create a new index.cjs file which conforms to CJS
  let jsEntry = path.resolve(
    'dist/node',
    path.basename(entry).replace('.ts', '.js')
  );
  let outfile = jsEntry.replace('.js', '.cjs');
  await esbuild.build({
    entryPoints: [jsEntry],
    bundle: true,
    format: 'cjs',
    platform: 'node',
    outfile,
    target,
    resolveExtensions: ['.node.js', '.ts', '.js'],
    allowOverwrite: true,
    plugins: [makeNodeModulesExternal(), makeJsooExternal()],
    dropLabels: ['ESM'],
    minify: false,
  });
}

function makeNodeModulesExternal() {
  let isNodeModule = /^[^./]|^\.[^./]|^\.\.[^/]/;
  return {
    name: 'plugin-external',
    setup(build) {
      build.onResolve({ filter: isNodeModule }, ({ path }) => ({
        path,
        external: true,
      }));
    },
  };
}

function makeJsooExternal() {
  let isJsoo = /(bc.cjs|plonk_wasm.cjs)$/;
  return {
    name: 'plugin-external',
    setup(build) {
      build.onResolve({ filter: isJsoo }, ({ path: filePath, resolveDir }) => ({
        path:
          './' +
          path.relative(
            path.resolve('.', 'dist/node'),
            path.resolve(resolveDir, filePath)
          ),
        external: true,
      }));
    },
  };
}
```

</file>

<file>

## path: /src/build/buildWeb.js

url: https://github.com/o1-labs/o1js/blob/main/src/build/buildWeb.js

```
import esbuild from 'esbuild';
import fse, { move } from 'fs-extra';
import { readFile, writeFile, unlink } from 'node:fs/promises';
import path from 'node:path';
import { fileURLToPath } from 'node:url';
import { exec } from 'node:child_process';
import glob from 'glob';

export { buildWeb };

const entry = './src/index.ts';
const target = 'es2022';

let nodePath = path.resolve(process.argv[1]);
let modulePath = path.resolve(fileURLToPath(import.meta.url));
let isMain = nodePath === modulePath;
```

```
if (isMain) {
  console.log('building', entry);
  await buildWeb({ production: process.env.NODE_ENV === 'production' });
  console.log('finished build');
}

async function buildWeb({ production }) {
  let minify = !!production;

  // prepare plonk_wasm.js with bundled wasm in function-wrapped form
  let bindings = await readFile(
    './src/bindings/compiled/web_bindings/plonk_wasm.js',
    'utf8'
  );
  bindings = rewriteWasmBindings(bindings);
  let tmpBindingsPath = 'src/bindings/compiled/web_bindings/plonk_wasm.tmp.js';
  await writeFile(tmpBindingsPath, bindings);
  await esbuild.build({
    entryPoints: [tmpBindingsPath],
    bundle: true,
    format: 'esm',
    outfile: tmpBindingsPath,
    target: 'esnext',
    plugins: [wasmPlugin()],
    allowOverwrite: true,
  });
  bindings = await readFile(tmpBindingsPath, 'utf8');
  bindings = rewriteBundledWasmBindings(bindings);
  await writeFile(tmpBindingsPath, bindings);

  // run typescript
  let tscPromise = execPromise('npx tsc -p tsconfig.web.json');

  // copy over pure js files
  let copyPromise = copy({
    './src/bindings/compiled/web_bindings/': './dist/web/web_bindings/',
    './src/snarky.d.ts': './dist/web/snarky.d.ts',
    './src/bindings/js/wrapper.web.js': './dist/web/bindings/js/wrapper.js',
    './src/bindings/js/web/': './dist/web/bindings/js/web/',
  });

  await Promise.all([tscPromise, copyPromise]);

  if (minify) {
    let o1jsWebPath = './dist/web/web_bindings/snarky_js_web.bc.js';
    let o1jsWeb = await readFile(o1jsWebPath, 'utf8');
    let { code } = await esbuild.transform(o1jsWeb, {
      target,
      logLevel: 'error',
      minify,
    });
    await writeFile(o1jsWebPath, code);
  }

  // overwrite plonk_wasm with bundled version
  await copy({ [tmpBindingsPath]: './dist/web/web_bindings/plonk_wasm.js' });
  await unlink(tmpBindingsPath);

  // move all .web.js files to their .js counterparts
  let webFiles = glob.sync('./dist/web/**/*.web.js');
  await Promise.all(
    webFiles.map((file) =>
      move(file, file.replace('.web.js', '.js'), { overwrite: true })
    )
  );

  // run esbuild on the js entrypoint
  let jsEntry = path.basename(entry).replace('.ts', '.js');
  await esbuild.build({
    entryPoints: [&nbsp;./dist/web/${jsEntry}&nbsp;],
    bundle: true,
    format: 'esm',
    outfile: 'dist/web/index.js',
    resolveExtensions: ['.js', '.ts'],
    plugins: [wasmPlugin(), srcStringPlugin()],
    dropLabels: ['CJS'],
    external: ['*.bc.js'],
    target,
    allowOverwrite: true,
    logLevel: 'error',
    minify,
  });
}

async function copy(copyMap) {
  let promises = [];
  for (let [source, target] of Object.entries(copyMap)) {
    promises.push(
      fse.copy(source, target, {
       recursive: true,
       overwrite: true,
       dereference: true,
      })
    );
  }
  await Promise.all(promises);
}

function execPromise(cmd) {
  return new Promise((res, rej) =>
    exec(cmd, (err, stdout) => {
      if (err) {
       console.log(stdout);
```

```
      return rej(err);
    }
    res(stdout);
  })
 );
}

function rewriteWasmBindings(src) {
 src = src
  .replace("new URL('plonk_wasm_bg.wasm', import.meta.url)", 'wasmCode')
  .replace('import.meta.url', '"/"');
 return &nbsp;import wasmCode from './plonk_wasm_bg.wasm';
 let startWorkers, terminateWorkers;
${src}&nbsp;;
}
function rewriteBundledWasmBindings(src) {
 let i = src.indexOf('export {');
 let exportSlice = src.slice(i);
 let defaultExport = exportSlice.match(/\w* as default/)[0];
 exportSlice = exportSlice
  .replace(defaultExport, &nbsp;default: __wbg_init&nbsp;)
  .replace('export', 'return');
 src = src.slice(0, i) + exportSlice;

 src = src.replace('var startWorkers;\n', '');
 src = src.replace('var terminateWorkers;\n', '');
 return &nbsp;import { startWorkers, terminateWorkers } from '../bindings/js/web/worker-helpers.js'
export {plonkWasm as default};
function plonkWasm() {
 ${src}
}
plonkWasm.deps = [startWorkers, terminateWorkers]&nbsp;;
}

function wasmPlugin() {
 return {
  name: 'wasm-plugin',
  setup(build) {
   build.onLoad({ filter: /\.wasm$/ }, async ({ path }) => {
    return {
     contents: await readFile(path),
     loader: 'binary',
    };
   });
  },
 };
}

function srcStringPlugin() {
 return {
  name: 'src-string-plugin',
  setup(build) {
    build.onResolve(
      { filter: /^string:/ },
      async ({ path: importPath, resolveDir }) => {
        let absPath = path.resolve(
          resolveDir,
          importPath.replace('string:', '')
        );
        return {
          path: absPath,
          namespace: 'src-string',
        };
      }
    );

    build.onLoad(
      { filter: /.*/, namespace: 'src-string' },
      async ({ path }) => {
        return {
          contents: await readFile(path, 'utf8'),
          loader: 'text',
        };
      }
    );
  },
 };
}

function deferExecutionPlugin() {
 return {
  name: 'defer-execution-plugin',
  setup(build) {
    build.onResolve(
      { filter: /^defer:/ },
      async ({ path: importPath, resolveDir }) => {
        let absPath = path.resolve(
          resolveDir,
          importPath.replace('defer:', '')
        );
        return {
          path: absPath,
          namespace: 'defer-execution',
        };
      }
    );

    build.onLoad(
      { filter: /.*/, namespace: 'defer-execution' },
      async ({ path }) => {
        let code = await readFile(path, 'utf8');
        // replace direct eval, because esbuild refuses to bundle it
        // code = code.replace(/eval\(/g, '(0, eval)(');
        code = code.replace(
          /function\(\)\s*\{\s*return this\s*\}\(\)/g,
          'window'
        );
        code = code.replace(
          /function\(\)\s*\{\s*return this;\s*\}\(\)/g,
          'window'
        );
        let deferedCode = &nbsp;
        let require = () => {};
        export default () => {\n${code}\n};&nbsp;;
        return {
          contents: deferedCode,
          loader: 'js',
        };
      }
    );
  },
 };
}
```

</file>

<file>

## path: /src/build/copy-artifacts.js

url: https://github.com/o1-labs/o1js/blob/main/src/build/copy-artifacts.js

```
// copy compiled jsoo/wasm artifacts from a folder in the source tree to the folder where they are imported
from
// (these are not the same folders so that we don't automatically pollute the source tree when rebuilding
artifacts)
import { copyFromTo } from './utils.js';

await copyFromTo(
  ['src/bindings/compiled/node_bindings/'],
  'node_bindings',
  '_node_bindings'
);
```

</file>

<file>

## path: /src/build/copy-to-dist.js

url: https://github.com/o1-labs/o1js/blob/main/src/build/copy-to-dist.js

```
// copy some files from /src to /dist/node that tsc doesn't copy because we have .d.ts files for them
import { copyFromTo } from './utils.js';

await copyFromTo(
  ['src/snarky.d.ts', 'src/bindings/compiled/_node_bindings'],
  'src/',
  'dist/node/'
);
```

</file>

<file>

## path: /src/build/e2eTestsBuildHelper.js

url: https://github.com/o1-labs/o1js/blob/main/src/build/e2eTestsBuildHelper.js

```
import replace from 'replace-in-file';

const options = {
  files: './dist/web/examples/zkapps/**/*.js',
  from: /from 'o1js'/g,
  to: "from '../../../index.js'",
};

try {
  const results = await replace(options);
  console.log('Replacement results:', results);
} catch (error) {
  console.error('Error occurred:', error);
}
```

</file>

<file>

## path: /src/build/fix-wasm-bindings-node.js

url: https://github.com/o1-labs/o1js/blob/main/src/build/fix-wasm-bindings-node.js

```
import fs from 'node:fs/promises';

const file = process.argv[2];

let src = await fs.readFile(file, 'utf8');

src = src.replace(
  "imports['env'] = require('env');",
  &nbsp;
let { isMainThread, workerData } = require('worker_threads');

let env = {};
if (isMainThread) {
  env.memory = new WebAssembly.Memory({
    initial: 20,
    maximum: 65536,
    shared: true,
  });
} else {
  env.memory = workerData.memory;
}

imports['env'] = env;
&nbsp;
);

await fs.writeFile(file, src, 'utf8');
```

</file>

<file>

## path: /src/build/jsLayoutToTypes.mjs

url: https://github.com/o1-labs/o1js/blob/main/src/build/jsLayoutToTypes.mjs

```
import fs from 'node:fs/promises';
import path from 'node:path';
import { fileURLToPath } from 'node:url';
import prettier from 'prettier';
import prettierRc from '../../.prettierrc.cjs';

// let jsLayout = JSON.parse(process.argv[2]);
let selfPath = fileURLToPath(import.meta.url);
let jsonPath = path.resolve(selfPath, '../../bindings/ocaml/jsLayout.json');
let jsLayout = JSON.parse(await fs.readFile(jsonPath, 'utf8'));

console.log(&nbsp;jsLayoutToTypes.mjs: generating TS types from ${jsonPath}&nbsp;);

let builtinLeafTypes = new Set([
  'number',
  'string',
  'null',
  'undefined',
  'bigint',
]);
let indent = '';

function writeType(typeData, isJson, withTypeMap) {
  let converters = {};
  if (!isJson && typeData.checkedType) {
    let name = typeData.checkedTypeName;
    converters[name] = {
      typeName: name,
      type: writeType(typeData.checkedType, false, withTypeMap).output,
      jsonType: writeType(typeData, true, true).output,
    };
    typeData = typeData.checkedType;
  }
  let { type, inner, entries, keys, optionType, docEntries } = typeData;
  if (type === 'array') {
    let {
      output,
      dependencies,
      converters: j,
    } = writeType(inner, isJson, withTypeMap);
    mergeObject(converters, j);
    return {
      output: &nbsp;${output}[]&nbsp;,
      dependencies,
      converters,
    };
  }
  if (type === 'option') {
    let {
      output,
      dependencies,
      converters: j,
    } = writeType(inner, isJson, withTypeMap);
    if (optionType === 'flaggedOption' || optionType === 'closedInterval') {
      dependencies ??= new Set();
      dependencies.add('Bool');
    }
    mergeObject(converters, j);
    return {
      output: isJson
        ? &nbsp;(${output} | null)&nbsp;
        : optionType === 'implicit'
        ? output
        : optionType === 'flaggedOption' || optionType === 'closedInterval'
        ? &nbsp;{isSome: Bool, value: ${output}}&nbsp;
        : &nbsp;(${output} | undefined)&nbsp;,
      dependencies,
      converters,
    };
  }
  if (type === 'object') {
    let dependencies = new Set();
    let output = '{\n';
    indent += '  ';
    // TODO: make docs work and use them for doccomments
    for (let key of keys) {
      let value = entries[key];
      let questionMark = '';
      if (
        !isJson &&
        value.type === 'option' &&
        value.optionType === 'orUndefined'
      ) {
        value = value.inner;
        questionMark = '?';
      }
      let inner = writeType(value, isJson, withTypeMap);
      mergeSet(dependencies, inner.dependencies);
      mergeObject(converters, inner.converters);
      output += indent + &nbsp;${key}${questionMark}: ${inner.output};\n&nbsp;;
    }
    indent = indent.slice(2);
    output += indent + '}';
    return { output, dependencies, converters };
  }
  if (withTypeMap & !builtinLeafTypes.has(type)) {
    type = &nbsp;${isJson ? 'Json.' : ''}TypeMap["${type}"]&nbsp;;
  }
  // built in type
  if (builtinLeafTypes.has(type)) return { output: type, converters };
  // else: leaf type that has to be specified manually
  return {
    output: type,
    dependencies: builtinLeafTypes.has(type) ? new Set() : new Set([type]),
    converters,
  };
}

function writeTsContent(types, isJson, leavesRelPath) {
  let output = '';
  let dependencies = new Set();
  let converters = {};
  let exports = new Set(isJson ? [] : ['customTypes']);
  for (let [Type, value] of Object.entries(types)) {
    let inner = writeType(value, isJson);
    exports.add(Type);
    mergeSet(dependencies, inner.dependencies);
    mergeObject(converters, inner.converters);
    output += &nbsp;type ${Type} = ${inner.output};\n\n&nbsp;;
    if (!isJson) {
      output += &nbsp;let ${Type} = provableFromLayout<${Type}, Json.${Type}>(jsLayout.${Type} as
any);\n\n&nbsp;;
    }
  }

  let customTypes = Object.values(converters);
  let customTypeNames = Object.values(converters).map((c) => c.typeName);
  let imports = new Set();
  mergeSet(imports, dependencies);
  mergeSet(imports, new Set(customTypeNames));
  let typeMapKeys = diffSets(dependencies, new Set(customTypeNames));

  let importPath = leavesRelPath;
  return &nbsp;// @generated this file is auto-generated - don't edit it directly

import { ${[...imports].join(', ')} } from '${importPath}';
${
  !isJson
    ? "import { GenericProvableExtended } from '../../lib/generic.js';\n" +
      "import { ProvableFromLayout, GenericLayout } from '../../lib/from-layout.js';\n" +
      "import * as Json from './transaction-json.js';\n" +
      "import { jsLayout } from './js-layout.js';\n"
    : ''
}

export { ${[...exports].join(', ')} };
${
  !isJson
    ? 'export { Json };\n' +
      &nbsp;export * from '${leavesRelPath}';\n&nbsp; +
      'export { provableFromLayout, toJSONEssential, emptyValue, Layout, TypeMap };\n'
    : &nbsp;export * from '${leavesRelPath}';\n&nbsp; + 'export { TypeMap };\n'
}

type TypeMap = {
  ${[...typeMapKeys].map((type) => &nbsp;${type}: ${type};&nbsp;).join('\n')}
}
${
  (!isJson || '') &&
  &nbsp;
const TypeMap: {
  [K in keyof TypeMap]: ProvableExtended<TypeMap[K], Json.TypeMap[K]>;
} = {
  ${[...typeMapKeys].join(', ')}
}
&nbsp;
}

${
  (!isJson || '') &&
  &nbsp;
```

```
type ProvableExtended<T, TJson> = GenericProvableExtended<T, TJson, Field>;
type Layout = GenericLayout<TypeMap>;

type CustomTypes = { ${customTypes
    .map((c) => &nbsp;${c.typeName}: ProvableExtended<${c.type}, ${c.jsonType}>;&nbsp;)
    .join(' ')} }
let customTypes: CustomTypes = { ${customTypeNames.join(', ')} };
let { provableFromLayout, toJSONEssential, emptyValue } = ProvableFromLayout<
  TypeMap,
  Json.TypeMap
>(TypeMap, customTypes);
&nbsp;
}

${output}&nbsp;;
}

async function writeTsFile(content, relPath) {
  let absPath = path.resolve(selfPath, relPath);
  content = prettier.format(content, {
    filepath: absPath,
    ...prettierRc,
  });
  await fs.writeFile(absPath, content);
}
let genPath = '../../bindings/mina-transaction/gen';
await ensureDir(genPath);

let jsonTypesContent = writeTsContent(
  jsLayout,
  true,
  '../transaction-leaves-json.js'
);
await writeTsFile(jsonTypesContent, &nbsp;${genPath}/transaction-json.ts&nbsp;);

let jsTypesContent = writeTsContent(
  jsLayout,
  false,
  '../transaction-leaves.js'
);
await writeTsFile(jsTypesContent, &nbsp;${genPath}/transaction.ts&nbsp;);

jsTypesContent = writeTsContent(
  jsLayout,
  false,
  '../transaction-leaves-bigint.js'
);
await writeTsFile(jsTypesContent, &nbsp;${genPath}/transaction-bigint.ts&nbsp;);

await writeTsFile(
  &nbsp;// @generated this file is auto-generated - don't edit it directly
export { jsLayout };
```

```
let jsLayout = ${JSON.stringify(jsLayout)};
&nbsp;;,
  &nbsp;${genPath}/js-layout.ts&nbsp;
);

function mergeSet(target, source) {
  if (source === undefined) return;
  for (let x of source) {
    target.add(x);
  }
}
function diffSets(s0, s1) {
  let s = new Set(s0);
  for (let x of s1) {
    s.delete(x);
  }
  return s;
}

function mergeObject(target, source) {
  if (source === undefined) return;
  for (let key in source) {
    target[key] = source[key];
  }
}

async function ensureDir(relPath) {
  let absPath = path.resolve(selfPath, relPath);
  let exists = false;
  try {
    await fs.stat(absPath);
    exists = true;
  } catch {}
  if (!exists) {
    await fs.mkdir(absPath, { recursive: true });
  }
}
```

</file>

<file>

## path: /src/build/run.js

url: https://github.com/o1-labs/o1js/blob/main/src/build/run.js

```
#!/usr/bin/env node
import minimist from 'minimist';
import { buildAndImport, buildOne } from './buildExample.js';

let {
  _: [filePath],
  main,
  default: runDefault,
  keep,
  bundle,
} = minimist(process.argv.slice(2));

if (!filePath) {
  console.log(&nbsp;Usage:
npx snarky-run [file]&nbsp;);
  process.exit(0);
}
if (!bundle) {
  let absPath = await buildOne(filePath);
  console.log(&nbsp;running ${absPath}&nbsp;);
  let module = await import(absPath);
  if (main) await module.main();
  if (runDefault) await module.default();
} else {
  let module = await buildAndImport(filePath, { keepFile: !!keep });
  if (main) await module.main();
  if (runDefault) await module.default();
}
```

</file>

<file>

## path: /src/build/utils.js

url: https://github.com/o1-labs/o1js/blob/main/src/build/utils.js

```
import fse from 'fs-extra';

export { copyFromTo };

function copyFromTo(files, srcDir = undefined, targetDir = undefined) {
  return Promise.all(
    files.map((source) => {
      let target = source.replace(srcDir, targetDir);
      return fse.copy(source, target, {
        recursive: true,
        overwrite: true,
        dereference: true,
      });
    })
  );
}
```

</file>

<file>

## path: /src/examples/README.md

url: https://github.com/o1-labs/o1js/blob/main/src/examples/README.md

# o1js Examples

This folder contains many examples for using o1js. Take a look around!

## Running examples

You can run most examples using Node.js from the root directory, using the ./run script:

```
./run src/examples/some-example.ts
```

Some examples depend on other files in addition to "o1js". For those examples, you need to add the --bundle option to bundle them before running:

```
./run src/examples/multi-file-example.ts --bundle
```

Most of the examples do not depend on Node.js specific APIs, and can also be run in a browser. To do so, use:

```
./run-in-browser.js src/examples/web-compatible-example.ts
```

After running the above, navigate to http://localhost:8000 and open your browser's developer console to see the example executing.
</file>

<file>

## path: /src/examples/api_exploration.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/api_exploration.ts

```
import {
  Field,
  Bool,
  Group,
  Scalar,
  PrivateKey,
  PublicKey,
  Signature,
  Int64,
  Provable,
  Struct,
} from 'o1js';

/* This file demonstrates the classes and functions available in o1js */

/* # Field */

/* The most basic type is Field, which is an element of a prime order field.
   The field is the [Pasta Fp field](https://electriccoin.co/blog/the-pasta-curves-for-halo-2-and-beyond/)
   of order 28948022309329048855892746252171976963363056481941560715954676764349967630337
*/

// You can initialize literal field elements with numbers, booleans, or decimal strings
const x0: Field = new Field('37');
// Typescript has type inference, so type annotations are usually optional.
let x1 = new Field(37);
console.assert(x0.equals(x1).toBoolean());

// The &nbsp;new&nbsp; keyword is optional as well:
x1 = Field(37);
console.assert(x0.equals(x1).toBoolean());

/* You can perform arithmetic operations on field elements.
   The arithmetic methods can take any "fieldy" values as inputs:
   Field, number, string, or boolean
*/
const b = Field(1);
const z = x0.mul(x1).add(b).div(234).square().neg().sub('67').add(0);

/* Field elements can be converted to their full, little endian binary representation. */
let bits: Bool[] = z.toBits();

/* If you know (or want to assert) that a field element fits in fewer bits, you can
   also unpack to a sequence of bits of a specified length. This is useful for doing
   range proofs for example. */
let smallFieldElt = new Field(23849);
let smallBits: Bool[] = smallFieldElt.toBits(32);
```

```
console.assert(smallBits.length === 32);

/* There are lots of other useful method on field elements, like comparison methods.
   Try right-clicking on the Field type, or and peeking the definition to see what they are.

   Or, you can look at the autocomplete list on a field element's methods. You can try typing

   z.

   to see the methods on &nbsp;z : Field&nbsp; for example.
*/

/* # Bool */

/* Another important type is Bool. The Bool type is the in-SNARK representation of booleans.
   They are different from normal booleans in that you cannot write an if-statement whose
   condition has type Bool. You need to use the Provable.if function, which is like a value-level
   if (or something like a ternary expression). We will see how that works in a little.
*/

/* Bool values can be initialized using booleans. */

const b0 = Bool(false);
const b1 = Bool(true);

/* There are a number of methods available on Bool, like &nbsp;and&nbsp;, &nbsp;or&nbsp;, and
&nbsp;not&nbsp;. */
const b3: Bool = b0.and(b1.not()).or(b1);

/* The most important thing you can do with a Bool is use the &nbsp;Provable.if&nbsp; function
   to conditionally select a value.

   &nbsp;Provable.if&nbsp; has the type

   &nbsp;&nbsp;&nbsp;
   if<T>(
     b: Bool | boolean,
     x: T,
     y: T
   ): T
   &nbsp;&nbsp;&nbsp;

   &nbsp;Provable.if(b, x, y)&nbsp; evaluates to &nbsp;x&nbsp; if &nbsp;b&nbsp; is true, and evalutes to
&nbsp;y&nbsp; if &nbsp;b&nbsp; is false,
   so it works like a ternary if expression &nbsp;b ? x : y&nbsp;.

   The generic type T can be instantiated to primitive types like Bool, Field, or Group, or
   compound types like arrays (as long as the lengths are equal) or objects (as long as the keys
   match).
*/

const v: Field = Provable.if(b0, x0, z);
```

```
/* b0 is false, so we expect v to be equal to z. */
console.assert(v.equals(z).toBoolean());

/* As mentioned, we can also use &nbsp;Provable.if&nbsp; with compound types. */
let CompoundType = Struct({
  foo: [Field, Field],
  bar: { x: Field, b: Bool },
});

const c = Provable.if(
  b1,
  CompoundType,
  { foo: [x0, z], bar: { x: x1, b: b1 } },
  { foo: [z, x0], bar: { x: z, b: b0 } }
);

console.assert(c.bar.x.equals(x1).toBoolean());

// Provable.switch is a generalization of Provable.if, for when you need to distinguish between multiple
cases.
let x = Provable.switch([Bool(false), Bool(true), Bool(false)], Int64, [
  Int64.from(1),
  Int64.from(2),
  Int64.from(3),
]);
x.assertEquals(Int64.from(2));

/* # Signature
 */

/* The standard library of o1js comes with a Signature scheme.
   The message to be signed is an array of field elements, so any application level
   message data needs to be encoded as an array of field elements before being signed.
*/

let privKey: PrivateKey = PrivateKey.random();
let pubKey: PublicKey = PublicKey.fromPrivateKey(privKey);

let msg0: Field[] = [0xba5eba11, 0x15, 0xbad].map((x) => new Field(x));
let msg1: Field[] = [0xfa1afe1, 0xc0ffee].map((x) => new Field(x));
let signature = Signature.create(privKey, msg0);

console.assert(signature.verify(pubKey, msg0).toBoolean());
console.assert(!signature.verify(pubKey, msg1).toBoolean());

/* # Group

  This type represents points on the [Pallas elliptic curve](https://electriccoin.co/blog/the-pasta-curves-for-
halo-2-and-beyond/).

  It is a prime-order curve defined by the equation
  y^2 = x^3 + 5
*/

/* You can initialize elements as literals as follows: */
let g0 = new Group(-1, 2);
let g1 = new Group({ x: -2, y: 2 });

/* There is also a predefined generator. */
let g2 = Group.generator;

/* Points can be added, subtracted, and negated */
let g3 = g0.add(g1).neg().sub(g2);

/* Points can also be scaled by scalar field elements. Note that Field and Scalar
   are distinct and represent elements of distinct fields. */
let s0: Scalar = Scalar.random();
let g4: Group = g3.scale(s0);
```

</file>

<file>

## path: /src/examples/benchmarks/hash-witness.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/benchmarks/hash-witness.ts

```
/**
 * benchmark witness generation for an all-mul circuit
 */
import { Field, Provable, Poseidon } from 'o1js';
import { tic, toc } from '../utils/tic-toc.js';

// parameters
let nPermutations = 1 << 12; // 2^12 x 11 rows < 2^16 rows, should just fit in a circuit

// the circuit: hash a number n times
let xConst = Field.random();

function main(nMuls: number) {
  let x = Provable.witness(Field, () => xConst);
  let z = x;
  for (let i = 0; i < nMuls; i++) {
    z = Poseidon.hash([z, x]);
  }
}

tic('run and check');
Provable.runAndCheck(() => main(nPermutations));
toc();
```

</file>

<file>

## path: /src/examples/benchmarks/import.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/benchmarks/import.ts

```
let start = performance.now();
await import('../../snarky.js');
let time = performance.now() - start;

console.log(&nbsp;import jsoo: ${time.toFixed(0)}ms&nbsp;);
```

</file>

<file>

## path: /src/examples/benchmarks/import.web.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/benchmarks/import.web.ts

```
let start = performance.now();
await import('o1js');
let time = performance.now() - start;

console.log(&nbsp;import jsoo: ${time.toFixed(0)}ms&nbsp;);
```

</file>

<file>

## path: /src/examples/benchmarks/mul-web.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/benchmarks/mul-web.ts

```
/**
 * benchmark a circuit filled with generic gates
 */
import { Circuit, Field, Provable, circuitMain, ZkProgram } from 'o1js';
import { tic, toc } from '../utils/tic-toc.js';

// parameters
let nMuls = (1 << 16) + (1 << 15); // not quite 2^17 generic gates = not quite 2^16 rows
// let nMuls = 1 << 5;
let withPickles = true;

// the circuit: multiply a number with itself n times
let xConst = Field.random();

function main(nMuls: number) {
  let x = Provable.witness(Field, () => xConst);
  let z = x;
  for (let i = 0; i < nMuls; i++) {
    z = z.mul(x);
  }
}

function getRows(nMuls: number) {
  let { rows } = Provable.constraintSystem(() => main(nMuls));
  return rows;
}

function simpleKimchiCircuit(nMuls: number) {
  class MulChain extends Circuit {
    @circuitMain
    static run() {
      main(nMuls);
    }
  }
  // circuitMain(MulChain, 'run');
  return MulChain;
}

function picklesCircuit(nMuls: number) {
  return ZkProgram({
    name: 'mul-chain',
    methods: {
      run: {
        privateInputs: [],
        method() {
          main(nMuls);
        },
      },
    },
  });
}

// the script

console.log('circuit size (without pickles overhead)', getRows(nMuls));

if (withPickles) {
  let circuit = picklesCircuit(nMuls);
  tic('compile 1 (includes srs creation)');
  await circuit.compile();
  toc();

  tic('compile 2');
  await circuit.compile();
  toc();

  tic('prove');
  let p = await circuit.run();
  toc();

  tic('verify');
  let ok = await circuit.verify(p);
  toc();
  if (!ok) throw Error('invalid proof');
} else {
  let circuit = simpleKimchiCircuit(nMuls);

  tic('compile 1 (includes srs creation)');
  let kp = await circuit.generateKeypair();
  toc();

  tic('compile 2');
  kp = await circuit.generateKeypair();
  toc();

  tic('prove');
  let p = await circuit.prove([], [], kp);
  toc();

  tic('verify');
  let ok = await circuit.verify([], kp.verificationKey(), p);
  toc();
  if (!ok) throw Error('invalid proof');
}
```

</file>

<file>

## path: /src/examples/benchmarks/mul-witness.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/benchmarks/mul-witness.ts

```
/**
 * benchmark witness generation for an all-mul circuit
 */
import { Field, Provable } from 'o1js';
import { tic, toc } from '../utils/tic-toc.js';

// parameters
let nMuls = (1 << 16) + (1 << 15); // not quite 2^17 generic gates = not quite 2^16 rows

// the circuit: multiply a number with itself n times
let xConst = Field.random();

function main(nMuls: number) {
  let x = Provable.witness(Field, () => xConst);
  let z = x;
  for (let i = 0; i < nMuls; i++) {
    z = z.mul(x);
  }
}

tic('run and check');
Provable.runAndCheck(() => main(nMuls));
toc();
```

</file>

<file>

## path: /src/examples/benchmarks/mul.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/benchmarks/mul.ts

```
/**
 * benchmark a circuit filled with generic gates
 */
import { Circuit, Field, Provable, circuitMain, ZkProgram } from 'o1js';
import { tic, toc } from '../utils/tic-toc.node.js';

// parameters
let nMuls = (1 << 16) + (1 << 15); // not quite 2^17 generic gates = not quite 2^16 rows
let withPickles = true;

// the circuit: multiply a number with itself n times
let xConst = Field.random();

function main(nMuls: number) {
  let x = Provable.witness(Field, () => xConst);
  let z = x;
  for (let i = 0; i < nMuls; i++) {
    z = z.mul(x);
  }
```

```
}

function getRows(nMuls: number) {
  let { rows } = Provable.constraintSystem(() => main(nMuls));
  return rows;
}

function simpleKimchiCircuit(nMuls: number) {
  class MulChain extends Circuit {
    @circuitMain
    static run() {
      main(nMuls);
    }
  }
  return MulChain;
}

function picklesCircuit(nMuls: number) {
  return ZkProgram({
    name: 'mul-chain',
    methods: {
      run: {
        privateInputs: [],
        method() {
          main(nMuls);
        },
      },
    },
  });
}

console.log('circuit size (without pickles overhead)', getRows(nMuls));

if (withPickles) {
  let circuit = picklesCircuit(nMuls);
  tic('compile 1 (includes srs creation)');
  await circuit.compile();
  toc();

  tic('compile 2');
  await circuit.compile();
  toc();

  tic('prove');
  let p = await circuit.run();
  toc();

  tic('verify');
  let ok = await circuit.verify(p);
  toc();
  if (!ok) throw Error('invalid proof');
} else {
  let circuit = simpleKimchiCircuit(nMuls);

  tic('compile 1 (includes srs creation)');
  let kp = await circuit.generateKeypair();
  toc();

  tic('compile 2');
  kp = await circuit.generateKeypair();
  toc();

  tic('prove');
  let p = await circuit.prove([], [], kp);
  toc();

  tic('verify');
  let ok = await circuit.verify([], kp.verificationKey(), p);
  toc();
  if (!ok) throw Error('invalid proof');
}
```

</file>

<file>

## path: /src/examples/circuit/README.md

url: https://github.com/o1-labs/o1js/blob/main/src/examples/circuit/README.md

# Circuit examples

These examples show how to use Circuit, which is a simple API to write a single circuit and create proofs for it.

In contrast to ZkProgram, Circuit does not pass through Pickles, but creates a proof with Kimchi directly. Therefore, it does not support recursion, but is also much faster.

Note that Circuit proofs are not compatible with Mina zkApps.
</file>

<file>

## path: /src/examples/circuit/preimage.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/circuit/preimage.ts

```
import { Poseidon, Field, Circuit, circuitMain, public_ } from 'o1js';

/**
 * Public input: a hash value h
 *
 * Prove:
 *   I know a value x such that hash(x) = h
 */
class Main extends Circuit {
  @circuitMain
  static main(preimage: Field, @public_ hash: Field) {
    Poseidon.hash([preimage]).assertEquals(hash);
  }
}

console.log('generating keypair...');
const kp = await Main.generateKeypair();

const preimage = Field(1);
const hash = Poseidon.hash([preimage]);

console.log('prove...');
const pi = await Main.prove([preimage], [hash], kp);

console.log('verify...');
let ok = await Main.verify([hash], kp.verificationKey(), pi);
console.log('ok?', ok);

if (!ok) throw Error('verification failed');
```

</file>

<file>

## path: /src/examples/circuit/root.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/circuit/root.ts

```
import { Field, Circuit, circuitMain, public_, Gadgets } from 'o1js';

/**
 * Public input: a field element x
 *
 * Prove:
 *   I know a value y < 2^64 that is a cube root of x.
 */
class Main extends Circuit {
  @circuitMain
  static main(@public_ x: Field, y: Field) {
    Gadgets.rangeCheck64(y);
    let y3 = y.square().mul(y);
    y3.assertEquals(x);
  }
}

console.log('generating keypair...');
console.time('generating keypair...');
const kp = await Main.generateKeypair();
console.timeEnd('generating keypair...');

console.log('prove...');
console.time('prove...');
const x = Field(8);
const y = Field(2);
const proof = await Main.prove([y], [x], kp);
console.timeEnd('prove...');

console.log('verify...');
console.time('verify...');
let vk = kp.verificationKey();
let ok = await Main.verify([x], vk, proof);
console.timeEnd('verify...');

console.log('ok?', ok);
```

</file>

<file>

## path: /src/examples/circuit_string.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/circuit_string.ts

```
import {
  isReady,
  CircuitString,
  SmartContract,
  method,
  Mina,
  PrivateKey,
} from 'o1js';
import * as assert from 'assert/strict';

// circuit which tests a couple of string features
class MyContract extends SmartContract {
  @method checkString(s: CircuitString) {
    let sWithExclamation = s.append(CircuitString.fromString('!'));
    sWithExclamation
      .equals(CircuitString.fromString('a string!'))
      .or(sWithExclamation.equals(CircuitString.fromString('some string!')))
      .assertTrue();
  }
}

await isReady;
let address = PrivateKey.random().toPublicKey();

console.log('compile...');
await MyContract.compile();
// should work
console.log('prove...');
let tx = await Mina.transaction(() => {
  new MyContract(address).checkString(CircuitString.fromString('a string'));
});
await tx.prove();
console.log('test 1 - ok');
// should work
tx = await Mina.transaction(() => {
  new MyContract(address).checkString(CircuitString.fromString('some string'));
});
await tx.prove();
console.log('test 2 - ok');
// should fail
let fails = await Mina.transaction(() => {
  new MyContract(address).checkString(CircuitString.fromString('different'));
})
  .then(() => false)
  .catch(() => true);
if (!fails) Error('proof was supposed to fail');
console.log('test 3 - ok');

const str = CircuitString.fromString('Your size');
const not_same_str = CircuitString.fromString('size');
assert.equal(str.equals(not_same_str).toBoolean(), false);

const equal1 = CircuitString.fromString('These strings are equivalent');
const equal2 = CircuitString.fromString('These strings are equivalent');

const circuitString = CircuitString.fromString(
  'This string completely encompasses this string'
);
```

```
const substring = CircuitString.fromString('this string');

if (!equal1.equals(equal2).toBoolean())
  throw Error('Strings are not equivalent 1');
console.log('Equivalent: "', equal1.toString(), '", "', equal2.toString(), '"');

if (!circuitString.substring(35, 46).equals(substring).toBoolean())
  throw Error('Strings are not equivalent 2');
console.log(
  'Equivalent: "',
  circuitString.substring(35, 46).toString(),
  '", "',
  substring.toString(),
  '"'
);

// if (!circuitString.contains(substring).toBoolean())
//   throw Error('String does not contain substring');

console.log(circuitString.append(substring).toString());

console.log('Everything looks good!');
```

</file>

<file>

## path: /src/examples/commonjs.cjs

url: https://github.com/o1-labs/o1js/blob/main/src/examples/commonjs.cjs

```
/**
 * Tests that o1js can be imported and used from commonJS files
 */
let {
  Field,
  State,
  PrivateKey,
  SmartContract,
  Mina,
  AccountUpdate,
  declareState,
  declareMethods,
} = require('o1js');

class SimpleZkapp extends SmartContract {
  constructor(address) {
    super(address);
    this.x = State();
  }
  events = { update: Field };

  init() {
    super.init();
    this.x.set(initialState);
  }

  update(y) {
    this.emitEvent('update', y);
    this.emitEvent('update', y);
    this.account.balance.assertEquals(this.account.balance.get());
    let x = this.x.get();
    this.x.assertEquals(x);
    this.x.set(x.add(y));
  }
}
declareState(SimpleZkapp, { x: Field });
declareMethods(SimpleZkapp, { update: [Field] });

let Local = Mina.LocalBlockchain();
Mina.setActiveInstance(Local);

let feePayerKey = Local.testAccounts[0].privateKey;
let feePayer = Local.testAccounts[0].publicKey;

let zkappKey = PrivateKey.random();
let zkappAddress = zkappKey.toPublicKey();

let initialState = Field(1);
let zkapp = new SimpleZkapp(zkappAddress);

main();

async function main() {
  console.log('compile');
  await SimpleZkapp.compile();

  console.log('deploy');
  let tx = await Mina.transaction(feePayer, () => {
    AccountUpdate.fundNewAccount(feePayer);
    zkapp.deploy();
  });
  await tx.sign([feePayerKey, zkappKey]).send();

  console.log('initial state: ' + zkapp.x.get());

  console.log('update');
  tx = await Mina.transaction(feePayer, () => zkapp.update(Field(3)));
  await tx.prove();
  await tx.sign([feePayerKey]).send();
  console.log('final state: ' + zkapp.x.get());
}
```

</file>

<file>

## path: /src/examples/constraint_system.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/constraint_system.ts

```
import { Field, Poseidon, Provable } from 'o1js';

let hash = Poseidon.hash([Field(1), Field(-1)]);

let { rows, digest, gates, publicInputSize } = Provable.constraintSystem(() => {
  let x = Provable.witness(Field, () => Field(1));
  let y = Provable.witness(Field, () => Field(-1));
  x.add(y).assertEquals(Field(0));
  let z = Poseidon.hash([x, y]);
  z.assertEquals(hash);
});

console.log(JSON.stringify(gates));
console.log({ rows, digest, publicInputSize });
```

</file>

<file>

## path: /src/examples/encoding-bijective.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/encoding-bijective.ts

```
import { Field, isReady, shutdown, Encoding } from 'o1js';

await isReady;
let n = 1000;

let { toBytes, fromBytes } = Encoding.Bijective.Fp;

// random fields
let fields = Array.from({ length: n }, () => Field.random());
let newFields = fromBytes(toBytes(fields));
let fieldsEqual = arrayEqual(fields, newFields, (f, g) =>
  f.equals(g).toBoolean()
);
if (!fieldsEqual) throw Error('roundtrip fields -> bytes -> fields failed');
else console.log('fields -> bytes -> fields: ok');

// random bytes
let bytes = (await import('node:crypto')).randomBytes(n * 32);
let newBytes = toBytes(fromBytes(bytes));
let bytesEqual = arrayEqual([...bytes], [...newBytes]);
if (!bytesEqual) throw Error('roundtrip bytes -> fields -> bytes failed');
else console.log('bytes -> fields -> bytes: ok');

shutdown();

function arrayEqual<T>(a: T[], b: T[], isEqual?: (a: T, b: T) => boolean) {
  if (isEqual === undefined) isEqual = (a, b) => a === b;
  if (a.length !== b.length) return false;
  for (let i = 0; i < a.length; i++) {
    if (!isEqual(a[i], b[i])) return false;
  }
  return true;
}
```

</file>

<file>

## path: /src/examples/encryption.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/encryption.ts

```
import {
  Encryption,
  Encoding,
  PrivateKey,
  isReady,
  Circuit,
  Provable,
} from 'o1js';
```

```
await isReady;

// generate keys
let privateKey = PrivateKey.random();
let publicKey = privateKey.toPublicKey();

// message
let message = 'This is a secret.';
let messageFields = Encoding.stringToFields(message);

// encrypt
let cipherText = Encryption.encrypt(messageFields, publicKey);

// decrypt
let decryptedFields = Encryption.decrypt(cipherText, privateKey);
let decryptedMessage = Encoding.stringFromFields(decryptedFields);

if (decryptedMessage !== message) throw Error('decryption failed');
console.log(&nbsp;Original message: "${message}"&nbsp;);
console.log(&nbsp;Recovered message: "${decryptedMessage}"&nbsp;);

// the same but in a checked computation

Provable.runAndCheck(() => {
  // encrypt
  let cipherText = Encryption.encrypt(messageFields, publicKey);

  // decrypt
  let decryptedFields = Encryption.decrypt(cipherText, privateKey);

  messageFields.forEach((m, i) => {
    m.assertEquals(decryptedFields[i]);
  });
});

// With a longer message
message = JSON.stringify({
  coinbase: {
    btc: 40000.0,
    eth: 3000.0,
    usdc: 1.0,
    ada: 1.02,
    avax: 70.43,
    mina: 2.13,
  },
  binance: {
    btc: 39999.0,
    eth: 3001.0,
    usdc: 1.01,
    ada: 0.99,
    avax: 70.21,
    mina: 2.07,
  },
});
messageFields = Encoding.stringToFields(message);

// encrypt
cipherText = Encryption.encrypt(messageFields, publicKey);

// decrypt
decryptedFields = Encryption.decrypt(cipherText, privateKey);
decryptedMessage = Encoding.stringFromFields(decryptedFields);

if (decryptedMessage !== message) throw Error('decryption failed');
console.log(&nbsp;Original message: "${message}"&nbsp;);
console.log(&nbsp;Recovered message: "${decryptedMessage}"&nbsp;);

// the same but in a checked computation

Provable.runAndCheck(() => {
  // encrypt
  let cipherText = Encryption.encrypt(messageFields, publicKey);

  // decrypt
  let decryptedFields = Encryption.decrypt(cipherText, privateKey);

  messageFields.forEach((m, i) => {
    m.assertEquals(decryptedFields[i]);
  });
});

console.log('everything works!');
```

</file>

<file>

## path: /src/examples/fetch.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/fetch.ts

```
import {
  fetchAccount,
  isReady,
  setGraphqlEndpoints,
  shutdown,
  fetchLastBlock,
  PublicKey,
  Types,
} from 'o1js';

await isReady;
setGraphqlEndpoints([
  'https://proxy.berkeley.minaexplorer.com/graphql',
  'https://berkeley.minascan.io/graphql',
]);

let zkappAddress = PublicKey.fromBase58(
  'B62qpRzFVjd56FiHnNfxokVbcHMQLT119My1FEdSq8ss7KomLiSZcan'
);
let { account, error } = await fetchAccount({
  publicKey: zkappAddress,
});
console.log('error', error);
console.log('account', Types.Account.toJSON(account!));

let block = await fetchLastBlock();
console.log('last block', JSON.stringify(block, null, 2));

await shutdown();
```

</file>

<file>

## path: /src/examples/internals/README.md

url: https://github.com/o1-labs/o1js/blob/main/src/examples/internals/README.md

# Examples: Internals

This folder contains examples which highlight inner workings and less-documented behaviours of o1js.

These examples might be useful for advanced users and contributors.
</file>

<file>

## path: /src/examples/internals/advanced-provable-types.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/internals/advanced-provable-types.ts

```
/**
 * This example explains some inner workings of provable types at the hand of a particularly
 * complex type: &nbsp;AccountUpdate&nbsp;.
 */
import assert from 'assert/strict';
import {
  AccountUpdate,
  PrivateKey,
  Provable,
  Empty,
  ProvableExtended,
} from 'o1js';
import { expect } from 'expect';

/**
 * Example of a complex provable type: &nbsp;AccountUpdate&nbsp;
 */
AccountUpdate satisfies Provable<AccountUpdate>;
console.log(&nbsp;an account update has ${AccountUpdate.sizeInFields()} fields&nbsp;);

let address = PrivateKey.random().toPublicKey();
let accountUpdate = AccountUpdate.defaultAccountUpdate(address);
accountUpdate.body.callDepth = 5;
accountUpdate.lazyAuthorization = {
  kind: 'lazy-signature',
  privateKey: PrivateKey.random(),
};

/**
 * Every provable type can be disassembled into its provable/in-circuit part (fields)
 * and a non-provable part (auxiliary).
 *
 * The parts can be assembled back together to create a new object which is deeply equal to the old one.
 */
let fields = AccountUpdate.toFields(accountUpdate);
let aux = AccountUpdate.toAuxiliary(accountUpdate);
let accountUpdateRecovered = AccountUpdate.fromFields(fields, aux);
expect(accountUpdateRecovered.body).toEqual(accountUpdate.body);
expect(accountUpdateRecovered.lazyAuthorization).toEqual(
  accountUpdate.lazyAuthorization
);

/**
 * Provable types which implement &nbsp;ProvableExtended&nbsp; can also be serialized to/from JSON.
 *
 * However, &nbsp;AccountUpdate&nbsp; specifically is a wrapper around an actual, core provable
extended type.
 * It has additional properties, like lazySignature, which are not part of the JSON representation
 * and therefore aren't recovered.
 */
AccountUpdate satisfies ProvableExtended<AccountUpdate>;
```

```
let json = AccountUpdate.toJSON(accountUpdate);
accountUpdateRecovered = AccountUpdate.fromJSON(json);
expect(accountUpdateRecovered.body).toEqual(accountUpdate.body);
expect(accountUpdateRecovered.lazyAuthorization).not.toEqual(
  accountUpdate.lazyAuthorization
);

/**
 * Provable.runAndCheck() can be used to run a circuit in "prover mode".
 * That means
 * -) witness() and asProver() blocks are excuted
 * -) constraints are checked; failing assertions throw an error
 */
Provable.runAndCheck(() => {
  /**
   * Provable.witness() is used to introduce all values to the circuit which are not hard-coded constants.
   *
   * Under the hood, it disassembles and reassembles the provable type with toFields(), toAuxiliary() and
fromFields().
   */
  let accountUpdateWitness = Provable.witness(
    AccountUpdate,
    () => accountUpdate
  );

  /**
   * The witness is "provably equal" to the original.
   * (this, under hood, calls assertEqual on all fields returned by .toFields()).
   */
  Provable.assertEqual(AccountUpdate, accountUpdateWitness, accountUpdate);

  /**
   * Auxiliary parts are also recovered in the witness.
   * Note, though, that this can't be enforced as part of a proof!
   */
  assert(
    accountUpdateWitness.body.callDepth === 5,
    'when witness block is executed, witness() recreates auxiliary parts of provable type'
  );
  Provable.assertEqual(
    PrivateKey,
    (accountUpdateWitness.lazyAuthorization as any).privateKey,
    (accountUpdate.lazyAuthorization as any).privateKey
  );
});

/**
 * Provable.constraintSystem() runs the circuit in "compile mode".
 * -) witness() and asProver() blocks are not executed
 * -) fields don't have actual values attached to them; they're purely abstract variables
 * -) constraints are not checked
 */
```

```
let result = Provable.constraintSystem(() => {
  /**
   * In compile mode, witness() returns
   * - abstract variables without values for fields
   * - dummy data for auxiliary
   */
  let accountUpdateWitness = Provable.witness(
    AccountUpdate,
    (): AccountUpdate => {
      throw 'not executed anyway';
    }
  );

  /**
   * Dummy data can take a different form depending on the provable type,
   * but in most cases it's "all-zeroes"
   */
  assert(
    accountUpdateWitness.body.callDepth === 0,
    'when witness block is not executed, witness() returns dummy data'
  );
  Provable.assertEqual(AccountUpdate, accountUpdateWitness, accountUpdate);
});

/**
 * Provable.constraintSystem() is a great way to investigate how many constraints operations take.
 *
 * Note that even just witnessing stuff takes constraints, for provable types which define a check() method.
 * Bools are proved to be 0 or 1, UInt64 is proved to be within [0, 2^64), etc.
 */
console.log(
  &nbsp;witnessing an account update and comparing it to another one creates ${result.rows} rows&nbsp;
);

/**
 * For account updates specifically, we typically don't want all the subfield checks. That's because
 * account updates are usually tied the _public input_. The public input is checked on the verifier side
 * already, including the well-formedness of its parts, so there's no need to include that in the proof.
 *
 * This is why we have this custom way of witnessing account updates, with the &nbsp;skipCheck&nbsp;
option.
 */
result = Provable.constraintSystem(() => {
  let { accountUpdate: accountUpdateWitness } = AccountUpdate.witness(
    Empty,
    () => ({ accountUpdate, result: undefined }),
    { skipCheck: true }
  );
  Provable.assertEqual(AccountUpdate, accountUpdateWitness, accountUpdate);
});
console.log(
  &nbsp;without all the checks on subfields, witnessing and comparing only creates ${result.rows}
rows&nbsp;
);

/**
 * To relate an account update to the hash which is the public input, we need to perform the hash in-circuit.
 * This is takes several 100 constraints, and is basically the minimal size of a zkApp method.
 */
result = Provable.constraintSystem(() => {
  let { accountUpdate: accountUpdateWitness } = AccountUpdate.witness(
    Empty,
    () => ({ accountUpdate, result: undefined }),
    { skipCheck: true }
  );
  accountUpdateWitness.hash();
});
console.log(&nbsp;hashing a witnessed account update creates ${result.rows} rows&nbsp;);
```

</file>

<file>

## path: /src/examples/matrix_mul.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/matrix_mul.ts

```
import { Field, provable, Provable } from 'o1js';

// there are two ways of specifying an n*m matrix

// provable
let Matrix3x3 = provable([
  [Field, Field, Field],
  [Field, Field, Field],
  [Field, Field, Field],
]);
// Provable.Array -- types somewhat more loosely but can be easier to write
let Matrix3x4 = Provable.Array(Provable.Array(Field, 4), 3);
let Matrix4x3 = Provable.Array(Provable.Array(Field, 3), 4);

/* @param x an n*m matrix, encoded as x[i][k] for row i column k.
 * @param y an m*o matrix, both encoded as y[k][j] for row j column j.
 * Returns an n*o matrix.
 */
function matrixMul(x: Field[][], y: Field[][]): Field[][] {
  let n = x.length;
  let m = y.length; // has to be === x[0].length
  let o = y[0].length;

  let result: Field[][] = [];

  // Compute the output matrix.
  for (let i = 0; i < n; i++) {
    result[i] = [];
    for (let j = 0; j < o; j++) {
      result[i][j] = Field(0);
      for (let k = 0; k < m; k++) {
        result[i][j] = result[i][j].add(x[i][k].mul(y[k][j]));
      }
    }
  }
  return result;
}

function circuit(): Field[][] {
  let x = Provable.witness(Matrix3x4, () => {
    return [
      [Field.random(), Field.random(), Field.random(), Field.random()],
      [Field.random(), Field.random(), Field.random(), Field.random()],
      [Field.random(), Field.random(), Field.random(), Field.random()],
    ];
  });
  let y = Provable.witness(Matrix4x3, () => {
    return [
      [Field.random(), Field.random(), Field.random()],
      [Field.random(), Field.random(), Field.random()],
      [Field.random(), Field.random(), Field.random()],
      [Field.random(), Field.random(), Field.random()],
    ];
  });
  return matrixMul(x, y);
}

let { rows } = Provable.constraintSystem(circuit);
let result: Field[][];
Provable.runAndCheck(() => {
  let result_ = circuit();
  Provable.asProver(() => {
    result = result_.map((x) => x.map((y) => y.toConstant()));
  });
});
console.log({ rows, result: Matrix3x3.toJSON(result!) });
```

</file>

<file>

## path: /src/examples/nullifier.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/nullifier.ts

```
import {
  PrivateKey,
  Nullifier,
  Field,
  SmartContract,
  state,
  State,
  method,
  MerkleMap,
  Circuit,
  MerkleMapWitness,
  Mina,
  AccountUpdate,
} from 'o1js';

class PayoutOnlyOnce extends SmartContract {
  @state(Field) nullifierRoot = State<Field>();
  @state(Field) nullifierMessage = State<Field>();

  @method payout(nullifier: Nullifier) {
    let nullifierRoot = this.nullifierRoot.getAndAssertEquals();
    let nullifierMessage = this.nullifierMessage.getAndAssertEquals();

    // verify the nullifier
    nullifier.verify([nullifierMessage]);

    let nullifierWitness = Circuit.witness(MerkleMapWitness, () =>
      NullifierTree.getWitness(nullifier.key())
    );

    // we compute the current root and make sure the entry is set to 0 (= unused)
    nullifier.assertUnused(nullifierWitness, nullifierRoot);

    // we set the nullifier to 1 (= used) and calculate the new root
    let newRoot = nullifier.setUsed(nullifierWitness);

    // we update the on-chain root
    this.nullifierRoot.set(newRoot);

    // we pay out a reward
    let balance = this.account.balance.getAndAssertEquals();

    let halfBalance = balance.div(2);
    // finally, we send the payout to the public key associated with the nullifier
    this.send({ to: nullifier.getPublicKey(), amount: halfBalance });
  }
}

const NullifierTree = new MerkleMap();

let Local = Mina.LocalBlockchain({ proofsEnabled: true });
Mina.setActiveInstance(Local);

// a test account that pays all the fees, and puts additional funds into the zkapp
```

```
let { privateKey: senderKey, publicKey: sender } = Local.testAccounts[0];

// the zkapp account
let zkappKey = PrivateKey.random();
let zkappAddress = zkappKey.toPublicKey();

// a special account that is allowed to pull out half of the zkapp balance, once
let privilegedKey = PrivateKey.random();
let privilegedAddress = privilegedKey.toPublicKey();

let initialBalance = 10_000_000_000;
let zkapp = new PayoutOnlyOnce(zkappAddress);

// a unique message
let nullifierMessage = Field(5);

console.log('compile');
await PayoutOnlyOnce.compile();

console.log('deploy');
let tx = await Mina.transaction(sender, () => {
  let senderUpdate = AccountUpdate.fundNewAccount(sender);
  senderUpdate.send({ to: zkappAddress, amount: initialBalance });
  zkapp.deploy({ zkappKey });

  zkapp.nullifierRoot.set(NullifierTree.getRoot());
  zkapp.nullifierMessage.set(nullifierMessage);
});
await tx.prove();
await tx.sign([senderKey]).send();

console.log(&nbsp;zkapp balance: ${zkapp.account.balance.get().div(1e9)} MINA&nbsp;);

console.log('generating nullifier');

let jsonNullifier = Nullifier.createTestNullifier(
  [nullifierMessage],
  privilegedKey
);
console.log(jsonNullifier);

console.log('pay out');
tx = await Mina.transaction(sender, () => {
  AccountUpdate.fundNewAccount(sender);
  zkapp.payout(Nullifier.fromJSON(jsonNullifier));
});
await tx.prove();
await tx.sign([senderKey]).send();

console.log(&nbsp;zkapp balance: ${zkapp.account.balance.get().div(1e9)} MINA&nbsp;);
console.log(
  &nbsp;user balance: ${Mina.getAccount(privilegedAddress).balance.div(1e9)} MINA&nbsp;
);

console.log('trying second pay out');

try {
  tx = await Mina.transaction(sender, () => {
    zkapp.payout(Nullifier.fromJSON(jsonNullifier));
  });

  await tx.prove();
  await tx.sign([senderKey]).send();
} catch (error: any) {
  console.log(
    'transaction failed, as expected! received the following error message:'
  );
  console.log(error.message);
}
```

</file>

<file>

## path: /src/examples/plain-html/index.html

url: https://github.com/o1-labs/o1js/blob/main/src/examples/plain-html/index.html

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>hello-snarkyjs</title>
    <script type="importmap">
      { "imports": { "snarkyjs": "./index.js" } }
    </script>
    <script type="module" src="./simple_zkapp.js">
    </script>
  </head>
  <body>
    <div>Check out the console (F12)</div>
  </body>
</html>
```

</file>

<file>

## path: /src/examples/plain-html/server.js

url: https://github.com/o1-labs/o1js/blob/main/src/examples/plain-html/server.js

```
import fs from 'node:fs/promises';
import path from 'node:path';
import http from 'node:http';

const port = 8000;
const defaultHeaders = {
  'content-type': 'text/html',
  'Cross-Origin-Embedder-Policy': 'require-corp',
  'Cross-Origin-Opener-Policy': 'same-origin',
};

const server = http.createServer(async (req, res) => {
  let file = '.' + req.url;
  console.log(file);

  if (file === './') file = './index.html';
  let content;
  try {
    content = await fs.readFile(path.resolve('./dist/web', file), 'utf8');
  } catch (err) {
    res.writeHead(404, defaultHeaders);
    res.write('<html><body>404</body><html>');
    res.end();
    return;
  }

  const extension = path.basename(file).split('.').pop();
  const contentType = {
    html: 'text/html',
    js: 'application/javascript',
    map: 'application/json',
  }[extension];
  const headers = { ...defaultHeaders, 'content-type': contentType };

  res.writeHead(200, headers);
  res.write(content);
  res.end();
});

server.listen(port, () => {
  console.log(&nbsp;Server is running on: http://localhost:${port}&nbsp;);
});
```

</file>

<file>

## path: /src/examples/simple_zkapp.js

url: https://github.com/o1-labs/o1js/blob/main/src/examples/simple_zkapp.js

```
/**
 * Demonstrates how to use o1js in pure JavaScript
 *
 * Decorators &nbsp;@method&nbsp; and &nbsp;@state&nbsp; are replaced by &nbsp;declareState&nbsp;
and &nbsp;declareMethods&nbsp;.
 */
import {
  Field,
  State,
  PrivateKey,
  SmartContract,
  Mina,
  AccountUpdate,
  isReady,
  declareState,
  declareMethods,
  shutdown,
} from 'o1js';

await isReady;

class SimpleZkapp extends SmartContract {
  constructor(address) {
    super(address);
    this.x = State();
  }

  events = { update: Field };

  init() {
    super.init();
    this.x.set(initialState);
  }

  update(y) {
    this.emitEvent('update', y);
    this.emitEvent('update', y);
    this.account.balance.assertEquals(this.account.balance.get());
    let x = this.x.get();
    this.x.assertEquals(x);
    this.x.set(x.add(y));
  }
}
declareState(SimpleZkapp, { x: Field });
declareMethods(SimpleZkapp, { update: [Field] });

let Local = Mina.LocalBlockchain();
Mina.setActiveInstance(Local);

let feePayerKey = Local.testAccounts[0].privateKey;
let feePayer = Local.testAccounts[0].publicKey;
let zkappKey = PrivateKey.random();
let zkappAddress = zkappKey.toPublicKey();

let initialState = Field(1);
let zkapp = new SimpleZkapp(zkappAddress);

console.log('compile');
await SimpleZkapp.compile();

console.log('deploy');
let tx = await Mina.transaction(feePayer, () => {
  AccountUpdate.fundNewAccount(feePayer);
  zkapp.deploy();
});
await tx.sign([feePayerKey, zkappKey]).send();

console.log('initial state: ' + zkapp.x.get());

console.log('update');
tx = await Mina.transaction(feePayer, () => zkapp.update(Field(3)));
await tx.prove();
await tx.sign([feePayerKey]).send();
console.log('final state: ' + zkapp.x.get());

shutdown();
```

</file>

<file>

## path: /src/examples/simple_zkapp.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/simple_zkapp.ts

```
import {
  Field,
  state,
  State,
  method,
  UInt64,
  PrivateKey,
  SmartContract,
  Mina,
  AccountUpdate,
  Bool,
  PublicKey,
} from 'o1js';
import { getProfiler } from './utils/profiler.js';

const doProofs = true;
const beforeGenesis = UInt64.from(Date.now());

class SimpleZkapp extends SmartContract {
  @state(Field) x = State<Field>();

  events = { update: Field, payout: UInt64, payoutReceiver: PublicKey };

  @method init() {
    super.init();
    this.x.set(initialState);
  }

  @method update(y: Field): Field {
    this.account.provedState.assertEquals(Bool(true));
    this.network.timestamp.assertBetween(beforeGenesis, UInt64.MAXINT());
    this.emitEvent('update', y);
    let x = this.x.get();
    this.x.assertEquals(x);
    let newX = x.add(y);
    this.x.set(newX);
    return newX;
  }

  /**
   * This method allows a certain privileged account to claim half of the zkapp balance, but only once
   * @param caller the privileged account
   */
  @method payout(caller: PrivateKey) {
    this.account.provedState.assertEquals(Bool(true));

    // check that caller is the privileged account
    let callerAddress = caller.toPublicKey();
    callerAddress.assertEquals(privilegedAddress);

    // assert that the caller account is new - this way, payout can only happen once
    let callerAccountUpdate = AccountUpdate.create(callerAddress);
    callerAccountUpdate.account.isNew.assertEquals(Bool(true));
    // pay out half of the zkapp balance to the caller
    let balance = this.account.balance.get();
    this.account.balance.assertEquals(balance);
    let halfBalance = balance.div(2);
    this.send({ to: callerAccountUpdate, amount: halfBalance });

    // emit some events
    this.emitEvent('payoutReceiver', callerAddress);
    this.emitEvent('payout', halfBalance);
  }
}

const SimpleProfiler = getProfiler('Simple zkApp');
SimpleProfiler.start('Simple zkApp test flow');
```

```
let Local = Mina.LocalBlockchain({ proofsEnabled: doProofs });
Mina.setActiveInstance(Local);

// a test account that pays all the fees, and puts additional funds into the zkapp
let { privateKey: senderKey, publicKey: sender } = Local.testAccounts[0];

// the zkapp account
let zkappKey = PrivateKey.random();
let zkappAddress = zkappKey.toPublicKey();

// a special account that is allowed to pull out half of the zkapp balance, once
let privilegedKey = PrivateKey.fromBase58(
  'EKEeoESE2A41YQnSht9f7mjiKpJSeZ4jnfHXYatYi8xJdYSxWBep'
);
let privilegedAddress = privilegedKey.toPublicKey();

let initialBalance = 10_000_000_000;
let initialState = Field(1);
let zkapp = new SimpleZkapp(zkappAddress);

if (doProofs) {
  console.log('compile');
  console.time('compile');
  await SimpleZkapp.compile();
  console.timeEnd('compile');
}

console.log('deploy');
let tx = await Mina.transaction(sender, () => {
  let senderUpdate = AccountUpdate.fundNewAccount(sender);
  senderUpdate.send({ to: zkappAddress, amount: initialBalance });
  zkapp.deploy({ zkappKey });
});
await tx.prove();
await tx.sign([senderKey]).send();

console.log('initial state: ' + zkapp.x.get());
console.log(&nbsp;initial balance: ${zkapp.account.balance.get().div(1e9)} MINA&nbsp;);

let account = Mina.getAccount(zkappAddress);
console.log('account state is proved:', account.zkapp?.provedState.toBoolean());

console.log('update');
tx = await Mina.transaction(sender, () => {
  zkapp.update(Field(3));
});
await tx.prove();
await tx.sign([senderKey]).send();

// pay more into the zkapp -- this doesn't need a proof
console.log('receive');
tx = await Mina.transaction(sender, () => {
  let payerAccountUpdate = AccountUpdate.createSigned(sender);
  payerAccountUpdate.send({ to: zkappAddress, amount: UInt64.from(8e9) });
});
await tx.sign([senderKey]).send();

console.log('payout');
tx = await Mina.transaction(sender, () => {
  AccountUpdate.fundNewAccount(sender);
  zkapp.payout(privilegedKey);
});
await tx.prove();
await tx.sign([senderKey]).send();
sender;

console.log('final state: ' + zkapp.x.get());
console.log(&nbsp;final balance: ${zkapp.account.balance.get().div(1e9)} MINA&nbsp;);

console.log('try to payout a second time..');
tx = await Mina.transaction(sender, () => {
  zkapp.payout(privilegedKey);
});
try {
  await tx.prove();
  await tx.sign([senderKey]).send();
} catch (err: any) {
  console.log('Transaction failed with error', err.message);
}

console.log('try to payout to a different account..');
try {
  tx = await Mina.transaction(sender, () => {
    zkapp.payout(Local.testAccounts[2].privateKey);
  });
  await tx.prove();
  await tx.sign([senderKey]).send();
} catch (err: any) {
  console.log('Transaction failed with error', err.message);
}

console.log(
  &nbsp;should still be the same final balance: ${zkapp.account.balance
    .get()
    .div(1e9)} MINA&nbsp;
);

SimpleProfiler.stop().store();
```

</file>

<file>

path: /src/examples/simple_zkapp.web.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/simple_zkapp.web.ts

```
import {
  Field,
  state,
  State,
  method,
  UInt64,
  PrivateKey,
  SmartContract,
  Mina,
  AccountUpdate,
  Bool,
  PublicKey,
} from 'o1js';

const doProofs = true;

const beforeGenesis = UInt64.from(Date.now());

class SimpleZkapp extends SmartContract {
  @state(Field) x = State<Field>();

  events = { update: Field, payout: UInt64, payoutReceiver: PublicKey };

  @method init() {
    super.init();
    this.x.set(initialState);
  }

  @method update(y: Field): Field {
    this.account.provedState.assertEquals(Bool(true));
    this.network.timestamp.assertBetween(beforeGenesis, UInt64.MAXINT());
    this.emitEvent('update', y);
    let x = this.x.get();
    this.x.assertEquals(x);
    let newX = x.add(y);
    this.x.set(newX);
    return newX;
  }

  /**
   * This method allows a certain privileged account to claim half of the zkapp balance, but only once
   * @param caller the privileged account
   */
  @method payout(caller: PrivateKey) {
    this.account.provedState.assertEquals(Bool(true));

    // check that caller is the privileged account
    let callerAddress = caller.toPublicKey();
    callerAddress.assertEquals(privilegedAddress);

    // assert that the caller account is new - this way, payout can only happen once
    let callerAccountUpdate = AccountUpdate.create(callerAddress);
    callerAccountUpdate.account.isNew.assertEquals(Bool(true));
    // pay out half of the zkapp balance to the caller
    let balance = this.account.balance.get();
    this.account.balance.assertEquals(balance);
    let halfBalance = balance.div(2);
    this.send({ to: callerAccountUpdate, amount: halfBalance });

    // emit some events
    this.emitEvent('payoutReceiver', callerAddress);
    this.emitEvent('payout', halfBalance);
  }
}

let Local = Mina.LocalBlockchain({ proofsEnabled: doProofs });
Mina.setActiveInstance(Local);

// a test account that pays all the fees, and puts additional funds into the zkapp
let { privateKey: senderKey, publicKey: sender } = Local.testAccounts[0];

// the zkapp account
let zkappKey = PrivateKey.random();
let zkappAddress = zkappKey.toPublicKey();

// a special account that is allowed to pull out half of the zkapp balance, once
let privilegedKey = PrivateKey.fromBase58(
  'EKEeoESE2A41YQnSht9f7mjiKpJSeZ4jnfHXYatYi8xJdYSxWBep'
);
let privilegedAddress = privilegedKey.toPublicKey();

let initialBalance = 10_000_000_000;
let initialState = Field(1);
let zkapp = new SimpleZkapp(zkappAddress);

if (doProofs) {
  console.log('compile');
  await SimpleZkapp.compile();
}

console.log('deploy');
let tx = await Mina.transaction(sender, () => {
  let senderUpdate = AccountUpdate.fundNewAccount(sender);
  senderUpdate.send({ to: zkappAddress, amount: initialBalance });
  zkapp.deploy({ zkappKey });
});
await tx.prove();
await tx.sign([senderKey]).send();
```

```
console.log('initial state: ' + zkapp.x.get());
console.log(&nbsp;initial balance: ${zkapp.account.balance.get().div(1e9)} MINA&nbsp;);

let account = Mina.getAccount(zkappAddress);
console.log('account state is proved:', account.zkapp?.provedState.toBoolean());

console.log('update');
tx = await Mina.transaction(sender, () => {
  zkapp.update(Field(3));
});
await tx.prove();
await tx.sign([senderKey]).send();

// pay more into the zkapp -- this doesn't need a proof
console.log('receive');
tx = await Mina.transaction(sender, () => {
  let payerAccountUpdate = AccountUpdate.createSigned(sender);
  payerAccountUpdate.send({ to: zkappAddress, amount: UInt64.from(8e9) });
});
await tx.sign([senderKey]).send();

console.log('payout');
tx = await Mina.transaction(sender, () => {
  AccountUpdate.fundNewAccount(sender);
  zkapp.payout(privilegedKey);
});
await tx.prove();
await tx.sign([senderKey]).send();

console.log('final state: ' + zkapp.x.get());
console.log(&nbsp;final balance: ${zkapp.account.balance.get().div(1e9)} MINA&nbsp;);
```

</file>

<file>

## path: /src/examples/simple_zkapp_berkeley.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/simple_zkapp_berkeley.ts

```
/**
 * This is an example for interacting with the Berkeley QANet, directly from o1js.
 *
 * At a high level, it does the following:
 * -) try fetching the account corresponding to the &nbsp;zkappAddress&nbsp; from chain
 * -) if the account doesn't exist or is not a zkapp account yet, deploy a zkapp to it and initialize on-chain
state
 * -) if the zkapp is already deployed, send a state-updating transaction which proves execution of the
"update" method
 */
```

```
import {
  Field,
  state,
  State,
  method,
  PrivateKey,
  SmartContract,
  Mina,
  AccountUpdate,
  isReady,
  shutdown,
  DeployArgs,
  fetchAccount,
} from 'o1js';

await isReady;

// a very simple SmartContract
class SimpleZkapp extends SmartContract {
  @state(Field) x = State<Field>();

  init() {
    super.init();
    this.x.set(initialState);
  }

  @method update(y: Field) {
    let x = this.x.get();
    this.x.assertEquals(x);
    y.assertGreaterThan(0);
    this.x.set(x.add(y));
  }
}

// you can use this with any spec-compliant graphql endpoint
let Berkeley = Mina.Network('https://proxy.berkeley.minaexplorer.com/graphql');
Mina.setActiveInstance(Berkeley);

// to use this test, change this private key to an account which has enough MINA to pay fees
let feePayerKey = PrivateKey.fromBase58(
  'EKEQc95PPQZnMY9d9p1vq1MWLeDJKtvKj4V75UDG3rjnf32BerWD'
);
let feePayerAddress = feePayerKey.toPublicKey();
let response = await fetchAccount({ publicKey: feePayerAddress });
if (response.error) throw Error(response.error.statusText);
let { nonce, balance } = response.account;
console.log(&nbsp;Using fee payer account with nonce ${nonce}, balance ${balance}&nbsp;);

// this used to be an actual zkapp that was deployed and updated with this script:
//
https://berkeley.minaexplorer.com/wallet/B62qpRzFVjd56FiHnNfxokVbcHMQLT119My1FEdSq8ss7KomLiS
Zcan
```

```
// replace this with a new zkapp key if you want to deploy another zkapp
// and please never expose actual private keys in public code repositories like this!
let zkappKey = PrivateKey.fromBase58(
  'EKFQZG2RuLMYyDsC9RGE5Y8gQGefkbUUUyEhFbgRRMHGgoF9eKpY'
);
let zkappAddress = zkappKey.toPublicKey();

let transactionFee = 100_000_000;
let initialState = Field(1);

// compile the SmartContract to get the verification key (if deploying) or cache the provers (if updating)
// this can take a while...
console.log('Compiling smart contract...');
let { verificationKey } = await SimpleZkapp.compile();

// check if the zkapp is already deployed, based on whether the account exists and its first zkapp state is !==
0
let zkapp = new SimpleZkapp(zkappAddress);
let x = await zkapp.x.fetch();
let isDeployed = x?.equals(0).not().toBoolean() ?? false;

// if the zkapp is not deployed yet, create a deploy transaction
if (!isDeployed) {
  console.log(&nbsp;Deploying zkapp for public key ${zkappAddress.toBase58()}.&nbsp;);
  // the &nbsp;transaction()&nbsp; interface is the same as when testing with a local blockchain
  let transaction = await Mina.transaction(
    { sender: feePayerAddress, fee: transactionFee },
    () => {
      AccountUpdate.fundNewAccount(feePayerAddress);
      zkapp.deploy({ verificationKey });
    }
  );
  // if you want to inspect the transaction, you can print it out:
  // console.log(transaction.toGraphqlQuery());

  // send the transaction to the graphql endpoint
  console.log('Sending the transaction...');
  await transaction.sign([feePayerKey, zkappKey]).send();
}

// if the zkapp is not deployed yet, create an update transaction
if (isDeployed) {
  let x = zkapp.x.get();
  console.log(&nbsp;Found deployed zkapp, updating state ${x} -> ${x.add(10)}.&nbsp;);
  let transaction = await Mina.transaction(
    { sender: feePayerAddress, fee: transactionFee },
    () => {
      zkapp.update(Field(10));
    }
  );
  // fill in the proof - this can take a while...
  console.log('Creating an execution proof...');
  await transaction.prove();

  // if you want to inspect the transaction, you can print it out:
  // console.log(transaction.toGraphqlQuery());

  // send the transaction to the graphql endpoint
  console.log('Sending the transaction...');
  await transaction.sign([feePayerKey]).send();
}

shutdown();
```

</file>

<file>

## path: /src/examples/tsconfig.json

url: https://github.com/o1-labs/o1js/blob/main/src/examples/tsconfig.json

```
{
  "extends": "../../tsconfig.json",
  "include": ["."],
  "exclude": [],
  "compilerOptions": {
    "rootDir": ".",
    "baseUrl": "../..",
    "paths": {
      "o1js": ["."]
    }
  }
}
```

</file>

<file>

## path: /src/examples/utils/README.md

url: https://github.com/o1-labs/o1js/blob/main/src/examples/utils/README.md

This folder doesn't contain stand-alone examples, but utilities used in our examples.
</file>

<file>

## path: /src/examples/utils/profiler.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/utils/profiler.ts

```
import fs from 'fs';

export { getProfiler };

const round = (x: number) => Math.round(x * 100) / 100;

function getProfiler(name: string) {
  let times: Record<string, any> = {};
  let label: string;

  return {
    get times() {
      return times;
    },
    start(label_: string) {
      label = label_;
      times = {
        ...times,
        [label]: {
          start: performance.now(),
        },
      };
    },
    stop() {
      times[label].end = performance.now();
      return this;
    },
    store() {
      let profilingData = &nbsp;## Times for ${name}\n\n&nbsp;;
      profilingData += &nbsp;| Name | time passed in s |\n|---|---|&nbsp;;
      let totalTimePassed = 0;

      Object.keys(times).forEach((k) => {
        let timePassed = (times[k].end - times[k].start) / 1000;
        totalTimePassed += timePassed;

        profilingData += &nbsp;\n|${k}|${round(timePassed)}|&nbsp;;
      });

      profilingData += &nbsp;\n\nIn total, it took ${round(
        totalTimePassed
      )} seconds to run the entire benchmark\n\n\n&nbsp;;

      fs.appendFileSync('profiling.md', profilingData);
    },
  };
}
```

</file>

<file>

## path: /src/examples/utils/tic-toc.node.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/utils/tic-toc.node.ts

```
/**
 * Helper for printing timings, in the spirit of Python's &nbsp;tic&nbsp; and &nbsp;toc&nbsp;.
 *
 * This is a slightly nicer version of './tic-tic.ts' which only works in Node.
 */

export { tic, toc };

let timingStack: [string, number][] = [];
let i = 0;

function tic(label = &nbsp;Run command ${i++}&nbsp;) {
  process.stdout.write(&nbsp;${label}... &nbsp;);
  timingStack.push([label, performance.now()]);
}

function toc() {
  let [label, start] = timingStack.pop()!;
  let time = (performance.now() - start) / 1000;
  process.stdout.write(&nbsp;\r${label}... ${time.toFixed(3)} sec\n&nbsp;);
}
```

</file>

<file>

## path: /src/examples/utils/tic-toc.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/utils/tic-toc.ts

```
/**
 * Helper for printing timings, in the spirit of Python's &nbsp;tic&nbsp; and &nbsp;toc&nbsp;.
 */

export { tic, toc };

let timingStack: [string, number][] = [];
let i = 0;

function tic(label = &nbsp;Run command ${i++}&nbsp;) {
  console.log(&nbsp;${label}... &nbsp;);
  timingStack.push([label, performance.now()]);
}

function toc() {
  let [label, start] = timingStack.pop()!;
  let time = (performance.now() - start) / 1000;
  console.log(&nbsp;\r${label}... ${time.toFixed(3)} sec\n&nbsp;);
  return time;
}
```

</file>

<file>

## path: /src/examples/zkapps/composability.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/composability.ts

```
/**
 * zkApps composability
 */
import {
  Field,
  isReady,
  method,
  Mina,
  AccountUpdate,
  PrivateKey,
  SmartContract,
  state,
  State,
} from 'o1js';
import { getProfiler } from '../utils/profiler.js';

const doProofs = true;

await isReady;

// contract which can add 1 to a number
class Incrementer extends SmartContract {
  @method increment(x: Field): Field {
    return x.add(1);
  }
}

// contract which can add two numbers, plus 1, and return the result
// incrementing by one is outsourced to another contract (it's cleaner that way, we want to stick to the single
responsibility principle)
class Adder extends SmartContract {
  @method addPlus1(x: Field, y: Field): Field {
    // compute result
    let sum = x.add(y);
    // call the other contract to increment
    let incrementer = new Incrementer(incrementerAddress);
    return incrementer.increment(sum);
  }
}

// contract which calls the Adder, stores the result on chain & emits an event
class Caller extends SmartContract {
  @state(Field) sum = State<Field>();
  events = { sum: Field };

  @method callAddAndEmit(x: Field, y: Field) {
    let adder = new Adder(adderAddress);
    let sum = adder.addPlus1(x, y);
    this.emitEvent('sum', sum);
    this.sum.set(sum);
  }
}

const ComposabilityProfiler = getProfiler('Composability zkApp');
ComposabilityProfiler.start('Composability test flow');
// script to deploy zkapps and do interactions

let Local = Mina.LocalBlockchain({ proofsEnabled: doProofs });
Mina.setActiveInstance(Local);

// a test account that pays all the fees, and puts additional funds into the zkapp
let feePayerKey = Local.testAccounts[0].privateKey;
let feePayer = Local.testAccounts[0].publicKey;

// the first contract's address
let incrementerKey = PrivateKey.random();
let incrementerAddress = incrementerKey.toPublicKey();
// the second contract's address
let adderKey = PrivateKey.random();
let adderAddress = adderKey.toPublicKey();
// the third contract's address
let zkappKey = PrivateKey.random();
let zkappAddress = zkappKey.toPublicKey();
let zkapp = new Caller(zkappAddress);
let adderZkapp = new Adder(adderAddress);
let incrementerZkapp = new Incrementer(incrementerAddress);

if (doProofs) {
  console.log('compile (incrementer)');
  await Incrementer.compile();
  console.log('compile (adder)');
  await Adder.compile();
  console.log('compile (caller)');
  await Caller.compile();
}

console.log('deploy');
let tx = await Mina.transaction(feePayer, () => {
  // TODO: enable funding multiple accounts properly
  AccountUpdate.fundNewAccount(feePayer, 3);
  zkapp.deploy();
  adderZkapp.deploy();
  incrementerZkapp.deploy();
});
await tx.sign([feePayerKey, zkappKey, adderKey, incrementerKey]).send();

console.log('call interaction');
tx = await Mina.transaction(feePayer, () => {
  // we just call one contract here, nothing special to do
  zkapp.callAddAndEmit(Field(5), Field(6));
});
console.log('proving (3 proofs.. can take a bit!)');
await tx.prove();
console.log(tx.toPretty());
await tx.sign([feePayerKey]).send();

// should hopefully be 12 since we added 5 + 6 + 1
console.log('state: ' + zkapp.sum.get());
ComposabilityProfiler.stop().store();
```

</file>

<file>

## path: /src/examples/zkapps/dex/arbitrary_token_interaction.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/dex/arbitrary_token_interaction.ts

```
import {
  isReady,
  Mina,
  AccountUpdate,
  UInt64,
  shutdown,
  TokenId,
} from 'o1js';
import { TokenContract, addresses, keys, tokenIds } from './dex.js';

await isReady;
let doProofs = true;

let Local = Mina.LocalBlockchain({ proofsEnabled: doProofs });
Mina.setActiveInstance(Local);
let accountFee = Mina.accountCreationFee();

let [{ privateKey: userKey, publicKey: userAddress }] = Local.testAccounts;
let tx;

console.log('-------------------------------------------------');
console.log('TOKEN X ADDRESS\t', addresses.tokenX.toBase58());
console.log('USER ADDRESS\t', userAddress.toBase58());
console.log('-------------------------------------------------');
console.log('TOKEN X ID\t', TokenId.toBase58(tokenIds.X));
console.log('-------------------------------------------------');

// compile & deploy all 5 zkApps
console.log('compile (token)...');
await TokenContract.compile();

let tokenX = new TokenContract(addresses.tokenX);

console.log('deploy & init token contracts...');
tx = await Mina.transaction(userKey, () => {
  // pay fees for creating 2 token contract accounts, and fund them so each can create 1 account themselves
  let feePayerUpdate = AccountUpdate.createSigned(userKey);
  feePayerUpdate.balance.subInPlace(accountFee.mul(1));
  tokenX.deploy();
});
await tx.prove();
tx.sign([keys.tokenX]);
await tx.send();

console.log('arbitrary token minting...');
tx = await Mina.transaction(userKey, () => {
  // pay fees for creating user's token X account
  AccountUpdate.createSigned(userKey).balance.subInPlace(accountFee.mul(1));
  //           mint any number of tokens to our account
  let tokenContract = new TokenContract(addresses.tokenX);
  tokenContract.token.mint({
    address: userAddress,
    amount: UInt64.from(1e18),
  });
});
await tx.prove();
console.log(tx.toPretty());
await tx.send();
```

```
console.log(
  'User tokens: ',
  Mina.getBalance(userAddress, tokenIds.X).value.toBigInt()
);

shutdown();
```

</file>

<file>

## path: /src/examples/zkapps/dex/dex-with-actions.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/dex/dex-with-actions.ts

```
/**
 * This DEX implementation differs from ./dex.ts in two ways:
 * - More minimal & realistic; stuff designed only for testing protocol features was removed
 * - Uses an async pattern with actions that lets users claim funds later and reduces account updates
 */
import {
  Account,
  method,
  AccountUpdate,
  PublicKey,
  SmartContract,
  UInt64,
  Struct,
  State,
  state,
  TokenId,
  Reducer,
  Field,
  Permissions,
  isReady,
  Mina,
  InferProvable,
  Provable,
} from 'o1js';

import { TokenContract, randomAccounts } from './dex.js';

export { Dex, DexTokenHolder, addresses, keys, tokenIds, getTokenBalances };

class RedeemAction extends Struct({ address: PublicKey, dl: UInt64 }) {}

class Dex extends SmartContract {
  // addresses of token contracts are constants
  tokenX = addresses.tokenX;
  tokenY = addresses.tokenY;
```

```
/**
  * state that keeps track of total lqXY supply -- this is needed to calculate what to return when redeeming
liquidity
  *
  * total supply is initially zero; it increases when supplying liquidity and decreases when redeeming it
  */
@state(UInt64) totalSupply = State<UInt64>();

/**
  * redeeming liquidity is a 2-step process leveraging actions, to get past the account update limit
  */
reducer = Reducer({ actionType: RedeemAction });

events = {
  'supply-liquidity': Struct({ address: PublicKey, dx: UInt64, dy: UInt64 }),
  'redeem-liquidity': Struct({ address: PublicKey, dl: UInt64 }),
};
// better-typed wrapper for &nbsp;this.emitEvent()&nbsp;. TODO: remove after fixing event typing
get typedEvents() {
  return getTypedEvents<Dex>(this);
}

/**
  * Initialization. _All_ permissions are set to impossible except the explicitly required permissions.
  */
init() {
  super.init();
  let proof = Permissions.proof();
  this.account.permissions.set({
    ...Permissions.allImpossible(),
    access: proof,
    editState: proof,
    editActionState: proof,
    send: proof,
  });
}

@method createAccount() {
  this.token.mint({ address: this.sender, amount: UInt64.from(0) });
}

/**
  * Mint liquidity tokens in exchange for X and Y tokens
  * @param dx input amount of X tokens
  * @param dy input amount of Y tokens
  * @return output amount of lqXY tokens
  *
  * This function fails if the X and Y token amounts don't match the current X/Y ratio in the pool.
  * This can also be used if the pool is empty. In that case, there is no check on X/Y;
  * instead, the input X and Y amounts determine the initial ratio.
  */
```

```
@method supplyLiquidityBase(dx: UInt64, dy: UInt64): UInt64 {
  let user = this.sender;
  let tokenX = new TokenContract(this.tokenX);
  let tokenY = new TokenContract(this.tokenY);

  // get balances of X and Y token
  let dexX = AccountUpdate.create(this.address, tokenX.token.id);
  let x = dexX.account.balance.getAndAssertEquals();

  let dexY = AccountUpdate.create(this.address, tokenY.token.id);
  let y = dexY.account.balance.getAndAssertEquals();

  // // assert dy === [dx * y/x], or x === 0
  let isXZero = x.equals(UInt64.zero);
  let xSafe = Provable.if(isXZero, UInt64.one, x);
  let isDyCorrect = dy.equals(dx.mul(y).div(xSafe));
  isDyCorrect.or(isXZero).assertTrue();

  tokenX.transfer(user, dexX, dx);
  tokenY.transfer(user, dexY, dy);

  // calculate liquidity token output simply as dl = dx + dx
  // => maintains ratio x/l, y/l
  let dl = dy.add(dx);
  this.token.mint({ address: user, amount: dl });

  // update l supply
  let l = this.totalSupply.get();
  this.totalSupply.assertEquals(l);
  this.totalSupply.set(l.add(dl));

  // emit event
  this.typedEvents.emit('supply-liquidity', { address: user, dx, dy });
  return dl;
}

/**
 * Mint liquidity tokens in exchange for X and Y tokens
 * @param dx input amount of X tokens
 * @return output amount of lqXY tokens
 *
 * This uses supplyLiquidityBase as the circuit, but for convenience,
 * the input amount of Y tokens is calculated automatically from the X tokens.
 * Fails if the liquidity pool is empty, so can't be used for the first deposit.
 */
supplyLiquidity(dx: UInt64): UInt64 {
  // calculate dy outside circuit
  let x = Account(this.address, TokenId.derive(this.tokenX)).balance.get();
  let y = Account(this.address, TokenId.derive(this.tokenY)).balance.get();
  if (x.value.isConstant() && x.value.isZero().toBoolean()) {
    throw Error(
      'Cannot call &nbsp;supplyLiquidity&nbsp; when reserves are zero. Use
&nbsp;supplyLiquidityBase&nbsp;.'
    );
  }
  let dy = dx.mul(y).div(x);
  return this.supplyLiquidityBase(dx, dy);
}

/**
 * Burn liquidity tokens to get back X and Y tokens
 * @param dl input amount of lqXY token
 *
 * The transaction needs to be signed by the user's private key.
 *
 * NOTE: this does not give back tokens in return for liquidity right away.
 * to get back the tokens, you have to call {@link DexTokenHolder}.redeemFinalize()
 * on both token holder contracts, after &nbsp;redeemInitialize()&nbsp; has been accepted into a block.
 *
 * @emits RedeemAction - action on the Dex account that will make the token holder
 * contracts pay you tokens when reducing the action.
 */
@method redeemInitialize(dl: UInt64) {
  this.reducer.dispatch(new RedeemAction({ address: this.sender, dl }));
  this.token.burn({ address: this.sender, amount: dl });
  // TODO: preconditioning on the state here ruins concurrent interactions,
  // there should be another &nbsp;finalize&nbsp; DEX method which reduces actions & updates state
  this.totalSupply.set(this.totalSupply.getAndAssertEquals().sub(dl));

  // emit event
  this.typedEvents.emit('redeem-liquidity', { address: this.sender, dl });
}

/**
 * Helper for &nbsp;DexTokenHolder.redeemFinalize()&nbsp; which adds preconditions on
 * the current action state and token supply
 */
@method assertActionsAndSupply(actionState: Field, totalSupply: UInt64) {
  this.account.actionState.assertEquals(actionState);
  this.totalSupply.assertEquals(totalSupply);
}

/**
 * Swap X tokens for Y tokens
 * @param dx input amount of X tokens
 * @return output amount Y tokens
 *
 * The transaction needs to be signed by the user's private key.
 *
 * Note: this is not a &nbsp;@method&nbsp;, since it doesn't do anything beyond
 * the called methods which requires proof authorization.
 */
swapX(dx: UInt64): UInt64 {
  let tokenY = new TokenContract(this.tokenY);
```

```
    let dexY = new DexTokenHolder(this.address, tokenY.token.id);
    let dy = dexY.swap(this.sender, dx, this.tokenX);
    tokenY.approveUpdateAndSend(dexY.self, this.sender, dy);
    return dy;
  }

  /**
   * Swap Y tokens for X tokens
   * @param dy input amount of Y tokens
   * @return output amount Y tokens
   *
   * The transaction needs to be signed by the user's private key.
   *
   * Note: this is not a &nbsp;@method&nbsp;, since it doesn't do anything beyond
   * the called methods which requires proof authorization.
   */
  swapY(dy: UInt64): UInt64 {
    let tokenX = new TokenContract(this.tokenX);
    let dexX = new DexTokenHolder(this.address, tokenX.token.id);
    let dx = dexX.swap(this.sender, dy, this.tokenY);
    tokenX.approveUpdateAndSend(dexX.self, this.sender, dx);
    return dx;
  }

  @method transfer(from: PublicKey, to: PublicKey, amount: UInt64) {
    this.token.send({ from, to, amount });
  }
}

class DexTokenHolder extends SmartContract {
  @state(Field) redeemActionState = State<Field>();
  static redeemActionBatchSize = 5;

  events = {
    swap: Struct({ address: PublicKey, dx: UInt64 }),
  };
  // better-typed wrapper for &nbsp;this.emitEvent()&nbsp;. TODO: remove after fixing event typing
  get typedEvents() {
    return getTypedEvents<DexTokenHolder>(this);
  }

  init() {
    super.init();
    this.redeemActionState.set(Reducer.initialActionState);
  }

  @method redeemLiquidityFinalize() {
    // get redeem actions
    let dex = new Dex(this.address);
    let fromActionState = this.redeemActionState.getAndAssertEquals();
    let actions = dex.reducer.getActions({ fromActionState });
    // get total supply of liquidity tokens _before_ applying these actions
    // (each redeem action _decreases_ the supply, so we increase it here)
    let l = Provable.witness(UInt64, (): UInt64 => {
      let l = dex.totalSupply.get().toBigInt();
      // dex.totalSupply.assertNothing();
      for (let [action] of actions) {
        l += action.dl.toBigInt();
      }
      return UInt64.from(l);
    });

    // get our token balance
    let x = this.account.balance.getAndAssertEquals();

    let redeemActionState = dex.reducer.forEach(
      actions,
      ({ address, dl }) => {
        // for every user that redeemed liquidity, we calculate the token output
        // and create a child account update which pays the user
        let dx = x.mul(dl).div(l);
        let receiver = this.send({ to: address, amount: dx });
        // note: this should just work when the reducer gives us dummy data

        // important: these child account updates inherit token permission from us
        receiver.body.mayUseToken = AccountUpdate.MayUseToken.InheritFromParent;

        // update l and x accordingly
        l = l.sub(dl);
        x = x.add(dx);
      },
      fromActionState,
      {
        maxTransactionsWithActions: DexTokenHolder.redeemActionBatchSize,
        // DEX contract doesn't allow setting preconditions from outside (= w/o proof)
        skipActionStatePrecondition: true,
      }
    );

    // update action state so these payments can't be triggered a 2nd time
    this.redeemActionState.set(redeemActionState);

    // precondition on the DEX contract, to prove we used the right actions & token supply
    dex.assertActionsAndSupply(redeemActionState, l);
  }

  // this works for both directions (in our case where both tokens use the same contract)
  @method swap(
    user: PublicKey,
    otherTokenAmount: UInt64,
    otherTokenAddress: PublicKey
  ): UInt64 {
    // we're writing this as if our token === y and other token === x
```

```
    let dx = otherTokenAmount;
    let tokenX = new TokenContract(otherTokenAddress);

    // get balances of X and Y token
    let dexX = AccountUpdate.create(this.address, tokenX.token.id);
    let x = dexX.account.balance.getAndAssertEquals();
    let y = this.account.balance.getAndAssertEquals();

    // send x from user to us (i.e., to the same address as this but with the other token)
    tokenX.transfer(user, dexX, dx);

    // compute and send dy
    let dy = y.mul(dx).div(x.add(dx));
    // just subtract dy balance and let adding balance be handled one level higher
    this.balance.subInPlace(dy);

    // emit event
    this.typedEvents.emit('swap', { address: this.sender, dx });

    return dy;
  }
}

await isReady;
let { keys, addresses } = randomAccounts(
  false,
  'tokenX',
  'tokenY',
  'dex',
  'user'
);
let tokenIds = {
  X: TokenId.derive(addresses.tokenX),
  Y: TokenId.derive(addresses.tokenY),
  lqXY: TokenId.derive(addresses.dex),
};

/**
 * Helper to get the various token balances for checks in tests
 */
function getTokenBalances() {
  let balances = {
    user: { MINA: 0n, X: 0n, Y: 0n, lqXY: 0n },
    dex: { X: 0n, Y: 0n, lqXYSupply: 0n },
  };
  for (let user of ['user'] as const) {
    try {
      balances[user].MINA =
        Mina.getBalance(addresses[user]).toBigInt() / 1_000_000_000n;
    } catch {}
    for (let token of ['X', 'Y', 'lqXY'] as const) {
      try {
        balances[user][token] = Mina.getBalance(
          addresses[user],
          tokenIds[token]
        ).toBigInt();
      } catch {}
    }
  }
  try {
    balances.dex.X = Mina.getBalance(addresses.dex, tokenIds.X).toBigInt();
  } catch {}
  try {
    balances.dex.Y = Mina.getBalance(addresses.dex, tokenIds.Y).toBigInt();
  } catch {}
  try {
    let dex = new Dex(addresses.dex);
    balances.dex.lqXYSupply = dex.totalSupply.get().toBigInt();
  } catch {}
  return balances;
}

function getTypedEvents<Contract extends SmartContract>(contract: Contract) {
  return {
    emit<Key extends keyof Contract['events']>(
      key: Key,
      event: InferProvable<Contract['events'][Key]>
    ) {
      contract.emitEvent(key, event);
    },
  };
}
```

</file>

<file>

## path: /src/examples/zkapps/dex/dex.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/dex/dex.ts

```
import {
  Account,
  Bool,
  Circuit,
  DeployArgs,
  Field,
  Int64,
  isReady,
  method,
  Mina,
  AccountUpdate,
  Permissions,
  PrivateKey,
  PublicKey,
  SmartContract,
  UInt64,
  VerificationKey,
  Struct,
  State,
  state,
  UInt32,
  TokenId,
  Provable,
} from 'o1js';

export { createDex, TokenContract, keys, addresses, tokenIds, randomAccounts };

class UInt64x2 extends Struct([UInt64, UInt64]) {}

function createDex({
  lockedLiquiditySlots,
}: { lockedLiquiditySlots?: number } = {}) {
  class Dex extends SmartContract {
    // addresses of token contracts are constants
    tokenX = addresses.tokenX;
    tokenY = addresses.tokenY;

    /**
     * state which keeps track of total lqXY supply -- this is needed to calculate what to return when
redeeming liquidity
     *
     * total supply is zero initially; it increases when supplying liquidity and decreases when redeeming it
     */
    @state(UInt64) totalSupply = State<UInt64>();

    /**
     * Mint liquidity tokens in exchange for X and Y tokens
     * @param dx input amount of X tokens
     * @param dy input amount of Y tokens
     * @return output amount of lqXY tokens
     *
     * This function fails if the X and Y token amounts don't match the current X/Y ratio in the pool.
     * This can also be used if the pool is empty. In that case, there is no check on X/Y;
     * instead, the input X and Y amounts determine the initial ratio.
     */
    @method supplyLiquidityBase(dx: UInt64, dy: UInt64): UInt64 {
      let user = this.sender;
      let tokenX = new TokenContract(this.tokenX);
      let tokenY = new TokenContract(this.tokenY);

      // get balances of X and Y token
      // TODO: this creates extra account updates. we need to reuse these by passing them to or returning
them from transfer()
      // but for that, we need the @method argument generalization
      let dexXUpdate = AccountUpdate.create(this.address, tokenX.token.id);
      let dexXBalance = dexXUpdate.account.balance.getAndAssertEquals();

      let dexYUpdate = AccountUpdate.create(this.address, tokenY.token.id);
      let dexYBalance = dexYUpdate.account.balance.getAndAssertEquals();

      // // assert dy === [dx * y/x], or x === 0
      let isXZero = dexXBalance.equals(UInt64.zero);
      let xSafe = Provable.if(isXZero, UInt64.one, dexXBalance);
      let isDyCorrect = dy.equals(dx.mul(dexYBalance).div(xSafe));
      isDyCorrect.or(isXZero).assertTrue();

      tokenX.transfer(user, dexXUpdate, dx);
      tokenY.transfer(user, dexYUpdate, dy);

      // calculate liquidity token output simply as dl = dx + dx
      // => maintains ratio x/l, y/l
      let dl = dy.add(dx);
      let userUpdate = this.token.mint({ address: user, amount: dl });
      if (lockedLiquiditySlots !== undefined) {
        /**
         * exercise the "timing" (vesting) feature to lock the received liquidity tokens.
         *
         * THIS IS HERE FOR TESTING!
         *
         * In reality, the timing feature is a bit awkward to use for time-locking liquidity tokens.
         * That's because, if there is currently a vesting schedule on an account, we can't modify it.
         * Thus, a liquidity provider would need to wait for their current tokens to unlock before being able to
         * supply liquidity again (or, create another account to supply liquidity from).
         */
        let amountLocked = dl;
        userUpdate.account.timing.set({
          initialMinimumBalance: amountLocked,
          cliffAmount: amountLocked,
          cliffTime: UInt32.from(lockedLiquiditySlots),
          vestingIncrement: UInt64.zero,
          vestingPeriod: UInt32.one,
        });
        userUpdate.requireSignature();
      }

      // update l supply
      let l = this.totalSupply.get();
      this.totalSupply.assertEquals(l);
      this.totalSupply.set(l.add(dl));
      return dl;
    }

    /**
     * Mint liquidity tokens in exchange for X and Y tokens
     * @param dx input amount of X tokens
     * @return output amount of lqXY tokens
```

```
   *
   * This uses supplyLiquidityBase as the circuit, but for convenience,
   * the input amount of Y tokens is calculated automatically from the X tokens.
   * Fails if the liquidity pool is empty, so can't be used for the first deposit.
   */
  supplyLiquidity(dx: UInt64): UInt64 {
    // calculate dy outside circuit
    let x = Account(this.address, TokenId.derive(this.tokenX)).balance.get();
    let y = Account(this.address, TokenId.derive(this.tokenY)).balance.get();
    if (x.value.isConstant() && x.value.isZero().toBoolean()) {
      throw Error(
        'Cannot call &nbsp;supplyLiquidity&nbsp; when reserves are zero. Use
&nbsp;supplyLiquidityBase&nbsp;.'
      );
    }
    let dy = dx.mul(y).div(x);
    return this.supplyLiquidityBase(dx, dy);
  }

  /**
   * Burn liquidity tokens to get back X and Y tokens
   * @param dl input amount of lqXY token
   * @return output amount of X and Y tokens, as a tuple [outputX, outputY]
   *
   * The transaction needs to be signed by the user's private key.
   *
   * Note: this is not a &nbsp;@method&nbsp; because there's nothing to prove which isn't already proven
   * by the called methods
   */
  redeemLiquidity(dl: UInt64) {
    // call the token X holder inside a token X-approved callback
    let tokenX = new TokenContract(this.tokenX);
    let dexX = new DexTokenHolder(this.address, tokenX.token.id);
    let dxdy = dexX.redeemLiquidity(this.sender, dl, this.tokenY);
    let dx = dxdy[0];
    tokenX.approveUpdateAndSend(dexX.self, this.sender, dx);
    return dxdy;
  }

  /**
   * Swap X tokens for Y tokens
   * @param dx input amount of X tokens
   * @return output amount Y tokens
   *
   * The transaction needs to be signed by the user's private key.
   */
  @method swapX(dx: UInt64): UInt64 {
    let tokenY = new TokenContract(this.tokenY);
    let dexY = new DexTokenHolder(this.address, tokenY.token.id);
    let dy = dexY.swap(this.sender, dx, this.tokenX);
    tokenY.approveUpdateAndSend(dexY.self, this.sender, dy);
    return dy;
  }

  /**
   * Swap Y tokens for X tokens
   * @param dy input amount of Y tokens
   * @return output amount Y tokens
   *
   * The transaction needs to be signed by the user's private key.
   */
  @method swapY(dy: UInt64): UInt64 {
    let tokenX = new TokenContract(this.tokenX);
    let dexX = new DexTokenHolder(this.address, tokenX.token.id);
    let dx = dexX.swap(this.sender, dy, this.tokenY);
    tokenX.approveUpdateAndSend(dexX.self, this.sender, dx);
    return dx;
  }

  /**
   * helper method to approve burning of user's liquidity.
   * this just burns user tokens, so there is no incentive to call this directly.
   * instead, the dex token holders call this and in turn pay back tokens.
   *
   * @param user caller address
   * @param dl input amount of lq tokens
   * @returns total supply of lq tokens _before_ burning dl, so that caller can calculate how much dx / dx to
returns
   *
   * The transaction needs to be signed by the user's private key.
   */
  @method burnLiquidity(user: PublicKey, dl: UInt64): UInt64 {
    // this makes sure there is enough l to burn (user balance stays >= 0), so l stays >= 0, so l was >0 before
    this.token.burn({ address: user, amount: dl });
    let l = this.totalSupply.get();
    this.totalSupply.assertEquals(l);
    this.totalSupply.set(l.sub(dl));
    return l;
  }

  @method transfer(from: PublicKey, to: PublicKey, amount: UInt64) {
    this.token.send({ from, to, amount });
  }
}

class ModifiedDex extends Dex {
  @method swapX(dx: UInt64): UInt64 {
    let tokenY = new TokenContract(this.tokenY);
    let dexY = new ModifiedDexTokenHolder(this.address, tokenY.token.id);
    let dy = dexY.swap(this.sender, dx, this.tokenX);
    tokenY.approveUpdateAndSend(dexY.self, this.sender, dy);
    return dy;
  }
}
```

```
 class DexTokenHolder extends SmartContract {
   // simpler circuit for redeeming liquidity -- direct trade between our token and lq token
   // it's incomplete, as it gives the user only the Y part for an lqXY token; but doesn't matter as there's no
incentive to call it directly
   // see the more complicated method &nbsp;redeemLiquidity&nbsp; below which gives back both tokens,
by calling this method,
   // for the other token, in a callback
   @method redeemLiquidityPartial(user: PublicKey, dl: UInt64): UInt64x2 {
     // user burns dl, approved by the Dex main contract
     let dex = new Dex(addresses.dex);
     let l = dex.burnLiquidity(user, dl);

     // in return, we give dy back
     let y = this.account.balance.get();
     this.account.balance.assertEquals(y);
     // we can safely divide by l here because the Dex contract logic wouldn't allow burnLiquidity if not l>0
     let dy = y.mul(dl).div(l);
     // just subtract the balance, user gets their part one level higher
     this.balance.subInPlace(dy);

     // be approved by the token owner parent
     this.self.body.mayUseToken = AccountUpdate.MayUseToken.ParentsOwnToken;

     // return l, dy so callers don't have to walk their child account updates to get it
     return [l, dy];
   }

   // more complicated circuit, where we trigger the Y(other)-lqXY trade in our child account updates and
then add the X(our) part
   @method redeemLiquidity(
     user: PublicKey,
     dl: UInt64,
     otherTokenAddress: PublicKey
   ): UInt64x2 {
     // first call the Y token holder, approved by the Y token contract; this makes sure we get dl, the user's
lqXY
     let tokenY = new TokenContract(otherTokenAddress);
     let dexY = new DexTokenHolder(this.address, tokenY.token.id);
     let result = dexY.redeemLiquidityPartial(user, dl);
     let l = result[0];
     let dy = result[1];
     tokenY.approveUpdateAndSend(dexY.self, user, dy);

     // in return for dl, we give back dx, the X token part
     let x = this.account.balance.get();
     this.account.balance.assertEquals(x);
     let dx = x.mul(dl).div(l);
     // just subtract the balance, user gets their part one level higher
     this.balance.subInPlace(dx);

     return [dx, dy];
```

```
  }

  // this works for both directions (in our case where both tokens use the same contract)
  @method swap(
    user: PublicKey,
    otherTokenAmount: UInt64,
    otherTokenAddress: PublicKey
  ): UInt64 {
    // we're writing this as if our token === y and other token === x
    let dx = otherTokenAmount;
    let tokenX = new TokenContract(otherTokenAddress);
    // get balances
    let x = tokenX.getBalance(this.address);
    let y = this.account.balance.get();
    this.account.balance.assertEquals(y);
    // send x from user to us (i.e., to the same address as this but with the other token)
    tokenX.transfer(user, this.address, dx);
    // compute and send dy
    let dy = y.mul(dx).div(x.add(dx));
    // just subtract dy balance and let adding balance be handled one level higher
    this.balance.subInPlace(dy);
    return dy;
  }
}

class ModifiedDexTokenHolder extends DexTokenHolder {
  /**
   * This swap method has a slightly changed formula
   */
  @method swap(
    user: PublicKey,
    otherTokenAmount: UInt64,
    otherTokenAddress: PublicKey
  ): UInt64 {
    let dx = otherTokenAmount;
    let tokenX = new TokenContract(otherTokenAddress);
    let x = tokenX.getBalance(this.address);
    let y = this.account.balance.get();
    this.account.balance.assertEquals(y);
    tokenX.transfer(user, this.address, dx);

    // this formula has been changed - we just give the user an additional 15 token
    let dy = y.mul(dx).div(x.add(dx)).add(15);

    this.balance.subInPlace(dy);
    return dy;
  }
}

/**
 * Helper to get the various token balances for checks in tests
 */
```

```
function getTokenBalances() {
  let balances = {
    user: { MINA: 0n, X: 0n, Y: 0n, lqXY: 0n },
    user2: { MINA: 0n, X: 0n, Y: 0n, lqXY: 0n },
    dex: { X: 0n, Y: 0n },
    tokenContract: { X: 0n, Y: 0n },
    total: { lqXY: 0n },
  };
  for (let user of ['user', 'user2'] as const) {
    try {
      balances[user].MINA =
        Mina.getBalance(addresses[user]).toBigInt() / 1_000_000_000n;
    } catch {}
    for (let token of ['X', 'Y', 'lqXY'] as const) {
      try {
        balances[user][token] = Mina.getBalance(
          addresses[user],
          tokenIds[token]
        ).toBigInt();
      } catch {}
    }
  }
  try {
    balances.dex.X = Mina.getBalance(addresses.dex, tokenIds.X).toBigInt();
  } catch {}
  try {
    balances.dex.Y = Mina.getBalance(addresses.dex, tokenIds.Y).toBigInt();
  } catch {}
  try {
    balances.tokenContract.X = Mina.getBalance(
      addresses.tokenX,
      tokenIds.X
    ).toBigInt();
  } catch {}
  try {
    balances.tokenContract.Y = Mina.getBalance(
      addresses.tokenY,
      tokenIds.Y
    ).toBigInt();
  } catch {}
  try {
    let dex = new Dex(addresses.dex);
    balances.total.lqXY = dex.totalSupply.get().toBigInt();
  } catch {}
  return balances;
}

return {
  Dex,
  DexTokenHolder,
  ModifiedDexTokenHolder,
  ModifiedDex,
    getTokenBalances,
  };
}

/**
 * Simple token with API flexible enough to handle all our use cases
 */
class TokenContract extends SmartContract {
  deploy(args?: DeployArgs) {
    super.deploy(args);
    this.account.permissions.set({
      ...Permissions.default(),
      access: Permissions.proofOrSignature(),
    });
  }
  @method init() {
    super.init();
    // mint the entire supply to the token account with the same address as this contract
    /**
     * DUMB STUFF FOR TESTING (change in real app)
     *
     * we mint the max uint64 of tokens here, so that we can overflow it in tests if we just mint a bit more
     */
    let receiver = this.token.mint({
      address: this.address,
      amount: UInt64.MAXINT(),
    });
    // assert that the receiving account is new, so this can be only done once
    receiver.account.isNew.assertEquals(Bool(true));
    // pay fees for opened account
    this.balance.subInPlace(Mina.accountCreationFee());
  }

  /**
   * DUMB STUFF FOR TESTING (delete in real app)
   *
   * mint additional tokens to some user, so we can overflow token balances
   */
  @method init2() {
    let receiver = this.token.mint({
      address: addresses.user,
      amount: UInt64.from(10n ** 6n),
    });
    // assert that the receiving account is new, so this can be only done once
    receiver.account.isNew.assertEquals(Bool(true));
    // pay fees for opened account
    this.balance.subInPlace(Mina.accountCreationFee());
  }

  // this is a very standardized deploy method. instead, we could also take the account update from a
callback
  // => need callbacks for signatures
```

```
@method deployZkapp(address: PublicKey, verificationKey: VerificationKey) {
  let tokenId = this.token.id;
  let zkapp = AccountUpdate.create(address, tokenId);
  zkapp.account.permissions.set(Permissions.default());
  zkapp.account.verificationKey.set(verificationKey);
  zkapp.requireSignature();
}

@method approveUpdate(zkappUpdate: AccountUpdate) {
  this.approve(zkappUpdate);
  let balanceChange = Int64.fromObject(zkappUpdate.body.balanceChange);
  balanceChange.assertEquals(Int64.from(0));
}

// FIXME: remove this
@method approveAny(zkappUpdate: AccountUpdate) {
  this.approve(zkappUpdate, AccountUpdate.Layout.AnyChildren);
}

// let a zkapp send tokens to someone, provided the token supply stays constant
@method approveUpdateAndSend(
  zkappUpdate: AccountUpdate,
  to: PublicKey,
  amount: UInt64
) {
  // TODO: THIS IS INSECURE. The proper version has a prover error (compile != prove) that must be fixed
  this.approve(zkappUpdate, AccountUpdate.Layout.AnyChildren);

  // THIS IS HOW IT SHOULD BE DONE:
  // // approve a layout of two grandchildren, both of which can't inherit the token permission
  // let { StaticChildren, AnyChildren } = AccountUpdate.Layout;
  // this.approve(zkappUpdate, StaticChildren(AnyChildren, AnyChildren));
  // zkappUpdate.body.mayUseToken.parentsOwnToken.assertTrue();
  // let [grandchild1, grandchild2] = zkappUpdate.children.accountUpdates;
  // grandchild1.body.mayUseToken.inheritFromParent.assertFalse();
  // grandchild2.body.mayUseToken.inheritFromParent.assertFalse();

  // see if balance change cancels the amount sent
  let balanceChange = Int64.fromObject(zkappUpdate.body.balanceChange);
  balanceChange.assertEquals(Int64.from(amount).neg());
  // add same amount of tokens to the receiving address
  this.token.mint({ address: to, amount });
}

transfer(from: PublicKey, to: PublicKey | AccountUpdate, amount: UInt64) {
  if (to instanceof PublicKey)
    return this.transferToAddress(from, to, amount);
  if (to instanceof AccountUpdate)
    return this.transferToUpdate(from, to, amount);
}
@method transferToAddress(from: PublicKey, to: PublicKey, value: UInt64) {
  this.token.send({ from, to, amount: value });
  }
  @method transferToUpdate(from: PublicKey, to: AccountUpdate, value: UInt64) {
    this.token.send({ from, to, amount: value });
  }

  @method getBalance(publicKey: PublicKey): UInt64 {
    let accountUpdate = AccountUpdate.create(publicKey, this.token.id);
    let balance = accountUpdate.account.balance.get();
    accountUpdate.account.balance.assertEquals(
      accountUpdate.account.balance.get()
    );
    return balance;
  }
}

const savedKeys = [
  'EKFcUu4FLygkyZR8Ch4F8hxuJps97GCfiMRSWXDP55sgvjcmNGHc',
  'EKENfq7tEdTf5dnNxUgVo9dUnAqrEaB9syTgFyuRWinR5gPuZtbG',
  'EKEPVj2PDzQUrMwL2yeUikoQYXvh4qrkSxsDa7gegVcDvNjAteS5',
  'EKDm7SHWHEP5xiSbu52M1Z4rTFZ5Wx7YMzeaC27BQdPvvGvF42VH',
  'EKEuJJmmHNVHD1W2qmwExDyGbkSoKdKmKNPZn8QbqybVfd2Sd4hs',
  'EKEyPVU37EGw8CdGtUYnfDcBT2Eu7B6rSdy64R68UHYbrYbVJett',
];

await isReady;
let { keys, addresses } = randomAccounts(
  false,
  'tokenX',
  'tokenY',
  'dex',
  'user',
  'user2',
  'user3'
);
let tokenIds = {
  X: TokenId.derive(addresses.tokenX),
  Y: TokenId.derive(addresses.tokenY),
  lqXY: TokenId.derive(addresses.dex),
};

/**
 * Sum of balances of the account update and all its descendants
 */
function balanceSum(accountUpdate: AccountUpdate, tokenId: Field) {
  let myTokenId = accountUpdate.body.tokenId;
  let myBalance = Int64.fromObject(accountUpdate.body.balanceChange);
  let balance = Provable.if(myTokenId.equals(tokenId), myBalance, Int64.zero);
  for (let child of accountUpdate.children.accountUpdates) {
    balance = balance.add(balanceSum(child, tokenId));
  }
  return balance;
}
```

```
/**
 * Predefined accounts keys, labeled by the input strings. Useful for testing/debugging with consistent keys.
 */
function randomAccounts<K extends string>(
  createNewAccounts: boolean,
  ...names: [K, ...K[]]
): { keys: Record<K, PrivateKey>; addresses: Record<K, PublicKey> } {
  let base58Keys = createNewAccounts
    ? Array(6)
        .fill('')
        .map(() => PrivateKey.random().toBase58())
    : savedKeys;
  let keys = Object.fromEntries(
    names.map((name, idx) => [name, PrivateKey.fromBase58(base58Keys[idx])])
  ) as Record<K, PrivateKey>;
  let addresses = Object.fromEntries(
    names.map((name) => [name, keys[name].toPublicKey()])
  ) as Record<K, PublicKey>;
  return { keys, addresses };
}
```

</file>

<file>

## path: /src/examples/zkapps/dex/erc20.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/dex/erc20.ts

```
import {
  ProvablePure,
  Bool,
  CircuitString,
  provablePure,
  DeployArgs,
  Field,
  method,
  AccountUpdate,
  PublicKey,
  SmartContract,
  UInt64,
  Account,
  Experimental,
  Permissions,
  Mina,
  Int64,
  VerificationKey,
} from 'o1js';

/**
```

```
 * ERC-20 token standard.
 * https://ethereum.org/en/developers/docs/standards/tokens/erc-20/
 */
type Erc20 = {
  // pure view functions which don't need @method
  name?: () => CircuitString;
  symbol?: () => CircuitString;
  decimals?: () => Field; // TODO: should be UInt8 which doesn't exist yet
  totalSupply(): UInt64;
  balanceOf(owner: PublicKey): UInt64;
  allowance(owner: PublicKey, spender: PublicKey): UInt64;

  // mutations which need @method
  transfer(to: PublicKey, value: UInt64): Bool; // emits "Transfer" event
  transferFrom(from: PublicKey, to: PublicKey, value: UInt64): Bool; // emits "Transfer" event
  approveSpend(spender: PublicKey, value: UInt64): Bool; // emits "Approve" event

  // events
  events: {
    Transfer: ProvablePure<{
      from: PublicKey;
      to: PublicKey;
      value: UInt64;
    }>;
    Approval: ProvablePure<{
      owner: PublicKey;
      spender: PublicKey;
      value: UInt64;
    }>;
  };
};

/**
 * A simple ERC20 token
 *
 * Tokenomics:
 * The supply is constant and the entire supply is initially sent to an account controlled by the zkApp
developer
 * After that, tokens can be sent around with authorization from their owner, but new ones can't be minted.
 *
 * Functionality:
 * Just enough to be swapped by the DEX contract, and be secure
 */
class TrivialCoin extends SmartContract implements Erc20 {
  // constant supply
  SUPPLY = UInt64.from(10n ** 18n);

  deploy(args: DeployArgs) {
    super.deploy(args);
    this.account.tokenSymbol.set('SOM');
    this.account.permissions.set({
      ...Permissions.default(),
      setPermissions: Permissions.proof(),
    });
  }
  @method init() {
    super.init();

    // mint the entire supply to the token account with the same address as this contract
    let address = this.self.body.publicKey;
    let receiver = this.token.mint({
      address,
      amount: this.SUPPLY,
    });
    // assert that the receiving account is new, so this can be only done once
    receiver.account.isNew.assertEquals(Bool(true));
    // pay fees for opened account
    this.balance.subInPlace(Mina.accountCreationFee());

    // since this is the only method of this zkApp that resets the entire state, provedState: true implies
    // that this function was run. Since it can be run only once, this implies it was run exactly once

    // make account non-upgradable forever
    this.account.permissions.set({
      ...Permissions.default(),
      setVerificationKey: Permissions.impossible(),
      setPermissions: Permissions.impossible(),
      access: Permissions.proofOrSignature(),
    });
  }

  // ERC20 API
  name(): CircuitString {
    return CircuitString.fromString('SomeCoin');
  }
  symbol(): CircuitString {
    return CircuitString.fromString('SOM');
  }
  decimals(): Field {
    return Field(9);
  }
  totalSupply(): UInt64 {
    return this.SUPPLY;
  }
  balanceOf(owner: PublicKey): UInt64 {
    let account = Account(owner, this.token.id);
    let balance = account.balance.get();
    account.balance.assertEquals(balance);
    return balance;
  }
  allowance(owner: PublicKey, spender: PublicKey): UInt64 {
    // TODO: implement allowances
    return UInt64.zero;
  }
```

```
@method transfer(to: PublicKey, value: UInt64): Bool {
  this.token.send({ from: this.sender, to, amount: value });
  this.emitEvent('Transfer', { from: this.sender, to, value });
  // we don't have to check the balance of the sender -- this is done by the zkApp protocol
  return Bool(true);
}
@method transferFrom(from: PublicKey, to: PublicKey, value: UInt64): Bool {
  this.token.send({ from, to, amount: value });
  this.emitEvent('Transfer', { from, to, value });
  // we don't have to check the balance of the sender -- this is done by the zkApp protocol
  return Bool(true);
}
@method approveSpend(spender: PublicKey, value: UInt64): Bool {
  // TODO: implement allowances
  return Bool(false);
}

events = {
  Transfer: provablePure({
    from: PublicKey,
    to: PublicKey,
    value: UInt64,
  }),
  Approval: provablePure({
    owner: PublicKey,
    spender: PublicKey,
    value: UInt64,
  }),
};

// additional API needed for zkApp token accounts

@method transferFromZkapp(
  from: PublicKey,
  to: PublicKey,
  value: UInt64,
  approve: Experimental.Callback<any>
): Bool {
  // TODO: need to be able to witness a certain layout of account updates, in this case
  // tokenContract --> sender --> receiver
  let fromUpdate = this.approve(approve, AccountUpdate.Layout.NoChildren);

  let negativeAmount = Int64.fromObject(fromUpdate.body.balanceChange);
  negativeAmount.assertEquals(Int64.from(value).neg());
  let tokenId = this.token.id;
  fromUpdate.body.tokenId.assertEquals(tokenId);
  fromUpdate.body.publicKey.assertEquals(from);

  let toUpdate = AccountUpdate.create(to, tokenId);
  toUpdate.balance.addInPlace(value);
  this.emitEvent('Transfer', { from, to, value });
```

```
    return Bool(true);
  }

  // this is a very standardized deploy method. instead, we could also take the account update from a
callback
  @method deployZkapp(
    zkappAddress: PublicKey,
    verificationKey: VerificationKey
  ) {
    let tokenId = this.token.id;
    let zkapp = Experimental.createChildAccountUpdate(
      this.self,
      zkappAddress,
      tokenId
    );
    zkapp.account.permissions.set(Permissions.default());
    zkapp.account.verificationKey.set(verificationKey);
    zkapp.requireSignature();
  }

  // for letting a zkapp do whatever it wants, as long as no tokens are transfered
  // TODO: atm, we have to restrict the zkapp to have no children
  //       -> need to be able to witness a general layout of account updates
  @method approveZkapp(callback: Experimental.Callback<any>) {
    let zkappUpdate = this.approve(callback, AccountUpdate.Layout.NoChildren);
    Int64.fromObject(zkappUpdate.body.balanceChange).assertEquals(UInt64.zero);
  }
}
```

</file>

<file>

## path: /src/examples/zkapps/dex/happy-path-with-actions.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/dex/happy-path-with-actions.ts

```
import { isReady, Mina, AccountUpdate, UInt64 } from 'o1js';
import {
  Dex,
  DexTokenHolder,
  addresses,
  keys,
  tokenIds,
  getTokenBalances,
} from './dex-with-actions.js';
import { TokenContract } from './dex.js';
import { expect } from 'expect';
import { tic, toc } from '../../utils/tic-toc.node.js';

await isReady;
```

```
let proofsEnabled = true;

tic('Happy path with actions');
console.log();

let Local = Mina.LocalBlockchain({
  proofsEnabled,
  enforceTransactionLimits: true,
});
Mina.setActiveInstance(Local);
let accountFee = Mina.accountCreationFee();
let [{ privateKey: feePayerKey, publicKey: feePayerAddress }] =
  Local.testAccounts;
let tx, balances, oldBalances;

if (proofsEnabled) {
  tic('compile (token)');
  await TokenContract.compile();
  toc();
  tic('compile (dex token holder)');
  await DexTokenHolder.compile();
  toc();
  tic('compile (dex main contract)');
  await Dex.compile();
  toc();
}

let tokenX = new TokenContract(addresses.tokenX);
let tokenY = new TokenContract(addresses.tokenY);
let dex = new Dex(addresses.dex);
let dexTokenHolderX = new DexTokenHolder(addresses.dex, tokenIds.X);
let dexTokenHolderY = new DexTokenHolder(addresses.dex, tokenIds.Y);

tic('deploy & init token contracts');
tx = await Mina.transaction(feePayerAddress, () => {
  // pay fees for creating 2 token contract accounts, and fund them so each can create 1 account themselves
  let feePayerUpdate = AccountUpdate.createSigned(feePayerAddress);
  feePayerUpdate.balance.subInPlace(accountFee.mul(2));
  feePayerUpdate.send({ to: addresses.tokenX, amount: accountFee });
  feePayerUpdate.send({ to: addresses.tokenY, amount: accountFee });
  tokenX.deploy();
  tokenY.deploy();
});
await tx.prove();
await tx.sign([feePayerKey, keys.tokenX, keys.tokenY]).send();
toc();
console.log('account updates length', tx.transaction.accountUpdates.length);

tic('deploy dex contracts');
tx = await Mina.transaction(feePayerAddress, () => {
  // pay fees for creating 3 dex accounts
  AccountUpdate.createSigned(feePayerAddress).balance.subInPlace(
    accountFee.mul(3)
  );
  dex.deploy();
  dexTokenHolderX.deploy();
  tokenX.approveUpdate(dexTokenHolderX.self);
  dexTokenHolderY.deploy();
  tokenY.approveUpdate(dexTokenHolderY.self);
});
await tx.prove();
await tx.sign([feePayerKey, keys.dex]).send();
toc();
console.log('account updates length', tx.transaction.accountUpdates.length);

tic('transfer tokens to user');
let USER_DX = 1_000n;
tx = await Mina.transaction(feePayerAddress, () => {
  // pay fees for creating 3 user accounts
  let feePayer = AccountUpdate.fundNewAccount(feePayerAddress, 3);
  feePayer.send({ to: addresses.user, amount: 20e9 }); // give users MINA to pay fees
  tokenX.transfer(addresses.tokenX, addresses.user, UInt64.from(USER_DX));
  tokenY.transfer(addresses.tokenY, addresses.user, UInt64.from(USER_DX));
});
await tx.prove();
await tx.sign([feePayerKey, keys.tokenX, keys.tokenY]).send();
toc();
console.log('account updates length', tx.transaction.accountUpdates.length);

// this is done in advance to avoid account update limit in &nbsp;supply&nbsp;
tic("create user's lq token account");
tx = await Mina.transaction(addresses.user, () => {
  AccountUpdate.fundNewAccount(addresses.user);
  dex.createAccount();
});
await tx.prove();
await tx.sign([keys.user]).send();
toc();
console.log('account updates length', tx.transaction.accountUpdates.length);

[oldBalances, balances] = [balances, getTokenBalances()];
expect(balances.user.X).toEqual(USER_DX);
console.log(balances);

tic('supply liquidity');
tx = await Mina.transaction(addresses.user, () => {
  dex.supplyLiquidityBase(UInt64.from(USER_DX), UInt64.from(USER_DX));
});
await tx.prove();
await tx.sign([keys.user]).send();
toc();
console.log('account updates length', tx.transaction.accountUpdates.length);
[oldBalances, balances] = [balances, getTokenBalances()];
```

```
expect(balances.user.X).toEqual(0n);
console.log(balances);

tic('redeem liquidity, step 1');
let USER_DL = 100n;
tx = await Mina.transaction(addresses.user, () => {
  dex.redeemInitialize(UInt64.from(USER_DL));
});
await tx.prove();
await tx.sign([keys.user]).send();
toc();
console.log('account updates length', tx.transaction.accountUpdates.length);
console.log(getTokenBalances());

tic('redeem liquidity, step 2a (get back token X)');
tx = await Mina.transaction(addresses.user, () => {
  dexTokenHolderX.redeemLiquidityFinalize();
  tokenX.approveAny(dexTokenHolderX.self);
});
await tx.prove();
await tx.sign([keys.user]).send();
toc();
console.log('account updates length', tx.transaction.accountUpdates.length);
console.log(getTokenBalances());

tic('redeem liquidity, step 2b (get back token Y)');
tx = await Mina.transaction(addresses.user, () => {
  dexTokenHolderY.redeemLiquidityFinalize();
  tokenY.approveAny(dexTokenHolderY.self);
});
await tx.prove();
await tx.sign([keys.user]).send();
toc();
console.log('account updates length', tx.transaction.accountUpdates.length);
console.log(getTokenBalances());

[oldBalances, balances] = [balances, getTokenBalances()];
expect(balances.user.X).toEqual(USER_DL / 2n);

tic('swap 10 X for Y');
USER_DX = 10n;
tx = await Mina.transaction(addresses.user, () => {
  dex.swapX(UInt64.from(USER_DX));
});
await tx.prove();
await tx.sign([keys.user]).send();
toc();
console.log('account updates length', tx.transaction.accountUpdates.length);

[oldBalances, balances] = [balances, getTokenBalances()];
expect(balances.user.X).toEqual(oldBalances.user.X - USER_DX);
console.log(balances);
```

```
toc();
console.log('dex happy path with actions was successful!     ');
```

</file>

<file>

## path: /src/examples/zkapps/dex/happy-path-with-proofs.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/dex/happy-path-with-proofs.ts

```
import { isReady, Mina, AccountUpdate, UInt64 } from 'o1js';
import { createDex, TokenContract, addresses, keys, tokenIds } from './dex.js';
import { expect } from 'expect';
import { tic, toc } from '../../utils/tic-toc.node.js';
import { getProfiler } from '../../utils/profiler.js';

await isReady;

const TokenProfiler = getProfiler('Token with Proofs');
TokenProfiler.start('Token with proofs test flow');
let proofsEnabled = true;

tic('Happy path with proofs');
console.log();

let Local = Mina.LocalBlockchain({
  proofsEnabled,
  enforceTransactionLimits: false,
});
Mina.setActiveInstance(Local);
let accountFee = Mina.accountCreationFee();
let [{ privateKey: feePayerKey, publicKey: feePayerAddress }] =
  Local.testAccounts;
let tx, balances, oldBalances;

let { Dex, DexTokenHolder, getTokenBalances } = createDex();

TokenContract.analyzeMethods();
DexTokenHolder.analyzeMethods();
Dex.analyzeMethods();

if (proofsEnabled) {
  tic('compile (token)');
  await TokenContract.compile();
  toc();
  tic('compile (dex token holder)');
  await DexTokenHolder.compile();
  toc();
  tic('compile (dex main contract)');
  await Dex.compile();
  toc();
}

let tokenX = new TokenContract(addresses.tokenX);
let tokenY = new TokenContract(addresses.tokenY);
let dex = new Dex(addresses.dex);
let dexTokenHolderX = new DexTokenHolder(addresses.dex, tokenIds.X);
let dexTokenHolderY = new DexTokenHolder(addresses.dex, tokenIds.Y);

tic('deploy & init token contracts');
tx = await Mina.transaction(feePayerAddress, () => {
  // pay fees for creating 2 token contract accounts, and fund them so each can create 1 account themselves
  let feePayerUpdate = AccountUpdate.createSigned(feePayerAddress);
  feePayerUpdate.balance.subInPlace(accountFee.mul(2));
  feePayerUpdate.send({ to: addresses.tokenX, amount: accountFee });
  feePayerUpdate.send({ to: addresses.tokenY, amount: accountFee });
  tokenX.deploy();
  tokenY.deploy();
});
await tx.prove();
await tx.sign([feePayerKey, keys.tokenX, keys.tokenY]).send();
toc();
console.log('account updates length', tx.transaction.accountUpdates.length);

tic('deploy dex contracts');
tx = await Mina.transaction(feePayerAddress, () => {
  // pay fees for creating 3 dex accounts
  AccountUpdate.createSigned(feePayerAddress).balance.subInPlace(
    accountFee.mul(3)
  );
  dex.deploy();
  dexTokenHolderX.deploy();
  tokenX.approveUpdate(dexTokenHolderX.self);
  dexTokenHolderY.deploy();
  tokenY.approveUpdate(dexTokenHolderY.self);
});
await tx.prove();
await tx.sign([feePayerKey, keys.dex]).send();
toc();
console.log('account updates length', tx.transaction.accountUpdates.length);

tic('transfer tokens to user');
let USER_DX = 1_000n;
tx = await Mina.transaction(feePayerAddress, () => {
  // pay fees for creating 3 user accounts
  let feePayer = AccountUpdate.fundNewAccount(feePayerAddress, 3);
  feePayer.send({ to: addresses.user, amount: 20e9 }); // give users MINA to pay fees
  tokenX.transfer(addresses.tokenX, addresses.user, UInt64.from(USER_DX));
  tokenY.transfer(addresses.tokenY, addresses.user, UInt64.from(USER_DX));
});
await tx.prove();
await tx.sign([feePayerKey, keys.tokenX, keys.tokenY]).send();
toc();
console.log('account updates length', tx.transaction.accountUpdates.length);
[oldBalances, balances] = [balances, getTokenBalances()];
expect(balances.user.X).toEqual(USER_DX);

tic('supply liquidity');
tx = await Mina.transaction(addresses.user, () => {
  AccountUpdate.fundNewAccount(addresses.user);
  dex.supplyLiquidityBase(UInt64.from(USER_DX), UInt64.from(USER_DX));
});
await tx.prove();
await tx.sign([keys.user]).send();
toc();
console.log('account updates length', tx.transaction.accountUpdates.length);
[oldBalances, balances] = [balances, getTokenBalances()];
expect(balances.user.X).toEqual(0n);

tic('redeem liquidity');
let USER_DL = 100n;
tx = await Mina.transaction(addresses.user, () => {
  dex.redeemLiquidity(UInt64.from(USER_DL));
});
await tx.prove();
await tx.sign([keys.user]).send();
toc();
console.log('account updates length', tx.transaction.accountUpdates.length);
[oldBalances, balances] = [balances, getTokenBalances()];
expect(balances.user.X).toEqual(USER_DL / 2n);

tic('swap 10 X for Y');
USER_DX = 10n;
tx = await Mina.transaction(addresses.user, () => {
  dex.swapX(UInt64.from(USER_DX));
});
await tx.prove();
await tx.sign([keys.user]).send();
toc();
console.log('account updates length', tx.transaction.accountUpdates.length);
[oldBalances, balances] = [balances, getTokenBalances()];
expect(balances.user.X).toEqual(oldBalances.user.X - USER_DX);

toc();
console.log('dex happy path with proofs was successful!      ');
TokenProfiler.stop().store();
```

</file>

<file>

## path: /src/examples/zkapps/dex/run-berkeley.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/dex/run-berkeley.ts

```
import {
  isReady,
  Mina,
  AccountUpdate,
  UInt64,
  PrivateKey,
  fetchAccount,
} from 'o1js';
import {
  Dex,
  DexTokenHolder,
  addresses,
  keys,
  tokenIds,
} from './dex-with-actions.js';
import { TokenContract } from './dex.js';
import { expect } from 'expect';
import { tic, toc } from '../../utils/tic-toc.node.js';

await isReady;

// setting this to a higher number allows you to skip a few transactions, to pick up after an error
const successfulTransactions = 0;

tic('Run DEX with actions, happy path, on Berkeley');
console.log();

let Berkeley = Mina.Network({
  mina: 'https://berkeley.minascan.io/graphql',
  archive: 'https://archive-node-api.p42.xyz',
});
Mina.setActiveInstance(Berkeley);
let accountFee = Mina.accountCreationFee();

let tx, pendingTx: Mina.TransactionId, balances, oldBalances;

// compile contracts & wait for fee payer to be funded
let { sender, senderKey } = await ensureFundedAccount(
  'EKDrVGPC6iVRqB2bMMakNBTdEi8M1TqMn5TViLe9bafcpEExPYui'
);

TokenContract.analyzeMethods();
DexTokenHolder.analyzeMethods();
Dex.analyzeMethods();

tic('compile (token)');
await TokenContract.compile();
toc();
tic('compile (dex token holder)');
await DexTokenHolder.compile();
```

```
toc();
tic('compile (dex main contract)');
await Dex.compile();
toc();

let tokenX = new TokenContract(addresses.tokenX);
let tokenY = new TokenContract(addresses.tokenY);
let dex = new Dex(addresses.dex);
let dexTokenHolderX = new DexTokenHolder(addresses.dex, tokenIds.X);
let dexTokenHolderY = new DexTokenHolder(addresses.dex, tokenIds.Y);

let senderSpec = { sender, fee: 0.1e9 };
let userSpec = { sender: addresses.user, fee: 0.1e9 };

if (successfulTransactions <= 0) {
  tic('deploy & init token contracts');
  tx = await Mina.transaction(senderSpec, () => {
    // pay fees for creating 2 token contract accounts, and fund them so each can create 1 account
themselves
    let feePayerUpdate = AccountUpdate.createSigned(sender);
    feePayerUpdate.balance.subInPlace(accountFee.mul(2));
    feePayerUpdate.send({ to: addresses.tokenX, amount: accountFee });
    feePayerUpdate.send({ to: addresses.tokenY, amount: accountFee });
    tokenX.deploy();
    tokenY.deploy();
  });
  await tx.prove();
  pendingTx = await tx.sign([senderKey, keys.tokenX, keys.tokenY]).send();
  toc();
  console.log('account updates length', tx.transaction.accountUpdates.length);
  logPendingTransaction(pendingTx);
  tic('waiting');
  await pendingTx.wait();
  await sleep(10);
  toc();
}

if (successfulTransactions <= 1) {
  tic('deploy dex contracts');
  tx = await Mina.transaction(senderSpec, () => {
    // pay fees for creating 3 dex accounts
    AccountUpdate.createSigned(sender).balance.subInPlace(accountFee.mul(3));
    dex.deploy();
    dexTokenHolderX.deploy();
    tokenX.approveUpdate(dexTokenHolderX.self);
    dexTokenHolderY.deploy();
    tokenY.approveUpdate(dexTokenHolderY.self);
  });
  await tx.prove();
  pendingTx = await tx.sign([senderKey, keys.dex]).send();
  toc();
  console.log('account updates length', tx.transaction.accountUpdates.length);
  logPendingTransaction(pendingTx);
  tic('waiting');
  await pendingTx.wait();
  await sleep(10);
  toc();
}

let USER_DX = 1_000n;

if (successfulTransactions <= 2) {
  tic('transfer tokens to user');
  tx = await Mina.transaction(senderSpec, () => {
    // pay fees for creating 3 user accounts
    let feePayer = AccountUpdate.fundNewAccount(sender, 3);
    feePayer.send({ to: addresses.user, amount: 8e9 }); // give users MINA to pay fees
    tokenX.transfer(addresses.tokenX, addresses.user, UInt64.from(USER_DX));
    tokenY.transfer(addresses.tokenY, addresses.user, UInt64.from(USER_DX));
  });
  await tx.prove();
  pendingTx = await tx.sign([senderKey, keys.tokenX, keys.tokenY]).send();
  toc();
  console.log('account updates length', tx.transaction.accountUpdates.length);
  logPendingTransaction(pendingTx);
  tic('waiting');
  await pendingTx.wait();
  await sleep(10);
  toc();
}

if (successfulTransactions <= 3) {
  // this is done in advance to avoid account update limit in &nbsp;supply&nbsp;
  tic("create user's lq token account");
  tx = await Mina.transaction(userSpec, () => {
    AccountUpdate.fundNewAccount(addresses.user);
    dex.createAccount();
  });
  await tx.prove();
  pendingTx = await tx.sign([keys.user]).send();
  toc();
  console.log('account updates length', tx.transaction.accountUpdates.length);
  logPendingTransaction(pendingTx);
  tic('waiting');
  await pendingTx.wait();
  await sleep(10);
  toc();

  [oldBalances, balances] = [balances, await getTokenBalances()];
  expect(balances.user.X).toEqual(USER_DX);
  console.log(balances);
}

if (successfulTransactions <= 4) {
  tic('supply liquidity');
  tx = await Mina.transaction(userSpec, () => {
    dex.supplyLiquidityBase(UInt64.from(USER_DX), UInt64.from(USER_DX));
  });
  await tx.prove();
  pendingTx = await tx.sign([keys.user]).send();
  toc();
  console.log('account updates length', tx.transaction.accountUpdates.length);
  logPendingTransaction(pendingTx);
  tic('waiting');
  await pendingTx.wait();
  await sleep(10);
  toc();

  [oldBalances, balances] = [balances, await getTokenBalances()];
  expect(balances.user.X).toEqual(0n);
  console.log(balances);
}

let USER_DL = 100n;

if (successfulTransactions <= 5) {
  tic('redeem liquidity, step 1');
  tx = await Mina.transaction(userSpec, () => {
    dex.redeemInitialize(UInt64.from(USER_DL));
  });
  await tx.prove();
  pendingTx = await tx.sign([keys.user]).send();
  toc();
  console.log('account updates length', tx.transaction.accountUpdates.length);
  logPendingTransaction(pendingTx);
  tic('waiting');
  await pendingTx.wait();
  await sleep(10);
  toc();

  console.log(await getTokenBalances());
}

if (successfulTransactions <= 6) {
  tic('redeem liquidity, step 2a (get back token X)');
  tx = await Mina.transaction(userSpec, () => {
    dexTokenHolderX.redeemLiquidityFinalize();
    tokenX.approveAny(dexTokenHolderX.self);
  });
  await tx.prove();
  pendingTx = await tx.sign([keys.user]).send();
  toc();
  console.log('account updates length', tx.transaction.accountUpdates.length);
  logPendingTransaction(pendingTx);
  tic('waiting');
  await pendingTx.wait();
  await sleep(10);
  toc();

  console.log(await getTokenBalances());
}

if (successfulTransactions <= 7) {
  tic('redeem liquidity, step 2b (get back token Y)');
  tx = await Mina.transaction(userSpec, () => {
    dexTokenHolderY.redeemLiquidityFinalize();
    tokenY.approveAny(dexTokenHolderY.self);
  });
  await tx.prove();
  pendingTx = await tx.sign([keys.user]).send();
  toc();
  console.log('account updates length', tx.transaction.accountUpdates.length);
  logPendingTransaction(pendingTx);
  tic('waiting');
  await pendingTx.wait();
  await sleep(10);
  toc();

  [oldBalances, balances] = [balances, await getTokenBalances()];
  expect(balances.user.X).toEqual(USER_DL / 2n);
  console.log(balances);
}

if (successfulTransactions <= 8) {
  oldBalances = await getTokenBalances();

  tic('swap 10 X for Y');
  USER_DX = 10n;
  tx = await Mina.transaction(userSpec, () => {
    dex.swapX(UInt64.from(USER_DX));
  });
  await tx.prove();
  pendingTx = await tx.sign([keys.user]).send();
  toc();
  console.log('account updates length', tx.transaction.accountUpdates.length);
  logPendingTransaction(pendingTx);
  tic('waiting');
  await pendingTx.wait();
  await sleep(10);
  toc();

  balances = await getTokenBalances();
  expect(balances.user.X).toEqual(oldBalances.user.X - USER_DX);
  console.log(balances);
}

toc();
console.log('dex happy path with actions was successful!     ');
```

```
async function ensureFundedAccount(privateKeyBase58: string) {
  let senderKey = PrivateKey.fromBase58(privateKeyBase58);
  let sender = senderKey.toPublicKey();
  let result = await fetchAccount({ publicKey: sender });
  let balance = result.account?.balance.toBigInt();
  if (balance === undefined || balance <= 15_000_000_000n) {
    await Mina.faucet(sender);
    await sleep(1);
  }
  return { senderKey, sender };
}

function logPendingTransaction(pendingTx: Mina.TransactionId) {
  if (!pendingTx.isSuccess) throw Error('transaction failed');
  console.log(
    &nbsp;tx sent: https://berkeley.minaexplorer.com/transaction/${pendingTx.hash()}&nbsp;
  );
}

async function getTokenBalances() {
  // fetch accounts
  await Promise.all(
    [
      { publicKey: addresses.user },
      { publicKey: addresses.user, tokenId: tokenIds.X },
      { publicKey: addresses.user, tokenId: tokenIds.Y },
      { publicKey: addresses.user, tokenId: tokenIds.lqXY },
      { publicKey: addresses.dex },
      { publicKey: addresses.dex, tokenId: tokenIds.X },
      { publicKey: addresses.dex, tokenId: tokenIds.Y },
    ].map((a) => fetchAccount(a))
  );

  let balances = {
    user: { MINA: 0n, X: 0n, Y: 0n, lqXY: 0n },
    dex: { X: 0n, Y: 0n, lqXYSupply: 0n },
  };
  let user = 'user' as const;
  try {
    balances.user.MINA =
      Mina.getBalance(addresses[user]).toBigInt() / 1_000_000_000n;
  } catch {}
  for (let token of ['X', 'Y', 'lqXY'] as const) {
    try {
      balances[user][token] = Mina.getBalance(
        addresses[user],
        tokenIds[token]
      ).toBigInt();
    } catch {}
  }
  try {
    balances.dex.X = Mina.getBalance(addresses.dex, tokenIds.X).toBigInt();
  } catch {}
  try {
    balances.dex.Y = Mina.getBalance(addresses.dex, tokenIds.Y).toBigInt();
  } catch {}
  try {
    let dex = new Dex(addresses.dex);
    balances.dex.lqXYSupply = dex.totalSupply.get().toBigInt();
  } catch {}
  return balances;
}

async function sleep(sec: number) {
  return new Promise((r) => setTimeout(r, sec * 1000));
}
```

</file>

<file>

## path: /src/examples/zkapps/dex/run.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/dex/run.ts

```
import {
  isReady,
  Mina,
  AccountUpdate,
  UInt64,
  shutdown,
  Permissions,
  TokenId,
} from 'o1js';
import { createDex, TokenContract, addresses, keys, tokenIds } from './dex.js';
import { expect } from 'expect';

import { getProfiler } from '../../utils/profiler.js';

await isReady;
let proofsEnabled = false;

let Local = Mina.LocalBlockchain({
  proofsEnabled,
  enforceTransactionLimits: false,
});
Mina.setActiveInstance(Local);
let accountFee = Mina.accountCreationFee();
let [{ privateKey: feePayerKey, publicKey: feePayerAddress }] =
  Local.testAccounts;
let tx, balances, oldBalances;
```

```
console.log('-------------------------------------------------');
console.log('FEE PAYER\t', feePayerAddress.toBase58());
console.log('TOKEN X ADDRESS\t', addresses.tokenX.toBase58());
console.log('TOKEN Y ADDRESS\t', addresses.tokenY.toBase58());
console.log('DEX ADDRESS\t', addresses.dex.toBase58());
console.log('USER ADDRESS\t', addresses.user.toBase58());
console.log('-------------------------------------------------');
console.log('TOKEN X ID\t', TokenId.toBase58(tokenIds.X));
console.log('TOKEN Y ID\t', TokenId.toBase58(tokenIds.Y));
console.log('-------------------------------------------------');

TokenContract.analyzeMethods();
if (proofsEnabled) {
  console.log('compile (token)...');
  await TokenContract.compile();
}

await main({ withVesting: false });

// swap out ledger so we can start fresh
Local = Mina.LocalBlockchain({
  proofsEnabled,
  enforceTransactionLimits: false,
});
Mina.setActiveInstance(Local);
[{ privateKey: feePayerKey }] = Local.testAccounts;
feePayerAddress = feePayerKey.toPublicKey();

await main({ withVesting: true });

console.log('all dex tests were successful!      ');

async function main({ withVesting }: { withVesting: boolean }) {
  const DexProfiler = getProfiler(
    &nbsp;DEX testing ${withVesting ? 'with vesting' : ''}&nbsp;
  );
  DexProfiler.start('DEX test flow');
  if (withVesting) console.log('\nWITH VESTING');
  else console.log('\nNO VESTING');

  let options = withVesting ? { lockedLiquiditySlots: 2 } : undefined;
  let { Dex, DexTokenHolder, getTokenBalances } = createDex(options);

  // analyze methods for quick error feedback
  DexTokenHolder.analyzeMethods();
  Dex.analyzeMethods();

  if (proofsEnabled) {
    // compile & deploy all zkApps
    console.log('compile (dex token holder)...');
    await DexTokenHolder.compile();
    console.log('compile (dex main contract)...');
    await Dex.compile();
  }

  let tokenX = new TokenContract(addresses.tokenX);
  let tokenY = new TokenContract(addresses.tokenY);
  let dex = new Dex(addresses.dex);
  let dexTokenHolderX = new DexTokenHolder(addresses.dex, tokenIds.X);
  let dexTokenHolderY = new DexTokenHolder(addresses.dex, tokenIds.Y);

  console.log('deploy & init token contracts...');
  tx = await Mina.transaction(feePayerAddress, () => {
    // pay fees for creating 2 token contract accounts, and fund them so each can create 2 accounts
themselves
    let feePayerUpdate = AccountUpdate.fundNewAccount(feePayerAddress, 2);
    feePayerUpdate.send({ to: addresses.tokenX, amount: accountFee.mul(2) });
    feePayerUpdate.send({ to: addresses.tokenY, amount: accountFee.mul(2) });
    tokenX.deploy();
    tokenY.deploy();
  });
  await tx.prove();
  tx.sign([feePayerKey, keys.tokenX, keys.tokenY]);
  await tx.send();
  balances = getTokenBalances();
  console.log(
    'Token contract tokens (X, Y):',
    balances.tokenContract.X,
    balances.tokenContract.Y
  );

  console.log('deploy dex contracts...');
  tx = await Mina.transaction(feePayerAddress, () => {
    // pay fees for creating 3 dex accounts
    AccountUpdate.fundNewAccount(feePayerAddress, 3);
    dex.deploy();
    dexTokenHolderX.deploy();
    tokenX.approveUpdate(dexTokenHolderX.self);
    dexTokenHolderY.deploy();
    tokenY.approveUpdate(dexTokenHolderY.self);
  });
  await tx.prove();
  tx.sign([feePayerKey, keys.dex]);
  await tx.send();

  console.log('transfer tokens to user');
  tx = await Mina.transaction(
    { sender: feePayerAddress, fee: accountFee.mul(1) },
    () => {
      let feePayer = AccountUpdate.fundNewAccount(feePayerAddress, 4);
      feePayer.send({ to: addresses.user, amount: 20e9 }); // give users MINA to pay fees
      feePayer.send({ to: addresses.user2, amount: 20e9 });
      // transfer to fee payer so they can provide initial liquidity
      tokenX.transfer(addresses.tokenX, feePayerAddress, UInt64.from(10_000));
      tokenY.transfer(addresses.tokenY, feePayerAddress, UInt64.from(10_000));
      // mint tokens to the user (this is additional to the tokens minted at the beginning, so we can overflow the
balance
      tokenX.init2();
      tokenY.init2();
    }
  );
  await tx.prove();
  tx.sign([feePayerKey, keys.tokenX, keys.tokenY]);
  await tx.send();
  [oldBalances, balances] = [balances, getTokenBalances()];
  console.log('User tokens (X, Y):', balances.user.X, balances.user.Y);
  console.log('User MINA:', balances.user.MINA);

  // supply the initial liquidity where the token ratio can be arbitrary
  console.log('supply liquidity -- base');
  tx = await Mina.transaction(
    { sender: feePayerAddress, fee: accountFee },
    () => {
      AccountUpdate.fundNewAccount(feePayerAddress);
      dex.supplyLiquidityBase(UInt64.from(10_000), UInt64.from(10_000));
    }
  );
  await tx.prove();
  tx.sign([feePayerKey]);
  await tx.send();
  [oldBalances, balances] = [balances, getTokenBalances()];
  console.log('DEX liquidity (X, Y):', balances.dex.X, balances.dex.Y);

  /**
   * SUPPLY LIQUIDITY
   *
   * Happy path (lqXY token was not created for user’s account before)
   *
   * Test Preconditions:
   * - Tokens X and Y created;
   * - Some amount of both tokens minted (balances > 0) and available for user's token account;
   * - Initial liquidity provided to the DEX contract, so that there exists a token X : Y ratio
   *   from which to calculate required liquidity inputs
   */
  expect(balances.tokenContract.X).toBeGreaterThan(0n);
  expect(balances.tokenContract.Y).toBeGreaterThan(0n);
  expect(balances.user.X).toBeGreaterThan(0n);
  expect(balances.user.Y).toBeGreaterThan(0n);
  expect(balances.total.lqXY).toBeGreaterThan(0n);

  /**
   * Actions:
   * - User calls the “Supply Liquidity” smart contract method providing the required tokens
   *   account information (if not derived automatically) and tokens amounts one is willing to supply.
   * - User provides the account creation fee, to be subtracted from its Mina account
   *
```

```
  * note: we supply much more liquidity here, so we can exercise the overflow failure case after that
  */
 let USER_DX = 500_000n;
 console.log('user supply liquidity (1)');
 tx = await Mina.transaction(addresses.user, () => {
   AccountUpdate.fundNewAccount(addresses.user);
   dex.supplyLiquidity(UInt64.from(USER_DX));
 });
 await tx.prove();
 tx.sign([keys.user]);
 await tx.send();
 [oldBalances, balances] = [balances, getTokenBalances()];
 console.log('DEX liquidity (X, Y):', balances.dex.X, balances.dex.Y);
 console.log('user DEX tokens:', balances.user.lqXY);
 console.log('user MINA:', balances.user.MINA);

 /**
  * Expected results:
  * - Smart contract transfers specified amount of tokens from user's account to SC account.
  *   - Check the balances.
  * - SC mints the "lqXY" tokens in the amount calculated based on the current liquidity pool state
  *   and AMM formula application, consumes the lqXY token creation fee from the user's default
  *   token account (in parent tokens, which is Mina) and transfers amount of minted lqXY tokens to user's
account;
  */
 expect(balances.user.X).toEqual(oldBalances.user.X - USER_DX);
 expect(balances.user.Y).toEqual(
   oldBalances.user.Y - (USER_DX * oldBalances.dex.Y) / oldBalances.dex.X
 );
 expect(balances.user.MINA).toEqual(oldBalances.user.MINA - 1n);
 expect(balances.user.lqXY).toEqual(
   (USER_DX * oldBalances.total.lqXY) / oldBalances.dex.X
 );

 /**
  * Happy path (lqXY token exists for users account)
  *
  * Same case but we are checking that no token creation fee is paid by the liquidity supplier.
  *
  * Note: with vesting, this is a failure case because we can't change timing on an account that currently has
an active timing
  */
 USER_DX = 1000n;
 console.log('user supply liquidity (2)');
 tx = await Mina.transaction(addresses.user, () => {
   dex.supplyLiquidity(UInt64.from(USER_DX));
 });
 await tx.prove();
 tx.sign([keys.user]);
 if (!withVesting) {
   await tx.send();
   [oldBalances, balances] = [balances, getTokenBalances()];
   console.log('DEX liquidity (X, Y):', balances.dex.X, balances.dex.Y);
   console.log('user DEX tokens:', balances.user.lqXY);
   expect(balances.user.X).toEqual(oldBalances.user.X - USER_DX);
   expect(balances.user.Y).toEqual(
     oldBalances.user.Y - (USER_DX * oldBalances.dex.Y) / oldBalances.dex.X
   );
   expect(balances.user.MINA).toEqual(oldBalances.user.MINA);
   expect(balances.user.lqXY).toEqual(
     oldBalances.user.lqXY +
       (USER_DX * oldBalances.total.lqXY) / oldBalances.dex.X
   );
 } else {
   await expect(tx.send()).rejects.toThrow(/Update_not_permitted_timing/);
 }

 /**
  * Check the method failures during an attempts to supply liquidity when:
  * - There is no token X or Y (or both) created yet for user's account;
  * - There is not enough tokens available for user's tokens accounts, one is willing to supply;
  */
 console.log('supplying with no tokens (should fail)');
 tx = await Mina.transaction(addresses.user2, () => {
   AccountUpdate.fundNewAccount(addresses.user2);
   dex.supplyLiquidityBase(UInt64.from(100), UInt64.from(100));
 });
 await tx.prove();
 tx.sign([keys.user2]);
 await expect(tx.send()).rejects.toThrow(/Overflow/);
 console.log('supplying with insufficient tokens (should fail)');
 tx = await Mina.transaction(addresses.user, () => {
   dex.supplyLiquidityBase(UInt64.from(1e9), UInt64.from(1e9));
 });
 await tx.prove();
 tx.sign([keys.user]);
 await expect(tx.send()).rejects.toThrow(/Overflow/);

 /**
  * - Resulting operation will overflow the SC's receiving token by type or by any other applicable limits;
  *
  * note: this throws not at the protocol level, but because the smart contract multiplies two UInt64s which
overflow.
  * this happens in all DEX contract methods!
  * => a targeted test with explicitly constructed account updates might be the better strategy to test overflow
  */
 console.log('prepare supplying overflowing liquidity');
 tx = await Mina.transaction(feePayerAddress, () => {
   AccountUpdate.fundNewAccount(feePayerAddress);
   tokenY.transfer(
     addresses.tokenY,
     addresses.tokenX,
     UInt64.MAXINT().sub(200_000)
   );
 });
 await tx.prove();
 await tx.sign([feePayerKey, keys.tokenY]).send();
 console.log('supply overflowing liquidity');
 await expect(async () => {
   tx = await Mina.transaction(addresses.tokenX, () => {
     dex.supplyLiquidityBase(
       UInt64.MAXINT().sub(200_000),
       UInt64.MAXINT().sub(200_000)
     );
   });
   await tx.prove();
   tx.sign([keys.tokenX]);
   await tx.send();
 }).rejects.toThrow();

 /**
  * - Value transfer is restricted (supplier end: withdrawal is prohibited, receiver end: receiving is prohibited)
for one or both accounts.
  */
 console.log('prepare test with forbidden send');
 tx = await Mina.transaction(addresses.tokenX, () => {
   let tokenXtokenAccount = AccountUpdate.create(addresses.tokenX, tokenIds.X);
   tokenXtokenAccount.account.permissions.set({
     ...Permissions.initial(),
     send: Permissions.impossible(),
   });
   tokenXtokenAccount.requireSignature();
   // token X owner approves w/ signature so we don't need another method for this test
   let tokenX = AccountUpdate.create(addresses.tokenX);
   tokenX.approve(tokenXtokenAccount);
   tokenX.requireSignature();
 });
 await tx.prove();
 await tx.sign([keys.tokenX]).send();
 console.log('supply with forbidden withdrawal (should fail)');
 tx = await Mina.transaction(addresses.tokenX, () => {
   AccountUpdate.fundNewAccount(addresses.tokenX);
   dex.supplyLiquidity(UInt64.from(10));
 });
 await tx.prove();
 await expect(tx.sign([keys.tokenX]).send()).rejects.toThrow(
   /Update_not_permitted_balance/
 );

 [oldBalances, balances] = [balances, getTokenBalances()];

 /**
  * REDEEM LIQUIDITY
  */
 if (withVesting) {
   /**
```

```
   * Happy path (vesting period applied)
   * - Same case but this time the “Supply Liquidity” happy path case was processed with vesting period
   *   applied for lqXY tokens. We’re checking that it is impossible to redeem lqXY tokens without respecting
   *   the timing first and then we check that tokens can be redeemed once timing conditions are met.
   */

  // liquidity is locked for 2 slots
  // step forward 1 slot => liquidity not unlocked yet
  Local.incrementGlobalSlot(1);
  let USER_DL = 100n;
  console.log('user redeem liquidity (before liquidity token unlocks)');
  tx = await Mina.transaction(addresses.user, () => {
    dex.redeemLiquidity(UInt64.from(USER_DL));
  });
  await tx.prove();
  tx.sign([keys.user]);
  await expect(tx.send()).rejects.toThrow(/Source_minimum_balance_violation/);

  // another slot => now it should work
  Local.incrementGlobalSlot(1);
}
/**
 * Happy path (no vesting applied)
 *
 * Test Preconditions:
 * - The "Supply Liquidity" happy path case was processed with no vesting period for lqXY tokens applied.
 * - User has some lqXY tokens
 *
 * Actions:
 * - User calls the “Liquidity Redemption” SC method providing the amount of lqXY tokens one is willing to
redeem.
 *
 * Note: we reuse this logic for successful redemption in the vesting case
 */
let USER_DL = 100n;
console.log('user redeem liquidity');
tx = await Mina.transaction(addresses.user, () => {
  dex.redeemLiquidity(UInt64.from(USER_DL));
});
await tx.prove();
tx.sign([keys.user]);
await tx.send();
[oldBalances, balances] = [balances, getTokenBalances()];
console.log('DEX liquidity (X, Y):', balances.dex.X, balances.dex.Y);
console.log('user DEX tokens:', balances.user.lqXY);
console.log('User tokens (X, Y):', balances.user.X, balances.user.Y);

/**
 * Expected results:
 * - Asked amount of lqXY tokens is burned off the user's account;
 *   - Check the balance before and after.
 * - The pool's liquidity will reflect the changes of lqXY tokens supply upon the next "Swap" operation;
 *   - We probably don't need to check it here?
 * - Calculated amount of X and Y tokens are transferred to user's tokens accounts;
 *   - Check balances on sender and receiver sides.
 */
expect(balances.user.lqXY).toEqual(oldBalances.user.lqXY - USER_DL);
expect(balances.total.lqXY).toEqual(oldBalances.total.lqXY - USER_DL);
let [dx, dy] = [
  (USER_DL * oldBalances.dex.X) / oldBalances.total.lqXY,
  (USER_DL * oldBalances.dex.Y) / oldBalances.total.lqXY,
];
expect(balances.user.X).toEqual(oldBalances.user.X + dx);
expect(balances.user.Y).toEqual(oldBalances.user.Y + dy);
expect(balances.dex.X).toEqual(oldBalances.dex.X - dx);
expect(balances.dex.Y).toEqual(oldBalances.dex.Y - dy);

/**
 * Bonus test (supply liquidity): check that now that the lock period is over, we can supply liquidity again
 */
if (withVesting) {
  USER_DX = 1000n;
  console.log('user supply liquidity -- again, after lock period ended');
  tx = await Mina.transaction(addresses.user, () => {
    dex.supplyLiquidity(UInt64.from(USER_DX));
  });
  await tx.prove();
  await tx.sign([keys.user]).send();
  [oldBalances, balances] = [balances, getTokenBalances()];
  console.log('User tokens (X, Y):', balances.user.X, balances.user.Y);
  expect(balances.user.X).toEqual(oldBalances.user.X - USER_DX);
  expect(balances.user.Y).toEqual(
    oldBalances.user.Y - (USER_DX * oldBalances.dex.Y) / oldBalances.dex.X
  );
}
// tests below are not specific to vesting
if (withVesting) {
  DexProfiler.stop().store();
  return;
}

/**
 * Happy path (tokens creation on receiver side in case of their absence)
 * - Same case but we are checking that one of the tokens will be created for the user
 *   (including fee payment for token creation) in case when it doesn't exist yet.
 *
 * Check the method failures during an attempts to redeem lqXY tokens when:
 * - Emulate conflicting balance preconditions due to concurrent user interactions
 *   by packing multiple redemptions into one transaction
 *
 * note: we transfer some lqXY tokens from &nbsp;user&nbsp; to &nbsp;user2&nbsp;, then we try to
redeem the with both users
 * -- which exercises a failure case -- and then redeem them all with &nbsp;user2&nbsp; (creating their
token accounts)
```

```
 */
USER_DL = 80n;
console.log('transfer liquidity tokens to user2');
tx = await Mina.transaction(addresses.user, () => {
  AccountUpdate.fundNewAccount(addresses.user);
  dex.transfer(addresses.user, addresses.user2, UInt64.from(USER_DL));
});
await tx.prove();
await tx.sign([keys.user]).send();
[oldBalances, balances] = [balances, getTokenBalances()];

console.log(
  'redeem liquidity with both users in one tx (fails because of conflicting balance preconditions)'
);
tx = await Mina.transaction(addresses.user2, () => {
  AccountUpdate.createSigned(addresses.user2).balance.subInPlace(
    accountFee.mul(2)
  );
  dex.redeemLiquidity(UInt64.from(USER_DL));
  dex.redeemLiquidity(UInt64.from(USER_DL));
});
await tx.prove();
tx.sign([keys.user, keys.user2]);
await expect(tx.send()).rejects.toThrow(
  /Account_balance_precondition_unsatisfied/
);

console.log('user2 redeem liquidity');
tx = await Mina.transaction(addresses.user2, () => {
  AccountUpdate.createSigned(addresses.user2).balance.subInPlace(
    accountFee.mul(2)
  );
  dex.redeemLiquidity(UInt64.from(USER_DL));
});
await tx.prove();
await tx.sign([keys.user2]).send();
[oldBalances, balances] = [balances, getTokenBalances()];

expect(balances.user2.lqXY).toEqual(oldBalances.user2.lqXY - USER_DL);
[dx, dy] = [
  (USER_DL * oldBalances.dex.X) / oldBalances.total.lqXY,
  (USER_DL * oldBalances.dex.Y) / oldBalances.total.lqXY,
];
expect(balances.user2.X).toEqual(oldBalances.user2.X + dx);
expect(balances.user2.Y).toEqual(oldBalances.user2.Y + dy);
expect(balances.user2.MINA).toEqual(oldBalances.user2.MINA - 2n);

/**
 * Check the method failures during an attempts to redeem lqXY tokens when:
 * - There is not enough lqXY tokens available for user’s account;
 *
 * note: user2's account is empty now, so redeeming more liquidity fails
 */
console.log('user2 redeem liquidity (fails because insufficient balance)');
tx = await Mina.transaction(addresses.user2, () => {
  dex.redeemLiquidity(UInt64.from(1n));
});
await tx.prove();
await expect(tx.sign([keys.user2]).send()).rejects.toThrow(/Overflow/);
[oldBalances, balances] = [balances, getTokenBalances()];

/**
 * SWAP
 *
 * Happy path (both tokens (X and Y) were created for user)
 *
 * Test Preconditions:
 * - User has token accounts;
 * - Balance of token X is > 0
 * - Liquidity Pool is capable of covering the Swap operation.
 *
 * Actions:
 * - User calls the “Swap” SC method providing the token (X for example) and amount it wants to swap.
 */
USER_DX = 10n;
console.log('swap 10 X for Y');
tx = await Mina.transaction(addresses.user, () => {
  dex.swapX(UInt64.from(USER_DX));
});
await tx.prove();
await tx.sign([keys.user]).send();
[oldBalances, balances] = [balances, getTokenBalances()];
console.log('User tokens (X, Y):', balances.user.X, balances.user.Y);
/**
 * Expected results:
  * - SC calculates (using AMM formula and current pool state) the resulting amount of Y token user should
receive as the result of the Swap operation;
  * - SC withdraws requested amount of X token from user’s account;
  * - SC sends to user previously calculated amount of Y tokens;
  * - It will be good to check if calculation was done correctly but correctness is not a major concern since
we’re checking
  *   the zkApps/o1js on/off-chain features, not the current application's logic;
  *   We're checking the balances of both tokens on caller and SC sides.
 */
dy = (USER_DX * oldBalances.dex.Y) / (oldBalances.dex.X + USER_DX);
expect(balances.user.X).toEqual(oldBalances.user.X - USER_DX);
expect(balances.user.Y).toEqual(oldBalances.user.Y + dy);
expect(balances.dex.X).toEqual(oldBalances.dex.X + USER_DX);
expect(balances.dex.Y).toEqual(oldBalances.dex.Y - dy);
// x*y is increasing (the dex doesn't lose money from rounding errors -- the user does)
expect(balances.dex.X * balances.dex.Y).toBeGreaterThanOrEqual(
  oldBalances.dex.X * oldBalances.dex.Y
);
```

```
  DexProfiler.stop().store();
}

shutdown();
```

</file>

<file>

## path: /src/examples/zkapps/dex/upgradability.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/dex/upgradability.ts

```
import {
  AccountUpdate,
  Mina,
  isReady,
  Permissions,
  PrivateKey,
  UInt64,
} from 'o1js';
import { createDex, TokenContract, addresses, keys, tokenIds } from './dex.js';
import { expect } from 'expect';
import { getProfiler } from '../../utils/profiler.js';

await isReady;

let proofsEnabled = false;

console.log('starting upgradeability tests');

await upgradeabilityTests({
  withVesting: false,
});
console.log('all upgradeability tests were successful!      ');

console.log('starting atomic actions tests');

await atomicActionsTest({
  withVesting: false,
});

console.log('all atomic actions tests were successful!');

async function atomicActionsTest({ withVesting }: { withVesting: boolean }) {
  const DexProfiler = getProfiler('DEX profiler atomic actions');
  DexProfiler.start('DEX test flow');
  let Local = Mina.LocalBlockchain({
    proofsEnabled,
    enforceTransactionLimits: false,
  });
```

```
Mina.setActiveInstance(Local);
let accountFee = Mina.accountCreationFee();
let [{ privateKey: feePayerKey, publicKey: feePayerAddress }] =
  Local.testAccounts;
let tx, balances;

let options = withVesting ? { lockedLiquiditySlots: 2 } : undefined;
let { Dex, DexTokenHolder, getTokenBalances } = createDex(options);

// analyze methods for quick error feedback
DexTokenHolder.analyzeMethods();
Dex.analyzeMethods();

if (proofsEnabled) {
  // compile & deploy all zkApps
  console.log('compile (dex token holder)...');
  await DexTokenHolder.compile();
  console.log('compile (dex main contract)...');
  await Dex.compile();
}

let tokenX = new TokenContract(addresses.tokenX);
let tokenY = new TokenContract(addresses.tokenY);
let dex = new Dex(addresses.dex);
let dexTokenHolderX = new DexTokenHolder(addresses.dex, tokenIds.X);
let dexTokenHolderY = new DexTokenHolder(addresses.dex, tokenIds.Y);

console.log('deploy & init token contracts...');
tx = await Mina.transaction(feePayerAddress, () => {
  // pay fees for creating 2 token contract accounts, and fund them so each can create 2 accounts
themselves
  let feePayerUpdate = AccountUpdate.fundNewAccount(feePayerAddress, 2);
  feePayerUpdate.send({ to: addresses.tokenX, amount: accountFee.mul(2) });
  feePayerUpdate.send({ to: addresses.tokenY, amount: accountFee.mul(2) });
  tokenX.deploy();
  tokenY.deploy();
});
await tx.prove();
tx.sign([feePayerKey, keys.tokenX, keys.tokenY]);
await tx.send();
balances = getTokenBalances();
console.log(
  'Token contract tokens (X, Y):',
  balances.tokenContract.X,
  balances.tokenContract.Y
);

/**
 * # Atomic Actions 1
 *
 * Preconditions:
 *  - SC deployed with a non-writable delegate field (impossible permission)
 *
 * Actions:
 *  - User sets permissions for delegate to be edible
 *  - User changes delegate field to something
 *
 * Expected:
 *  - delegate now holds a new address
 */

console.log('deploy dex contracts...');
tx = await Mina.transaction(feePayerAddress, () => {
  // pay fees for creating 3 dex accounts
  AccountUpdate.fundNewAccount(feePayerAddress, 3);
  dex.deploy();
  dexTokenHolderX.deploy();
  tokenX.approveUpdate(dexTokenHolderX.self);
  dexTokenHolderY.deploy();
  tokenY.approveUpdate(dexTokenHolderY.self);
  console.log('manipulating setDelegate field to impossible...');
  // setting the setDelegate permission field to impossible
  let dexAccount = AccountUpdate.create(addresses.dex);
  dexAccount.account.permissions.set({
    ...Permissions.initial(),
    setDelegate: Permissions.impossible(),
  });
  dexAccount.requireSignature();
});
await tx.prove();
tx.sign([feePayerKey, keys.dex]);
await tx.send();

console.log(
  'trying to change delegate (setDelegate=impossible, should fail)'
);
tx = await Mina.transaction(feePayerAddress, () => {
  // setting the delegate field to something, although permissions forbid it
  let dexAccount = AccountUpdate.create(addresses.dex);
  dexAccount.account.delegate.set(PrivateKey.random().toPublicKey());
  dexAccount.requireSignature();
});
await tx.prove();

await expect(tx.sign([feePayerKey, keys.dex]).send()).rejects.toThrow(
  /Cannot update field 'delegate'/
);

console.log('changing delegate permission back to normal');

tx = await Mina.transaction(feePayerAddress, () => {
  let dexAccount = AccountUpdate.create(addresses.dex);
  dexAccount.account.permissions.set({
    ...Permissions.initial(),
    setDelegate: Permissions.proofOrSignature(),
  });
  dexAccount.requireSignature();
});
await tx.prove();
await tx.sign([feePayerKey, keys.dex]).send();

console.log('changing delegate field to a new address');

let newDelegate = PrivateKey.random().toPublicKey();
tx = await Mina.transaction(feePayerAddress, () => {
  let dexAccount = AccountUpdate.create(addresses.dex);
  dexAccount.account.delegate.set(newDelegate);
  dexAccount.requireSignature();
});
await tx.prove();
await tx.sign([feePayerKey, keys.dex]).send();

Mina.getAccount(addresses.dex).delegate?.assertEquals(newDelegate);

/**
 * # Atomic Actions 2
 *
 * Preconditions:
 *  - Similar to happy path (SC deployed), but changing order of transaction and updates (within one
transaction)
 *
 * Actions:
 *  - Include multiple actions of different order in the same transaction
 *
 * Expected:
 *  - tx fails if the order of actions is not valid
 */

console.log(
  'changing permission to impossible and then trying to change delegate field - in one transaction'
);

tx = await Mina.transaction(feePayerAddress, () => {
  // changing the permission to impossible and then trying to change the delegate field

  let permissionUpdate = AccountUpdate.create(addresses.dex);
  permissionUpdate.account.permissions.set({
    ...Permissions.initial(),
    setDelegate: Permissions.impossible(),
  });
  permissionUpdate.requireSignature();

  let fieldUpdate = AccountUpdate.create(addresses.dex);
  fieldUpdate.account.delegate.set(PrivateKey.random().toPublicKey());
  fieldUpdate.requireSignature();
});
```

```
  await tx.prove();
  await expect(tx.sign([feePayerKey, keys.dex]).send()).rejects.toThrow(
    /Update_not_permitted_delegate/
  );

  /**
   * # Atomic Actions 3
   *
   * Preconditions:
   *  - Similar to happy path (SC deployed) and the test before, but editing fields and changing permissions
will be successful
   *
   * Actions:
   *  - Include multiple actions in one transaction
   *    edit field -> change permission back -> successful state transition
   *
   * Expected:
   *  - tx is successful and state updated
   */

  console.log('creating multiple valid account updates in one transaction');

  newDelegate = PrivateKey.random().toPublicKey();
  tx = await Mina.transaction(feePayerAddress, () => {
    // changing field
    let fieldUpdate = AccountUpdate.create(addresses.dex);
    fieldUpdate.account.delegate.set(newDelegate);
    fieldUpdate.requireSignature();

    // changing permissions back to impossible
    let permissionUpdate2 = AccountUpdate.create(addresses.dex);
    permissionUpdate2.account.permissions.set({
      ...Permissions.initial(),
      setDelegate: Permissions.impossible(),
    });
    permissionUpdate2.requireSignature();
  });
  await tx.prove();
  await tx.sign([feePayerKey, keys.dex]).send();

  Mina.getAccount(addresses.dex).delegate?.assertEquals(newDelegate);
  DexProfiler.stop().store();
}

async function upgradeabilityTests({ withVesting }: { withVesting: boolean }) {
  const DexProfiler = getProfiler('DEX profiler upgradeability tests');
  DexProfiler.start('DEX test flow');
  let Local = Mina.LocalBlockchain({
    proofsEnabled: proofsEnabled,
    enforceTransactionLimits: false,
  });
  Mina.setActiveInstance(Local);
```

```
let accountFee = Mina.accountCreationFee();
let [{ privateKey: feePayerKey, publicKey: feePayerAddress }] =
  Local.testAccounts;
let tx, balances, oldBalances;

let options = withVesting ? { lockedLiquiditySlots: 2 } : undefined;
let {
  Dex,
  DexTokenHolder,
  ModifiedDexTokenHolder,
  ModifiedDex,
  getTokenBalances,
} = createDex(options);

// analyze methods for quick error feedback
DexTokenHolder.analyzeMethods();
Dex.analyzeMethods();

// compile & deploy all zkApps
console.log('compile (token contract)...');
await TokenContract.compile();
console.log('compile (dex token holder)...');
await DexTokenHolder.compile();
console.log('compile (dex main contract)...');
await Dex.compile();

let tokenX = new TokenContract(addresses.tokenX);
let tokenY = new TokenContract(addresses.tokenY);
let dex = new Dex(addresses.dex);

console.log('deploy & init token contracts...');
tx = await Mina.transaction(feePayerAddress, () => {
  // pay fees for creating 2 token contract accounts, and fund them so each can create 2 accounts
themselves
  let feePayerUpdate = AccountUpdate.createSigned(feePayerAddress);
  feePayerUpdate.balance.subInPlace(accountFee.mul(2));
  feePayerUpdate.send({ to: addresses.tokenX, amount: accountFee.mul(2) });
  feePayerUpdate.send({ to: addresses.tokenY, amount: accountFee.mul(2) });
  tokenX.deploy();
  tokenY.deploy();
});
await tx.prove();
tx.sign([feePayerKey, keys.tokenX, keys.tokenY]);
await tx.send();
balances = getTokenBalances();
console.log(
  'Token contract tokens (X, Y):',
  balances.tokenContract.X,
  balances.tokenContract.Y
);

/**
```

```
 * # Upgradeability 1 - Happy Path
 *
 * Preconditions:
 *  - Initially the SC was deployed with the ability to be upgradable
 *
 * Actions:
 *  - deploy valid SC with upgradeable permissions
 *  - compile and upgrade new contract with different source code
 *  - set permissions to be non upgradeable
 *
 * Expected:
 *  - upgrade is successful
 *  - updates to the source code and methods are recognized
 *    (e.g. new formula applies to the swap)
 */

console.log('deploy dex contracts...');

tx = await Mina.transaction(feePayerKey, () => {
  // pay fees for creating 3 dex accounts
  AccountUpdate.fundNewAccount(feePayerAddress, 3);
  dex.deploy();
  tokenX.deployZkapp(addresses.dex, DexTokenHolder._verificationKey!);
  tokenY.deployZkapp(addresses.dex, DexTokenHolder._verificationKey!);
});
await tx.prove();
tx.sign([feePayerKey, keys.dex]);
await tx.send();

console.log('transfer tokens to user');
tx = await Mina.transaction(
  { sender: feePayerAddress, fee: accountFee.mul(1) },
  () => {
    let feePayer = AccountUpdate.createSigned(feePayerAddress);
    feePayer.balance.subInPlace(Mina.accountCreationFee().mul(4));
    feePayer.send({ to: addresses.user, amount: 20e9 }); // give users MINA to pay fees
    feePayer.send({ to: addresses.user2, amount: 20e9 });
    // transfer to fee payer so they can provide initial liquidity
    tokenX.transfer(addresses.tokenX, feePayerAddress, UInt64.from(10_000));
    tokenY.transfer(addresses.tokenY, feePayerAddress, UInt64.from(10_000));
    // mint tokens to the user (this is additional to the tokens minted at the beginning, so we can overflow the
balance
    tokenX.init2();
    tokenY.init2();
  }
);
await tx.prove();
tx.sign([feePayerKey, keys.tokenX, keys.tokenY]);
await tx.send();
[oldBalances, balances] = [balances, getTokenBalances()];
console.log('User tokens (X, Y):', balances.user.X, balances.user.Y);
console.log('User MINA:', balances.user.MINA);
console.log(
  'deploying an upgraded DexTokenHolder contract (adjusted swap method) and Dex contract'
);

console.log('compiling modified DexTokenHolder contract...');
await ModifiedDexTokenHolder.compile();

console.log('compiling modified Dex contract...');
await ModifiedDex.compile();
let modifiedDex = new ModifiedDex(addresses.dex);

tx = await Mina.transaction(feePayerAddress, () => {
  modifiedDex.deploy();
  tokenX.deployZkapp(addresses.dex, ModifiedDexTokenHolder._verificationKey!);
  tokenY.deployZkapp(addresses.dex, ModifiedDexTokenHolder._verificationKey!);
});
await tx.prove();
tx.sign([feePayerKey, keys.dex]);
await tx.send();

// Making sure that both token holder accounts have been updated with the new modified verification key
expect(
  Mina.getAccount(addresses.dex, tokenX.token.id).zkapp?.verificationKey?.data
).toEqual(ModifiedDexTokenHolder._verificationKey?.data);

expect(
  Mina.getAccount(addresses.dex, tokenY.token.id).zkapp?.verificationKey?.data
).toEqual(ModifiedDexTokenHolder._verificationKey?.data);

// this is important; we have to re-enable proof production (and verification) to make sure the proofs are
valid against the newly deployed VK
Local.setProofsEnabled(true);

console.log('supply liquidity -- base');
tx = await Mina.transaction(
  { sender: feePayerAddress, fee: accountFee.mul(1) },
  () => {
    AccountUpdate.fundNewAccount(feePayerAddress);
    modifiedDex.supplyLiquidityBase(UInt64.from(10_000), UInt64.from(10_000));
  }
);
await tx.prove();
tx.sign([feePayerKey]);
await tx.send();
[oldBalances, balances] = [balances, getTokenBalances()];
console.log('DEX liquidity (X, Y):', balances.dex.X, balances.dex.Y);

let USER_DX = 10n;
console.log('swap 10 X for Y');
tx = await Mina.transaction(addresses.user, () => {
  modifiedDex.swapX(UInt64.from(USER_DX));
});
await tx.prove();
await tx.sign([keys.user]).send();
[oldBalances, balances] = [balances, getTokenBalances()];
console.log('User tokens (X, Y):', balances.user.X, balances.user.Y);

// according to the newly modified formula &nbsp;dy = y.mul(dx).div(x.add(dx)).add(15)&nbsp;;
// the user should have a balance of 1_000_024 Y
expect(oldBalances.user.Y).toEqual(1_000_000n);
expect(balances.user.Y).toEqual(1_000_024n);

/**
 * # Upgradeability 2 and 3 - Upgrading forbidden, previous methods should still be valid
 *
 * Preconditions:
 *  - DEX contract deployed, but upgrading forbidden
 *
 * Actions:
 *  - setVerificationKey permission to impossible
 *  - try upgrading contract
 *
 * Expected:
 *  - tx fails and contract cannot be upgraded
 *  - methods from the previous contract should still be valid
 */

console.log('changing upgrade permissions to impossible');

tx = await Mina.transaction(feePayerAddress, () => {
  // pay fees for creating 3 dex accounts
  let update = AccountUpdate.createSigned(addresses.dex);
  update.account.permissions.set({
    ...Permissions.initial(),
    setVerificationKey: Permissions.impossible(),
  });
});
await tx.prove();
tx.sign([feePayerKey, keys.dex]);
await tx.send();

console.log('trying to upgrade contract - should fail');

tx = await Mina.transaction(feePayerAddress, () => {
  modifiedDex.deploy(); // cannot deploy new VK because its forbidden
});
await tx.prove();
await expect(tx.sign([feePayerKey, keys.dex]).send()).rejects.toThrow(
  /Cannot update field 'verificationKey'/
);

console.log('trying to invoke modified swap method');
// method should still be valid since the upgrade was forbidden
```

```
  USER_DX = 10n;
  console.log('swap 10 X for Y');
  tx = await Mina.transaction(addresses.user, () => {
    modifiedDex.swapX(UInt64.from(USER_DX));
  });
  await tx.prove();
  await tx.sign([keys.user]).send();
  DexProfiler.stop().store();
}
```

</file>

<file>

## path: /src/examples/zkapps/escrow/escrow.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/escrow/escrow.ts

```
import {
  SmartContract,
  method,
  UInt64,
  AccountUpdate,
  PublicKey,
} from 'o1js';

export class Escrow extends SmartContract {
  @method deposit(user: PublicKey) {
    // add your deposit logic circuit here
    // that will adjust the amount

    const payerUpdate = AccountUpdate.createSigned(user);
    payerUpdate.send({ to: this.address, amount: UInt64.from(1000000) });
  }

  @method withdraw(user: PublicKey) {
    // add your withdrawal logic circuit here
    // that will adjust the amount

    this.send({ to: user, amount: UInt64.from(1000000) });
  }
}
```

</file>

<file>

## path: /src/examples/zkapps/hello_world/hello_world.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/hello_world/hello_world.ts

```
import {
  Field,
  PrivateKey,
  SmartContract,
  State,
  method,
  state,
} from 'o1js';

export const adminPrivateKey = PrivateKey.random();
export const adminPublicKey = adminPrivateKey.toPublicKey();

export class HelloWorld extends SmartContract {
  @state(Field) x = State<Field>();

  init() {
    super.init();
    this.x.set(Field(2));
    this.account.delegate.set(adminPublicKey);
  }

  @method update(squared: Field, admin: PrivateKey) {
    const x = this.x.get();
    this.x.assertNothing();
    x.square().assertEquals(squared);
    this.x.set(squared);

    const adminPk = admin.toPublicKey();

    this.account.delegate.assertEquals(adminPk);
  }
}
```

</file>

<file>

## path: /src/examples/zkapps/hello_world/run.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/hello_world/run.ts

```
import { AccountUpdate, Field, Mina, PrivateKey } from 'o1js';
import { getProfiler } from '../../utils/profiler.js';
import { HelloWorld, adminPrivateKey } from './hello_world.js';

const HelloWorldProfier = getProfiler('Hello World');
HelloWorldProfier.start('Hello World test flow');

let txn, txn2, txn3, txn4;
// setup local ledger
let Local = Mina.LocalBlockchain({ proofsEnabled: false });
```

```
Mina.setActiveInstance(Local);

// test accounts that pays all the fees, and puts additional funds into the zkapp
const feePayer1 = Local.testAccounts[0];
const feePayer2 = Local.testAccounts[1];
const feePayer3 = Local.testAccounts[2];
const feePayer4 = Local.testAccounts[3];

// zkapp account
const zkAppPrivateKey = PrivateKey.random();
const zkAppAddress = zkAppPrivateKey.toPublicKey();
const zkAppInstance = new HelloWorld(zkAppAddress);

console.log('Deploying Hello World ....');

txn = await Mina.transaction(feePayer1.publicKey, () => {
  AccountUpdate.fundNewAccount(feePayer1.publicKey);
  zkAppInstance.deploy();
});
await txn.sign([feePayer1.privateKey, zkAppPrivateKey]).send();

const initialState =
  Mina.getAccount(zkAppAddress).zkapp?.appState?.[0].toString();

let currentState;

console.log('Initial State', initialState);

// update state with value that satisfies preconditions and correct admin private key
console.log(
  &nbsp;Updating state from ${initialState} to 4 with Admin Private Key ...&nbsp;
);

txn = await Mina.transaction(feePayer1.publicKey, () => {
  zkAppInstance.update(Field(4), adminPrivateKey);
});
await txn.prove();
await txn.sign([feePayer1.privateKey]).send();

currentState = Mina.getAccount(zkAppAddress).zkapp?.appState?.[0].toString();

if (currentState !== '4') {
  throw Error(
    &nbsp;Current state of ${currentState} does not match 4 after calling update with 4&nbsp;
  );
}

console.log(&nbsp;Current state successfully updated to ${currentState}&nbsp;);

const wrongAdminPrivateKey = PrivateKey.random();
// attempt to update state with value that satisfies preconditions and incorrect admin private key
console.log(
 &nbsp;Attempting to update state from ${currentState} to 16 with incorrect Admin Private Key ...&nbsp;
);

let correctlyFails = false;

try {
  txn = await Mina.transaction(feePayer1.publicKey, () => {
    zkAppInstance.update(Field(16), wrongAdminPrivateKey);
  });
  await txn.prove();
  await txn.sign([feePayer1.privateKey]).send();
} catch (err: any) {
  handleError(err, 'Account_delegate_precondition_unsatisfied');
}

if (!correctlyFails) {
  throw Error('We could update the state with the wrong admin key');
}

// attempt to update state with value that fails precondition x.square().assertEquals(squared).
correctlyFails = false;

try {
  console.log(
    &nbsp;Attempting to update state from ${currentState} to the value that fails precondition of 30 ...&nbsp;
  );

  txn = await Mina.transaction(feePayer1.publicKey, () => {
    zkAppInstance.update(Field(30), adminPrivateKey);
  });
  await txn.prove();
  await txn.sign([feePayer1.privateKey]).send();
} catch (err: any) {
  handleError(err, 'assertEquals');
}

if (!correctlyFails) {
  throw Error(
    'We could update the state to a value that violates the precondition'
  );
}

// attempt simultaneous “Update” method invocation by different users
correctlyFails = false;

try {
  console.log(
    &nbsp;Attempting to simultaneously update the current state of ${currentState} from multiple users
...&nbsp;
  );

  // expected to fail and current state stays at 4
  txn = await Mina.transaction(
    { sender: feePayer1.publicKey, fee: '10' },
    () => {
      zkAppInstance.update(Field(256), adminPrivateKey);
    }
  );
  await txn.prove();
  await txn.sign([feePayer1.privateKey]).send();
} catch (err: any) {
  handleError(err, 'assertEquals');
}

if (!correctlyFails) {
  throw Error(
    'We could update the state with input that fails the precondition'
  );
}

// expected to succeed and update state to 16
txn2 = await Mina.transaction({ sender: feePayer2.publicKey, fee: '2' }, () => {
  zkAppInstance.update(Field(16), adminPrivateKey);
});
await txn2.prove();
await txn2.sign([feePayer2.privateKey]).send();

currentState = Mina.getAccount(zkAppAddress).zkapp?.appState?.[0].toString();

if (currentState !== '16') {
  throw Error(
    &nbsp;Current state of ${currentState} does not match 16 after calling update with 16&nbsp;
  );
}

console.log(&nbsp;Update successful. Current state is ${currentState}.&nbsp;);

// expected to succeed and update state to 256
txn3 = await Mina.transaction({ sender: feePayer3.publicKey, fee: '1' }, () => {
  zkAppInstance.update(Field(256), adminPrivateKey);
});
await txn3.prove();
await txn3.sign([feePayer3.privateKey]).send();

currentState = Mina.getAccount(zkAppAddress).zkapp?.appState?.[0].toString();

if (currentState !== '256') {
  throw Error(
    &nbsp;Current state of ${currentState} does not match 256 after calling update with 256&nbsp;
  );
}

console.log(&nbsp;Update successful. Current state is ${currentState}.&nbsp;);
```

```
correctlyFails = false;

try {
  // expected to fail and current state remains 256
  txn4 = await Mina.transaction(
    { sender: feePayer4.publicKey, fee: '1' },
    () => {
      zkAppInstance.update(Field(16), adminPrivateKey);
    }
  );
  await txn4.prove();
  await txn4.sign([feePayer4.privateKey]).send();
} catch (err: any) {
  handleError(err, 'assertEquals');
}

if (!correctlyFails) {
  throw Error(
    'We could update the state with input that fails the precondition'
  );
}

/**
 * Test for expected failure case. Original error thrown if not expected failure case.
 * @param {any} error  The error thrown in the catch block.
 * @param {string} errorMessage  The expected error message.
 */

function handleError(error: any, errorMessage: string) {
  currentState = Mina.getAccount(zkAppAddress).zkapp?.appState?.[0].toString();

  if (error.message.includes(errorMessage)) {
    correctlyFails = true;
    console.log(
      &nbsp;Update correctly rejected with failing precondition. Current state is still ${currentState}.&nbsp;
    );
  } else {
    throw Error(error);
  }
}

HelloWorldProfier.stop().store();
```

</file>

<file>

## path: /src/examples/zkapps/hello_world/run_live.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/hello_world/run_live.ts

```
// Live integration test against real Mina network.
import {
  AccountUpdate,
  Field,
  Lightnet,
  Mina,
  PrivateKey,
  fetchAccount,
} from 'o1js';
import { HelloWorld, adminPrivateKey } from './hello_world.js';

const useCustomLocalNetwork = process.env.USE_CUSTOM_LOCAL_NETWORK === 'true';
const zkAppKey = PrivateKey.random();
const zkAppAddress = zkAppKey.toPublicKey();
const transactionFee = 100_000_000;

// Network configuration
const network = Mina.Network({
  mina: useCustomLocalNetwork
    ? 'http://localhost:8080/graphql'
    : 'https://proxy.berkeley.minaexplorer.com/graphql',
  lightnetAccountManager: 'http://localhost:8181',
});
Mina.setActiveInstance(network);

// Fee payer setup
const senderKey = useCustomLocalNetwork
  ? (await Lightnet.acquireKeyPair()).privateKey
  : PrivateKey.random();
const sender = senderKey.toPublicKey();
if (!useCustomLocalNetwork) {
  console.log(&nbsp;Funding the fee payer account.&nbsp;);
  await Mina.faucet(sender);
}
console.log(&nbsp;Fetching the fee payer account information.&nbsp;);
const accountDetails = (await fetchAccount({ publicKey: sender })).account;
console.log(
  &nbsp;Using the fee payer account ${sender.toBase58()} with nonce: ${
    accountDetails?.nonce
  } and balance: ${accountDetails?.balance}.&nbsp;
);
console.log('');

// zkApp compilation
console.log('Compiling the smart contract.');
const { verificationKey } = await HelloWorld.compile();
const zkApp = new HelloWorld(zkAppAddress);
console.log('');

// zkApp deployment
console.log(&nbsp;Deploying zkApp for public key ${zkAppAddress.toBase58()}.&nbsp;);
```

```
let transaction = await Mina.transaction(
  { sender, fee: transactionFee },
  () => {
    AccountUpdate.fundNewAccount(sender);
    zkApp.deploy({ verificationKey });
  }
);
transaction.sign([senderKey, zkAppKey]);
console.log('Sending the transaction.');
let pendingTx = await transaction.send();
if (pendingTx.hash() !== undefined) {
  console.log(&nbsp;Success! Deploy transaction sent.
Your smart contract will be deployed
as soon as the transaction is included in a block.
Txn hash: ${pendingTx.hash()}&nbsp;);
}
console.log('Waiting for transaction inclusion in a block.');
await pendingTx.wait({ maxAttempts: 90 });
console.log('');

// zkApp state update
console.log('Trying to update deployed zkApp state.');
transaction = await Mina.transaction({ sender, fee: transactionFee }, () => {
  zkApp.update(Field(4), adminPrivateKey);
});
await transaction.sign([senderKey]).prove();
console.log('Sending the transaction.');
pendingTx = await transaction.send();
if (pendingTx.hash() !== undefined) {
  console.log(&nbsp;Success! Update transaction sent.
Your smart contract state will be updated
as soon as the transaction is included in a block.
Txn hash: ${pendingTx.hash()}&nbsp;);
}
console.log('Waiting for transaction inclusion in a block.');
await pendingTx.wait({ maxAttempts: 90 });
console.log('');

// zkApp state check
console.log('Validating zkApp state update.');
try {
  (await zkApp.x.fetch())?.assertEquals(Field(4));
} catch (error) {
  throw new Error(
    &nbsp;On-chain zkApp account state doesn't match the expected state. ${error}&nbsp;
  );
}
console.log('Success!');

// Tear down
const keyPairReleaseMessage = await Lightnet.releaseKeyPair({
  publicKey: sender.toBase58(),
});
if (keyPairReleaseMessage) console.info(keyPairReleaseMessage);
```

</file>

<file>

## path: /src/examples/zkapps/local_events_zkapp.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/local_events_zkapp.ts

```
import {
  Field,
  state,
  State,
  method,
  UInt64,
  PrivateKey,
  SmartContract,
  Mina,
  AccountUpdate,
  isReady,
  UInt32,
  PublicKey,
  Struct,
} from 'o1js';

const doProofs = false;

await isReady;

class Event extends Struct({ pub: PublicKey, value: Field }) {}

class SimpleZkapp extends SmartContract {
  @state(Field) x = State<Field>();

  events = {
    complexEvent: Event,
    simpleEvent: Field,
  };

  init() {
    super.init();
    this.x.set(initialState);
  }

  @method update(y: Field) {
    this.emitEvent('complexEvent', {
      pub: PrivateKey.random().toPublicKey(),
      value: y,
    });
    this.emitEvent('simpleEvent', y);
    let x = this.x.get();
    this.x.assertEquals(x);
    this.x.set(x.add(y));
  }
}

let Local = Mina.LocalBlockchain({ proofsEnabled: false });
Mina.setActiveInstance(Local);

// a test account that pays all the fees, and puts additional funds into the zkapp
let feePayerKey = Local.testAccounts[0].privateKey;
let feePayer = Local.testAccounts[0].publicKey;

// the zkapp account
let zkappKey = PrivateKey.random();
let zkappAddress = zkappKey.toPublicKey();

let initialState = Field(1);
let zkapp = new SimpleZkapp(zkappAddress);

if (doProofs) {
  console.log('compile');
  await SimpleZkapp.compile();
}

console.log('deploy');
let tx = await Mina.transaction(feePayer, () => {
  AccountUpdate.fundNewAccount(feePayer);
  zkapp.deploy();
});
await tx.sign([feePayerKey, zkappKey]).send();

console.log('call update');
tx = await Mina.transaction(feePayer, () => {
  zkapp.update(Field(1));
});
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('call update');
tx = await Mina.transaction(feePayer, () => {
  zkapp.update(Field(2));
});
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('---- emitted events: ----');
// fetches all events from zkapp starting block height 0
let events = await zkapp.fetchEvents(UInt32.from(0));
console.log(events);
console.log('---- emitted events: ----');
```

```
// fetches all events from zkapp starting block height 0 and ending at block height 10
events = await zkapp.fetchEvents(UInt32.from(0), UInt64.from(10));
console.log(events);
console.log('---- emitted events: ----');
// fetches all events
events = await zkapp.fetchEvents();
console.log(events);
```

</file>

<file>

## path: /src/examples/zkapps/merkle_tree/merkle_zkapp.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/merkle_tree/merkle_zkapp.ts

```
/*
Description:

This example describes how developers can use Merkle Trees as a basic off-chain storage tool.

zkApps on Mina can only store a small amount of data on-chain, but many use cases require your
application to at least reference big amounts of data.
Merkle Trees give developers the power of storing large amounts of data off-chain, but proving its integrity to
the on-chain smart contract!


! Unfamiliar with Merkle Trees? No problem! Check out https://blog.ethereum.org/2015/11/15/merkling-in-
ethereum/
*/

import {
  SmartContract,
  Poseidon,
  Field,
  State,
  state,
  PublicKey,
  Mina,
  method,
  UInt32,
  PrivateKey,
  AccountUpdate,
  MerkleTree,
  MerkleWitness,
  Struct,
} from 'o1js';

const doProofs = true;

class MyMerkleWitness extends MerkleWitness(8) {}
class Account extends Struct({
  publicKey: PublicKey,
  points: UInt32,
}) {
  hash(): Field {
    return Poseidon.hash(Account.toFields(this));
  }

  addPoints(points: number) {
    return new Account({
      publicKey: this.publicKey,
      points: this.points.add(points),
    });
  }
}
// we need the initiate tree root in order to tell the contract about our off-chain storage
let initialCommitment: Field = Field(0);
/*
  We want to write a smart contract that serves as a leaderboard,
  but only has the commitment of the off-chain storage stored in an on-chain variable.
  The accounts of all participants will be stored off-chain!
  If a participant can guess the preimage of a hash, they will be granted one point :)
*/

class Leaderboard extends SmartContract {
  // a commitment is a cryptographic primitive that allows us to commit to data, with the ability to "reveal" it
later
  @state(Field) commitment = State<Field>();

  @method init() {
    super.init();
    this.commitment.set(initialCommitment);
  }

  @method
  guessPreimage(guess: Field, account: Account, path: MyMerkleWitness) {
    // this is our hash! its the hash of the preimage "22", but keep it a secret!
    let target = Field(
      '17057234437185175411792943285768571642343179330449434169483610110583519635705'
    );
    // if our guess preimage hashes to our target, we won a point!
    Poseidon.hash([guess]).assertEquals(target);

    // we fetch the on-chain commitment
    let commitment = this.commitment.get();
    this.commitment.assertEquals(commitment);

    // we check that the account is within the committed Merkle Tree
    path.calculateRoot(account.hash()).assertEquals(commitment);

    // we update the account and grant one point!
    let newAccount = account.addPoints(1);

    // we calculate the new Merkle Root, based on the account changes
    let newCommitment = path.calculateRoot(newAccount.hash());

    this.commitment.set(newCommitment);
  }
}

type Names = 'Bob' | 'Alice' | 'Charlie' | 'Olivia';

let Local = Mina.LocalBlockchain({ proofsEnabled: doProofs });
Mina.setActiveInstance(Local);
let initialBalance = 10_000_000_000;

let feePayerKey = Local.testAccounts[0].privateKey;
let feePayer = Local.testAccounts[0].publicKey;

// the zkapp account
let zkappKey = PrivateKey.random();
let zkappAddress = zkappKey.toPublicKey();

// this map serves as our off-chain in-memory storage
let Accounts: Map<string, Account> = new Map<Names, Account>(
  ['Bob', 'Alice', 'Charlie', 'Olivia'].map((name: string, index: number) => {
    return [
      name as Names,
      new Account({
        publicKey: Local.testAccounts[index].publicKey,
        points: UInt32.from(0),
      }),
    ];
  })
);

// we now need "wrap" the Merkle tree around our off-chain storage
// we initialize a new Merkle Tree with height 8
const Tree = new MerkleTree(8);

Tree.setLeaf(0n, Accounts.get('Bob')!.hash());
Tree.setLeaf(1n, Accounts.get('Alice')!.hash());
Tree.setLeaf(2n, Accounts.get('Charlie')!.hash());
Tree.setLeaf(3n, Accounts.get('Olivia')!.hash());

// now that we got our accounts set up, we need the commitment to deploy our contract!
initialCommitment = Tree.getRoot();

let leaderboardZkApp = new Leaderboard(zkappAddress);
console.log('Deploying leaderboard..');
if (doProofs) {
  await Leaderboard.compile();
}
```

```
let tx = await Mina.transaction(feePayer, () => {
  AccountUpdate.fundNewAccount(feePayer).send({
    to: zkappAddress,
    amount: initialBalance,
  });
  leaderboardZkApp.deploy();
});
await tx.prove();
await tx.sign([feePayerKey, zkappKey]).send();

console.log('Initial points: ' + Accounts.get('Bob')?.points);

console.log('Making guess..');
await makeGuess('Bob', 0n, 22);

console.log('Final points: ' + Accounts.get('Bob')?.points);

async function makeGuess(name: Names, index: bigint, guess: number) {
  let account = Accounts.get(name)!;
  let w = Tree.getWitness(index);
  let witness = new MyMerkleWitness(w);

  let tx = await Mina.transaction(feePayer, () => {
    leaderboardZkApp.guessPreimage(Field(guess), account, witness);
  });
  await tx.prove();
  await tx.sign([feePayerKey, zkappKey]).send();

  // if the transaction was successful, we can update our off-chain storage as well
  account.points = account.points.add(1);
  Tree.setLeaf(index, account.hash());
  leaderboardZkApp.commitment.get().assertEquals(Tree.getRoot());
}
```

</file>

<file>

## path: /src/examples/zkapps/reducer/reducer.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/reducer/reducer.ts

```
import {
  Field,
  state,
  State,
  method,
  PrivateKey,
  SmartContract,
  Mina,
  AccountUpdate,
  isReady,
  Permissions,
  Reducer,
} from 'o1js';

await isReady;

const INCREMENT = Field(1);

class CounterZkapp extends SmartContract {
  // the "reducer" field describes a type of action that we can dispatch, and reduce later
  reducer = Reducer({ actionType: Field });

  // on-chain version of our state. it will typically lag behind the
  // version that's implicitly represented by the list of actions
  @state(Field) counter = State<Field>();
  // helper field to store the point in the action history that our on-chain state is at
  @state(Field) actionState = State<Field>();

  @method incrementCounter() {
    this.reducer.dispatch(INCREMENT);
  }

  @method rollupIncrements() {
    // get previous counter & actions hash, assert that they're the same as on-chain values
    let counter = this.counter.get();
    this.counter.assertEquals(counter);
    let actionState = this.actionState.get();
    this.actionState.assertEquals(actionState);

    // compute the new counter and hash from pending actions
    let pendingActions = this.reducer.getActions({
      fromActionState: actionState,
    });

    let { state: newCounter, actionState: newActionState } =
      this.reducer.reduce(
        pendingActions,
        // state type
        Field,
        // function that says how to apply an action
        (state: Field, _action: Field) => {
          return state.add(1);
        },
        { state: counter, actionState }
      );

    // update on-chain state
    this.counter.set(newCounter);
    this.actionState.set(newActionState);
  }
}
```

```
const doProofs = true;
const initialCounter = Field(0);

let Local = Mina.LocalBlockchain({ proofsEnabled: doProofs });
Mina.setActiveInstance(Local);

// a test account that pays all the fees, and puts additional funds into the zkapp
let feePayerKey = Local.testAccounts[0].privateKey;
let feePayer = Local.testAccounts[0].publicKey;

// the zkapp account
let zkappKey = PrivateKey.fromBase58(
  'EKEQc95PPQZnMY9d9p1vq1MWLeDJKtvKj4V75UDG3rjnf32BerWD'
);
let zkappAddress = zkappKey.toPublicKey();
let zkapp = new CounterZkapp(zkappAddress);
if (doProofs) {
  console.log('compile');
  await CounterZkapp.compile();
} else {
  // TODO: if we don't do this, then &nbsp;analyzeMethods()&nbsp; will be called during
&nbsp;runUnchecked()&nbsp; in the tx callback below,
  // which currently fails due to &nbsp;finalize_is_running&nbsp; in snarky not resetting internal state, and
instead setting is_running unconditionally to false,
  // so we can't nest different snarky circuit runners
  CounterZkapp.analyzeMethods();
}

console.log('deploy');
let tx = await Mina.transaction(feePayer, () => {
  AccountUpdate.fundNewAccount(feePayer);
  zkapp.deploy();
  zkapp.counter.set(initialCounter);
  zkapp.actionState.set(Reducer.initialActionState);
});
await tx.sign([feePayerKey, zkappKey]).send();

console.log('applying actions..');

console.log('action 1');

tx = await Mina.transaction(feePayer, () => {
  zkapp.incrementCounter();
});
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('action 2');
tx = await Mina.transaction(feePayer, () => {
  zkapp.incrementCounter();
});
```

```
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('action 3');
tx = await Mina.transaction(feePayer, () => {
  zkapp.incrementCounter();
});
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('rolling up pending actions..');

console.log('state before: ' + zkapp.counter.get());

tx = await Mina.transaction(feePayer, () => {
  zkapp.rollupIncrements();
});
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('state after rollup: ' + zkapp.counter.get());

console.log('applying more actions');

console.log('action 4');
tx = await Mina.transaction(feePayer, () => {
  zkapp.incrementCounter();
});
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('action 5');
tx = await Mina.transaction(feePayer, () => {
  zkapp.incrementCounter();
});
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('rolling up pending actions..');

console.log('state before: ' + zkapp.counter.get());

tx = await Mina.transaction(feePayer, () => {
  zkapp.rollupIncrements();
});
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('state after rollup: ' + zkapp.counter.get());
```

</file>

<file>

## path: /src/examples/zkapps/reducer/reducer_composite.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/reducer/reducer_composite.ts

```
import {
  Field,
  state,
  State,
  method,
  PrivateKey,
  SmartContract,
  Mina,
  AccountUpdate,
  isReady,
  Bool,
  Struct,
  Reducer,
  Provable,
} from 'o1js';
import assert from 'node:assert/strict';
import { getProfiler } from '../../utils/profiler.js';

await isReady;

class MaybeIncrement extends Struct({
  isIncrement: Bool,
  otherData: Field,
}) {}
const INCREMENT = { isIncrement: Bool(true), otherData: Field(0) };

class CounterZkapp extends SmartContract {
  // the "reducer" field describes a type of action that we can dispatch, and reduce later
  reducer = Reducer({ actionType: MaybeIncrement });

  // on-chain version of our state. it will typically lag behind the
  // version that's implicitly represented by the list of actions
  @state(Field) counter = State<Field>();
  // helper field to store the point in the action history that our on-chain state is at
  @state(Field) actionState = State<Field>();

  @method incrementCounter() {
    this.reducer.dispatch(INCREMENT);
  }
  @method dispatchData(data: Field) {
    this.reducer.dispatch({ isIncrement: Bool(false), otherData: data });
  }

  @method rollupIncrements() {
    // get previous counter & actions hash, assert that they're the same as on-chain values
    let counter = this.counter.get();
    this.counter.assertEquals(counter);
    let actionState = this.actionState.get();
    this.actionState.assertEquals(actionState);

    // compute the new counter and hash from pending actions
    let pendingActions = this.reducer.getActions({
      fromActionState: actionState,
    });

    let { state: newCounter, actionState: newActionState } =
      this.reducer.reduce(
        pendingActions,
        // state type
        Field,
        // function that says how to apply an action
        (state: Field, action: MaybeIncrement) => {
          return Provable.if(action.isIncrement, state.add(1), state);
        },
        { state: counter, actionState }
      );

    // update on-chain state
    this.counter.set(newCounter);
    this.actionState.set(newActionState);
  }
}

const ReducerProfiler = getProfiler('Reducer zkApp');
ReducerProfiler.start('Reducer zkApp test flow');
const doProofs = true;
const initialCounter = Field(0);

let Local = Mina.LocalBlockchain({ proofsEnabled: doProofs });
Mina.setActiveInstance(Local);

// a test account that pays all the fees, and puts additional funds into the zkapp
let feePayerKey = Local.testAccounts[0].privateKey;
let feePayer = Local.testAccounts[0].publicKey;

// the zkapp account
let zkappKey = PrivateKey.fromBase58(
  'EKEQc95PPQZnMY9d9p1vq1MWLeDJKtvKj4V75UDG3rjnf32BerWD'
);
let zkappAddress = zkappKey.toPublicKey();
let zkapp = new CounterZkapp(zkappAddress);
if (doProofs) {
  console.log('compile');
  await CounterZkapp.compile();
}

console.log('deploy');
```

```
let tx = await Mina.transaction(feePayer, () => {
  AccountUpdate.fundNewAccount(feePayer);
  zkapp.deploy();
  zkapp.counter.set(initialCounter);
  zkapp.actionState.set(Reducer.initialActionState);
});
await tx.sign([feePayerKey, zkappKey]).send();

console.log('applying actions..');

console.log('action 1');

tx = await Mina.transaction(feePayer, () => {
  zkapp.incrementCounter();
});
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('action 2');
tx = await Mina.transaction(feePayer, () => {
  zkapp.incrementCounter();
});
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('action 3');
tx = await Mina.transaction(feePayer, () => {
  zkapp.incrementCounter();
});
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('rolling up pending actions..');

console.log('state before: ' + zkapp.counter.get());

tx = await Mina.transaction(feePayer, () => {
  zkapp.rollupIncrements();
});
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('state after rollup: ' + zkapp.counter.get());
assert.deepEqual(zkapp.counter.get().toString(), '3');

console.log('applying more actions');

console.log('action 4 (no increment)');
tx = await Mina.transaction(feePayer, () => {
  zkapp.dispatchData(Field.random());
});
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('action 5');
tx = await Mina.transaction(feePayer, () => {
  zkapp.incrementCounter();
});
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('rolling up pending actions..');

console.log('state before: ' + zkapp.counter.get());

tx = await Mina.transaction(feePayer, () => {
  zkapp.rollupIncrements();
});
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('state after rollup: ' + zkapp.counter.get());
assert.equal(zkapp.counter.get().toString(), '4');
ReducerProfiler.stop().store();
```

</file>

<file>

## path: /src/examples/zkapps/set_local_preconditions_zkapp.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/set_local_preconditions_zkapp.ts

```
/*
Description:

This example described how developers can manipulate the network state of the local blockchain instance.
Changing preconditions might be useful for integration tests, when you want to test your smart contracts
behavior in different situations.
For example, you only want your smart contract to initiate a pay out when the
&nbsp;blockchainLength&nbsp; is at a special height. (lock up period)
*/

import {
  method,
  UInt64,
  PrivateKey,
  SmartContract,
  Mina,
  AccountUpdate,
  isReady,
  Permissions,
  DeployArgs,
  UInt32,
} from 'o1js';

const doProofs = false;

await isReady;

class SimpleZkapp extends SmartContract {
  @method blockheightEquals(y: UInt32) {
    let length = this.network.blockchainLength.get();
    this.network.blockchainLength.assertEquals(length);

    length.assertEquals(y);
  }
}

let Local = Mina.LocalBlockchain({ proofsEnabled: doProofs });
Mina.setActiveInstance(Local);

// a test account that pays all the fees, and puts additional funds into the zkapp
let feePayerKey = Local.testAccounts[0].privateKey;
let feePayer = Local.testAccounts[0].publicKey;

// the zkapp account
let zkappKey = PrivateKey.random();
let zkappAddress = zkappKey.toPublicKey();

let zkapp = new SimpleZkapp(zkappAddress);

if (doProofs) {
  console.log('compile');
  await SimpleZkapp.compile();
}

console.log('deploy');
let tx = await Mina.transaction(feePayer, () => {
  AccountUpdate.fundNewAccount(feePayer);
  zkapp.deploy();
});
await tx.sign([feePayerKey, zkappKey]).send();

let blockHeight: UInt32 = UInt32.zero;

console.log('assert block height 0');
tx = await Mina.transaction(feePayer, () => {
  // block height starts at 0
  zkapp.blockheightEquals(UInt32.from(blockHeight));
});
await tx.prove();
await tx.sign([feePayerKey]).send();

blockHeight = UInt32.from(500);
```

```
Local.setBlockchainLength(blockHeight);

console.log('assert block height 500');
tx = await Mina.transaction(feePayer, () => {
  zkapp.blockheightEquals(UInt32.from(blockHeight));
});
await tx.prove();
await tx.sign([feePayerKey]).send();

blockHeight = UInt32.from(300);
Local.setBlockchainLength(UInt32.from(5));
console.log('invalid block height precondition');
try {
  tx = await Mina.transaction(feePayer, () => {
    zkapp.blockheightEquals(UInt32.from(blockHeight));
  });
  await tx.prove();
  await tx.sign([feePayerKey]).send();
} catch (error) {
  console.log(
    &nbsp;Expected to fail! block height is ${Local.getNetworkState().blockchainLength.toString()}, but trying
to assert ${blockHeight.toString()}&nbsp;
  );
}
```

</file>

<file>

## path: /src/examples/zkapps/simple_zkapp_payment.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/simple_zkapp_payment.ts

```
import {
  isReady,
  method,
  Mina,
  AccountUpdate,
  PrivateKey,
  SmartContract,
  UInt64,
  shutdown,
  Permissions,
} from 'o1js';

await isReady;

class SendMINAExample extends SmartContract {
  init() {
    super.init();
    this.account.permissions.set({
      ...Permissions.default(),
      send: Permissions.proofOrSignature(),
    });
  }

  @method withdraw(amount: UInt64) {
    this.send({ to: this.sender, amount });
  }

  @method deposit(amount: UInt64) {
    let senderUpdate = AccountUpdate.createSigned(this.sender);
    senderUpdate.send({ to: this, amount });
  }
}

let proofsEnabled = false;
if (proofsEnabled) await SendMINAExample.compile();

let Local = Mina.LocalBlockchain({ proofsEnabled });
Mina.setActiveInstance(Local);

// a test account that pays all the fees, and puts additional funds into the zkapp
let feePayerKey = Local.testAccounts[0].privateKey;
let feePayer = Local.testAccounts[0].publicKey;

// the zkapp account
let zkappKey = PrivateKey.random();
let zkappAddress = zkappKey.toPublicKey();

let account1Key = PrivateKey.random();
let account1Address = account1Key.toPublicKey();

let account2Key = PrivateKey.random();
let account2Address = account2Key.toPublicKey();

function printBalances() {
  console.log(
    &nbsp;zkApp balance:    ${Mina.getBalance(zkappAddress).div(1e9)} MINA&nbsp;
  );
  console.log(
    &nbsp;account1 balance: ${Mina.getBalance(account1Address).div(1e9)} MINA&nbsp;
  );
  console.log(
    &nbsp;account2 balance: ${Mina.getBalance(account2Address).div(1e9)} MINA\n&nbsp;
  );
}

let zkapp = new SendMINAExample(zkappAddress);
let tx;

console.log('deploy and fund user accounts');
tx = await Mina.transaction(feePayer, () => {
  let feePayerUpdate = AccountUpdate.fundNewAccount(feePayer, 3);
  feePayerUpdate.send({ to: account1Address, amount: 2e9 });
  feePayerUpdate.send({ to: account2Address, amount: 0 }); // just touch account #2, so it's created
  zkapp.deploy();
});
await tx.sign([feePayerKey, zkappKey]).send();
printBalances();

console.log('---------- deposit MINA into zkApp (with proof) ----------');
tx = await Mina.transaction(account1Address, () => {
  zkapp.deposit(UInt64.from(1e9));
});
await tx.prove();
await tx.sign([account1Key]).send();
printBalances();

console.log('---------- send MINA from zkApp (with proof) ----------');
tx = await Mina.transaction(account1Address, () => {
  zkapp.withdraw(UInt64.from(1e9));
});
await tx.prove();
await tx.sign([account1Key]).send();
printBalances();

console.log(
  '---------- send MINA between accounts (with signature) ----------'
);
tx = await Mina.transaction(account1Address, () => {
  let account1Update = AccountUpdate.createSigned(account1Address);
  account1Update.send({ to: account2Address, amount: 1e9 });
});
await tx.sign([account1Key]).send();
printBalances();

shutdown();
```

</file>

<file>

## path: /src/examples/zkapps/simple_zkapp_with_proof.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/simple_zkapp_with_proof.ts

```
import {
  Field,
  state,
  State,
  method,
  PrivateKey,
  SmartContract,
  Mina,
  AccountUpdate,
  isReady,
  ZkappPublicInput,
  SelfProof,
  verify,
  Empty,
} from 'o1js';

await isReady;

class TrivialZkapp extends SmartContract {
  @method proveSomething(hasToBe1: Field) {
    hasToBe1.assertEquals(1);
  }
}
class TrivialProof extends TrivialZkapp.Proof() {}

class NotSoSimpleZkapp extends SmartContract {
  @state(Field) x = State<Field>();

  @method initialize(proof: TrivialProof) {
    proof.verify();
    this.x.set(Field(1));
  }

  @method update(
    y: Field,
    oldProof: SelfProof<ZkappPublicInput, Empty>,
    trivialProof: TrivialProof
  ) {
    oldProof.verify();
    trivialProof.verify();
    let x = this.x.get();
    this.x.assertEquals(x);
    this.x.set(x.add(y));
  }
}

let Local = Mina.LocalBlockchain();
Mina.setActiveInstance(Local);

// a test account that pays all the fees, and puts additional funds into the zkapp
let feePayerKey = Local.testAccounts[0].privateKey;
let feePayer = Local.testAccounts[0].publicKey;

// the zkapp account
let zkappKey = PrivateKey.random();
let zkappAddress = zkappKey.toPublicKey();

// trivial zkapp account
let zkappKey2 = PrivateKey.random();
let zkappAddress2 = zkappKey2.toPublicKey();

// compile and prove trivial zkapp
console.log('compile (trivial zkapp)');
let { verificationKey: trivialVerificationKey } = await TrivialZkapp.compile();
// TODO: should we have a simpler API for zkapp proving without
// submitting transactions? or is this an irrelevant use case?
// would also improve the return type -- &nbsp;Proof&nbsp; instead of &nbsp;(Proof | undefined)[]&nbsp;
console.log('prove (trivial zkapp)');
let [trivialProof] = await (
  await Mina.transaction(feePayer, () => {
    new TrivialZkapp(zkappAddress2).proveSomething(Field(1));
  })
).prove();

trivialProof = await testJsonRoundtripAndVerify(
  TrivialProof,
  trivialProof,
  trivialVerificationKey
);

console.log('compile');
let { verificationKey } = await NotSoSimpleZkapp.compile();

let zkapp = new NotSoSimpleZkapp(zkappAddress);

console.log('deploy');
let tx = await Mina.transaction(feePayer, () => {
  AccountUpdate.fundNewAccount(feePayer);
  zkapp.deploy({ zkappKey });
});
await tx.prove();
await tx.sign([feePayerKey]).send();

console.log('initialize');
tx = await Mina.transaction(feePayerKey, () => {
  zkapp.initialize(trivialProof!);
});
let [proof] = await tx.prove();
await tx.sign([feePayerKey]).send();

proof = await testJsonRoundtripAndVerify(
  NotSoSimpleZkapp.Proof(),
  proof,
  verificationKey
);

console.log('initial state: ' + zkapp.x.get());

console.log('update');
tx = await Mina.transaction(feePayer, () => {
  zkapp.update(Field(3), proof!, trivialProof!);
});
[proof] = await tx.prove();
await tx.sign([feePayerKey]).send();

proof = await testJsonRoundtripAndVerify(
  NotSoSimpleZkapp.Proof(),
  proof,
  verificationKey
);

console.log('state 2: ' + zkapp.x.get());

console.log('update');
tx = await Mina.transaction(feePayer, () => {
  zkapp.update(Field(3), proof!, trivialProof!);
});
[proof] = await tx.prove();
await tx.sign([feePayerKey]).send();

proof = await testJsonRoundtripAndVerify(
  NotSoSimpleZkapp.Proof(),
  proof,
  verificationKey
);

console.log('final state: ' + zkapp.x.get());

async function testJsonRoundtripAndVerify(
  Proof: any,
  proof: any,
  verificationKey: { data: string }
): Promise<any> {
  let jsonProof = proof.toJSON();
  console.log(
    'json proof:',
    JSON.stringify({ ...jsonProof, proof: jsonProof.proof.slice(0, 10) + '..' })
  );
  let ok = await verify(jsonProof, verificationKey.data);
  if (!ok) throw Error('proof cannot be verified');
  return Proof.fromJSON(jsonProof);
}
```

</file>

<file>

## path: /src/examples/zkapps/sudoku/index.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/sudoku/index.ts

```
import { Sudoku, SudokuZkApp } from './sudoku.js';
import { cloneSudoku, generateSudoku, solveSudoku } from './sudoku-lib.js';
import { AccountUpdate, Mina, PrivateKey, shutdown } from 'o1js';

// setup
const Local = Mina.LocalBlockchain();
Mina.setActiveInstance(Local);

const { publicKey: account, privateKey: accountKey } = Local.testAccounts[0];
const sudoku = generateSudoku(0.5);
const zkAppPrivateKey = PrivateKey.random();
const zkAppAddress = zkAppPrivateKey.toPublicKey();
// create an instance of the smart contract
const zkApp = new SudokuZkApp(zkAppAddress);

let methods = SudokuZkApp.analyzeMethods();
console.log(
  'first 5 gates of submitSolution method:',
  ...methods.submitSolution.gates.slice(0, 5)
);

console.log('Deploying and initializing Sudoku...');
await SudokuZkApp.compile();
let tx = await Mina.transaction(account, () => {
  AccountUpdate.fundNewAccount(account);
  zkApp.deploy();
  zkApp.update(Sudoku.from(sudoku));
});
await tx.prove();
/**
 * note: this tx needs to be signed with the &nbsp;zkAppPrivateKey&nbsp;, because &nbsp;deploy&nbsp;
uses &nbsp;requireSignature()&nbsp; under the hood,
 * so one of the account updates in this tx has to be authorized with a signature (vs proof).
 * this is necessary for the deploy tx because the initial permissions for all account fields are "signature".
 * (but &nbsp;deploy()&nbsp; changes some of those permissions to "proof" and adds the verification key
that enables proofs.
 * that's why we don't need &nbsp;tx.sign()&nbsp; for the later transactions.)
 */
await tx.sign([zkAppPrivateKey, accountKey]).send();

console.log('Is the sudoku solved?', zkApp.isSolved.get().toBoolean());

let solution = solveSudoku(sudoku);
if (solution === undefined) throw Error('cannot happen');

// submit a wrong solution
let noSolution = cloneSudoku(solution);
noSolution[0][0] = (noSolution[0][0] % 9) + 1;

console.log('Submitting wrong solution...');
try {
  let tx = await Mina.transaction(account, () => {
    zkApp.submitSolution(Sudoku.from(sudoku), Sudoku.from(noSolution));
  });
  await tx.prove();
  await tx.sign([accountKey]).send();
} catch {
  console.log('There was an error submitting the solution, as expected');
}
console.log('Is the sudoku solved?', zkApp.isSolved.get().toBoolean());

// submit the actual solution
console.log('Submitting solution...');
tx = await Mina.transaction(account, () => {
  zkApp.submitSolution(Sudoku.from(sudoku), Sudoku.from(solution!));
});
await tx.prove();
await tx.sign([accountKey]).send();
console.log('Is the sudoku solved?', zkApp.isSolved.get().toBoolean());

// cleanup
await shutdown();
```

</file>

<file>

## path: /src/examples/zkapps/sudoku/sudoku-lib.js

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/sudoku/sudoku-lib.js

```
export { generateSudoku, solveSudoku, cloneSudoku };

/**
 * Generates a random 9x9 sudoku. Cells are either filled out (1,...,9) or empty (0).
 *
 * @param {number?} difficulty number between 0 (easiest = full sudoku) and 1 (hardest = empty sudoku)
 * @returns {number[][]} the sudoku
 */
function generateSudoku(difficulty = 0.5) {
  let solution = solveSudokuInternal(emptySudoku(), false);
  let partial = deleteRandomValues(solution, difficulty);
  return partial;
}

function deleteRandomValues(sudoku, p) {
  // p \in [0,1] ... probability to delete a value
  return sudoku.map((row) => row.map((x) => (Math.random() < p ? 0 : x)));
}

// sudoku = {0,...,9}^(9x9) matrix, 0 means empty cell
function emptySudoku() {
  return Array.from({ length: 9 }, () => Array(9).fill(0));
}
```

```
/**
 * Solve a given sudoku. Returns undefined if there is no solution.
 *
 * @param {number[][]} sudoku - The input sudoku with some cell values equal to zero
 * @returns {number[][] | undefined} - The full sudoku, or undefined if no solution exists
 */
function solveSudoku(sudoku) {
  return solveSudokuInternal(sudoku, true);
}

function solveSudokuInternal(sudoku, deterministic, possible) {
  // find *a* compatible solution to the sudoku - not checking for uniqueness
  // if deterministic = true: always take the smallest possible cell value
  // if deterministic = false: take random compatible values
  if (possible === undefined) {
    possible = possibleFromSudoku(sudoku);
    sudoku = cloneSudoku(sudoku);
  }
  while (true) {
    let [i, j, n] = cellWithFewestPossible(sudoku, possible);

    // no free values => sudoku is solved!
    if (n === Infinity) return sudoku;

    if (n === 1) {
      let x = chooseFirstPossible(i, j, possible);
      // console.log('determined', x, 'at', i, j);
      sudoku[i][j] = x;
      fixValue(i, j, x, possible);
      continue;
    }

    while (true) {
      let x = deterministic
        ? chooseFirstPossible(i, j, possible)
        : chooseRandomPossible(i, j, possible);

      // no values possible! we failed to get a solution
      if (x === 0) return;

      let sudoku_ = cloneSudoku(sudoku);
      let possible_ = cloneSudoku(possible);
      sudoku_[i][j] = x;
      fixValue(i, j, x, possible_);

      let solution = solveSudokuInternal(sudoku_, deterministic, possible_);

      // found a solution? return it!
      if (solution !== undefined) return solution;

      // there is no solution with x at i, j!
      // mark this value as impossible and try again
      possible[i][j][x - 1] = 0;
    }
  }
}

// possible = {0,1}^(9x9x9) tensor of possibilities
// possible[i][j][x] contains a 1 if entry x \in {1,...,9} is possible in sudoku[i][j], 0 otherwise
// => allows us to quickly determine cells where few values are possible
function possibleFromSudoku(sudoku) {
  let possible = Array.from({ length: 9 }, () =>
    Array.from({ length: 9 }, () => Array(9).fill(1))
  );
  for (let i = 0; i < 9; i++) {
    for (let j = 0; j < 9; j++) {
      let x = sudoku[i][j];
      if (x !== 0) {
        fixValue(i, j, x, possible);
      }
    }
  }
  return possible;
}

function chooseFirstPossible(i, j, possible) {
  let possibleIJ = possible[i][j];
  let x = 0;
  for (let k = 0; k < 9; k++) {
    if (possibleIJ[k] === 1) {
      x = k + 1;
      break;
    }
  }
  return x;
}

function chooseRandomPossible(i, j, possible) {
  let possibleValues = possible[i][j]
    .map((b, m) => b && m + 1)
    .filter((b) => b);
  let n = possibleValues.length;
  if (n === 0) return 0;
  let k = Math.floor(Math.random() * n);
  return possibleValues[k];
}

function fixValue(i, j, x, possible) {
  // mark the value as impossible in the same row
  for (let k = 0; k < 9; k++) {
    if (k === j) {
      possible[i][k][x - 1] = 1;
    } else {
      possible[i][k][x - 1] = 0;
    }
  }
  // mark the value as impossible in the same column
  for (let k = 0; k < 9; k++) {
    if (k === i) {
      possible[k][j][x - 1] = 1;
    } else {
      possible[k][j][x - 1] = 0;
    }
  }
  // mark the value as impossible in the same square
  let [i0, i1] = divmod(i, 3);
  let [j0, j1] = divmod(j, 3);
  for (let k = 0; k < 9; k++) {
    let [ii, jj] = divmod(k, 3);
    if (ii === i1 && jj === j1) {
      possible[3 * i0 + ii][3 * j0 + jj][x - 1] = 1;
    } else {
      possible[3 * i0 + ii][3 * j0 + jj][x - 1] = 0;
    }
  }
  // mark all other values as impossible in the same cell
  for (let k = 0; k < 9; k++) {
    if (k === x - 1) {
      possible[i][j][k] = 1;
    } else {
      possible[i][j][k] = 0;
    }
  }
}

function cellWithFewestPossible(sudoku, possible) {
  let i0, j0;
  let fewest = Infinity;
  for (let i = 0; i < 9; i++) {
    for (let j = 0; j < 9; j++) {
      if (sudoku[i][j] !== 0) continue;
      let possibleIJ = possible[i][j];
      let n = 0;
      for (let k = 0; k < 9; k++) {
        if (possibleIJ[k] === 1) n++;
      }
      if (n < fewest) {
        if (n === 1 || n === 0) return [i, j, n];
        fewest = n;
        [i0, j0] = [i, j];
      }
    }
  }
  return [i0, j0, fewest];
}
```

```
function divmod(k, n) {
  let q = Math.floor(k / n);
  return [q, k - q * n];
}

/**
 * Clones a sudoku.
 *
 * @template T
 * @param {T[]} sudoku
 * @returns {T[]}
 */
function cloneSudoku(sudoku) {
  if (Array.isArray(sudoku[0])) {
    return sudoku.map((x) => cloneSudoku(x));
  } else {
    return [...sudoku];
  }
}
```

</file>

<file>

## path: /src/examples/zkapps/sudoku/sudoku.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/sudoku/sudoku.ts

```
import {
  Field,
  SmartContract,
  method,
  Bool,
  state,
  State,
  isReady,
  Poseidon,
  Struct,
  Circuit,
} from 'o1js';

export { Sudoku, SudokuZkApp };

await isReady;

class Sudoku extends Struct({
  value: Provable.Array(Provable.Array(Field, 9), 9),
}) {
  static from(value: number[][]) {
    return new Sudoku({ value: value.map((row) => row.map(Field)) });
  }

  hash() {
    return Poseidon.hash(this.value.flat());
  }
}

class SudokuZkApp extends SmartContract {
  @state(Field) sudokuHash = State<Field>();
  @state(Bool) isSolved = State<Bool>();

  /**
   * by making this a &nbsp;@method&nbsp;, we ensure that a proof is created for the state initialization.
   * alternatively (and, more efficiently), we could have used &nbsp;super.init()&nbsp; inside
&nbsp;update()&nbsp; below,
   * to ensure the entire state is overwritten.
   * however, it's good to have an example which tests the CLI's ability to handle init() decorated with
&nbsp;@method&nbsp;.
   */
  @method init() {
    super.init();
  }

  @method update(sudokuInstance: Sudoku) {
    this.sudokuHash.set(sudokuInstance.hash());
    this.isSolved.set(Bool(false));
  }

  @method submitSolution(sudokuInstance: Sudoku, solutionInstance: Sudoku) {
    let sudoku = sudokuInstance.value;
    let solution = solutionInstance.value;

    // first, we check that the passed solution is a valid sudoku

    // define helpers
    let range9 = Array.from({ length: 9 }, (_, i) => i);
    let oneTo9 = range9.map((i) => Field(i + 1));

    function assertHas1To9(array: Field[]) {
      oneTo9
        .map((k) => range9.map((i) => array[i].equals(k)).reduce(Bool.or))
        .reduce(Bool.and)
        .assertTrue('array contains the numbers 1...9');
    }

    // check all rows
    for (let i = 0; i < 9; i++) {
      let row = solution[i];
      assertHas1To9(row);
    }
    // check all columns
    for (let j = 0; j < 9; j++) {
```

```
    let column = solution.map((row) => row[j]);
    assertHas1To9(column);
  }
  // check 3x3 squares
  for (let k = 0; k < 9; k++) {
    let [i0, j0] = divmod(k, 3);
    let square = range9.map((m) => {
      let [i1, j1] = divmod(m, 3);
      return solution[3 * i0 + i1][3 * j0 + j1];
    });
    assertHas1To9(square);
  }

  // next, we check that the solution extends the initial sudoku
  for (let i = 0; i < 9; i++) {
    for (let j = 0; j < 9; j++) {
      let cell = sudoku[i][j];
      let solutionCell = solution[i][j];
      // either the sudoku has nothing in it (indicated by a cell value of 0),
      // or it is equal to the solution
      Bool.or(cell.equals(0), cell.equals(solutionCell)).assertTrue(
        &nbsp;solution cell (${i + 1},${j + 1}) matches the original sudoku&nbsp;
      );
    }
  }

  // finally, we check that the sudoku is the one that was originally deployed
  let sudokuHash = this.sudokuHash.get(); // get the hash from the blockchain
  this.sudokuHash.assertEquals(sudokuHash); // precondition that links this.sudokuHash.get() to the actual
on-chain state
  sudokuInstance
    .hash()
    .assertEquals(sudokuHash, 'sudoku matches the one committed on-chain');

  // all checks passed => the sudoku is solved!
  this.isSolved.set(Bool(true));
 }
}

function divmod(k: number, n: number) {
  let q = Math.floor(k / n);
  return [q, k - q * n];
}
```

</file>

<file>

## path: /src/examples/zkapps/token_with_proofs.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/token_with_proofs.ts

```
import {
  isReady,
  method,
  Mina,
  AccountUpdate,
  PrivateKey,
  SmartContract,
  PublicKey,
  UInt64,
  shutdown,
  Int64,
  Experimental,
  Permissions,
  DeployArgs,
  VerificationKey,
  TokenId,
} from 'o1js';

await isReady;

class TokenContract extends SmartContract {
  deploy(args: DeployArgs) {
    super.deploy(args);
    this.setPermissions({
      ...Permissions.default(),
      access: Permissions.proofOrSignature(),
    });
    this.balance.addInPlace(UInt64.from(initialBalance));
  }

  @method tokenDeploy(deployer: PrivateKey, verificationKey: VerificationKey) {
    let address = deployer.toPublicKey();
    let tokenId = this.token.id;
    let deployUpdate = Experimental.createChildAccountUpdate(
      this.self,
      address,
      tokenId
    );
    deployUpdate.account.permissions.set(Permissions.default());
    deployUpdate.account.verificationKey.set(verificationKey);
    deployUpdate.sign(deployer);
  }

  @method mint(receiverAddress: PublicKey) {
    let amount = UInt64.from(1_000_000);
    this.token.mint({ address: receiverAddress, amount });
  }

  @method burn(receiverAddress: PublicKey) {
    let amount = UInt64.from(1_000);
    this.token.burn({ address: receiverAddress, amount });
  }

  @method sendTokens(
    senderAddress: PublicKey,
    receiverAddress: PublicKey,
    callback: Experimental.Callback<any>
  ) {
    let senderAccountUpdate = this.approve(
      callback,
      AccountUpdate.Layout.AnyChildren
    );
    let amount = UInt64.from(1_000);
    let negativeAmount = Int64.fromObject(
      senderAccountUpdate.body.balanceChange
    );
    negativeAmount.assertEquals(Int64.from(amount).neg());
    let tokenId = this.token.id;
    senderAccountUpdate.body.tokenId.assertEquals(tokenId);
    senderAccountUpdate.body.publicKey.assertEquals(senderAddress);
    let receiverAccountUpdate = Experimental.createChildAccountUpdate(
      this.self,
      receiverAddress,
      tokenId
    );
    receiverAccountUpdate.balance.addInPlace(amount);
  }
}

class ZkAppB extends SmartContract {
  @method approveSend() {
    let amount = UInt64.from(1_000);
    this.balance.subInPlace(amount);
  }
}

class ZkAppC extends SmartContract {
  @method approveSend() {
    let amount = UInt64.from(1_000);
    this.balance.subInPlace(amount);
  }
}

let Local = Mina.LocalBlockchain();
Mina.setActiveInstance(Local);

let feePayer = Local.testAccounts[0].privateKey;
let initialBalance = 10_000_000;

let tokenZkAppKey = PrivateKey.random();
let tokenZkAppAddress = tokenZkAppKey.toPublicKey();

let zkAppCKey = PrivateKey.random();
let zkAppCAddress = zkAppCKey.toPublicKey();

let zkAppBKey = PrivateKey.random();
let zkAppBAddress = zkAppBKey.toPublicKey();

let tokenAccount1Key = Local.testAccounts[1].privateKey;
let tokenAccount1 = tokenAccount1Key.toPublicKey();

let tokenZkApp = new TokenContract(tokenZkAppAddress);
let tokenId = tokenZkApp.token.id;

let zkAppB = new ZkAppB(zkAppBAddress, tokenId);
let zkAppC = new ZkAppC(zkAppCAddress, tokenId);
let tx;

console.log('tokenZkAppAddress', tokenZkAppAddress.toBase58());
console.log('zkAppB', zkAppBAddress.toBase58());
console.log('zkAppC', zkAppCAddress.toBase58());
console.log('receiverAddress', tokenAccount1.toBase58());
console.log('feePayer', feePayer.toPublicKey().toBase58());
console.log('-----------------------------------------');

console.log('compile (TokenContract)');
await TokenContract.compile();
console.log('compile (ZkAppB)');
await ZkAppB.compile();
console.log('compile (ZkAppC)');
await ZkAppC.compile();

console.log('deploy tokenZkApp');
tx = await Local.transaction(feePayer, () => {
  AccountUpdate.fundNewAccount(feePayer, { initialBalance });
  tokenZkApp.deploy({ zkappKey: tokenZkAppKey });
});
await tx.send();

console.log('deploy zkAppB');
tx = await Local.transaction(feePayer, () => {
  AccountUpdate.fundNewAccount(feePayer);
  tokenZkApp.tokenDeploy(zkAppBKey, ZkAppB._verificationKey!);
});
console.log('deploy zkAppB (proof)');
await tx.prove();
await tx.send();

console.log('deploy zkAppC');
tx = await Local.transaction(feePayer, () => {
  AccountUpdate.fundNewAccount(feePayer);
  tokenZkApp.tokenDeploy(zkAppCKey, ZkAppC._verificationKey!);
});
console.log('deploy zkAppC (proof)');
await tx.prove();
await tx.send();

console.log('mint token to zkAppB');
tx = await Local.transaction(feePayer, () => {
  tokenZkApp.mint(zkAppBAddress);
});
await tx.prove();
await tx.send();

console.log('approve send from zkAppB');
tx = await Local.transaction(feePayer, () => {
  let approveSendingCallback = Experimental.Callback.create(
    zkAppB,
    'approveSend',
    []
  );
  // we call the token contract with the callback
  tokenZkApp.sendTokens(zkAppBAddress, zkAppCAddress, approveSendingCallback);
});
console.log('approve send (proof)');
await tx.prove();
await tx.send();

console.log(
  &nbsp;zkAppC's balance for tokenId: ${TokenId.toBase58(tokenId)}&nbsp;,
  Mina.getBalance(zkAppCAddress, tokenId).value.toBigInt()
);

console.log('approve send from zkAppC');
tx = await Local.transaction(feePayer, () => {
  // Pay for tokenAccount1's account creation
  AccountUpdate.fundNewAccount(feePayer);
  let approveSendingCallback = Experimental.Callback.create(
    zkAppC,
    'approveSend',
    []
  );
  // we call the token contract with the callback
  tokenZkApp.sendTokens(zkAppCAddress, tokenAccount1, approveSendingCallback);
});
console.log('approve send (proof)');
await tx.prove();
await tx.send();

console.log(
  &nbsp;tokenAccount1's balance for tokenId: ${TokenId.toBase58(tokenId)}&nbsp;,
  Mina.getBalance(tokenAccount1, tokenId).value.toBigInt()
);

shutdown();
```

</file>

<file>

## path: /src/examples/zkapps/voting/demo.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/voting/demo.ts

```
// used to do a dry run, without tests
// ./run ./src/examples/zkapps/voting/demo.ts

import {
  Field,
  Mina,
  AccountUpdate,
  PrivateKey,
  UInt64,
  Reducer,
  Bool,
} from 'o1js';
import { VotingApp, VotingAppParams } from './factory.js';
import { Member, MyMerkleWitness } from './member.js';
import { OffchainStorage } from './off_chain_storage.js';
import {
  ParticipantPreconditions,
  ElectionPreconditions,
} from './preconditions.js';

let Local = Mina.LocalBlockchain({
  proofsEnabled: false,
  enforceTransactionLimits: false,
});
Mina.setActiveInstance(Local);

let feePayer = Local.testAccounts[0].publicKey;
let feePayerKey = Local.testAccounts[0].privateKey;

let tx;

// B62qra25W4URGXxZYqYjfkXBa6SfwwrSjX2ZFJ24x12sSy8khGRcRH1
let voterKey = PrivateKey.fromBase58(
  'EKEgiGWBmGG77ERKU7ihArYbUTfroEr466Gs1RKUph8bgpvF5BSD'
);
// B62qohqUFi8iy5mA4roZDNEuHdj1bWtyriYZouybC33wb8Q6AiUc7D7
let candidateKey = PrivateKey.fromBase58(
  'EKELdqBuWoNa4KFibyumCJNCr1SzMFJi5mV3pCASXfNH3geh6ezG'
);
// B62qq2s61y9gzALPWSAFitucxq1PhLEjQLGwb65gQ7UgsVFNTtjrzRj
let votingKey = PrivateKey.fromBase58(
  'EKFHGpCJTuQk1xHTkQH3q3xXJCHMQLPwhy5iTJk3L2bK4FG9iVnv'
);
```

```
let params: VotingAppParams = {
  candidatePreconditions: new ParticipantPreconditions(
    UInt64.from(100),
    UInt64.from(1000)
  ),
  voterPreconditions: new ParticipantPreconditions(
    UInt64.from(10),
    UInt64.from(200)
  ),
  electionPreconditions: ElectionPreconditions.default,
  voterKey,
  candidateKey,
  votingKey,
  doProofs: true,
};
params.electionPreconditions.enforce = Bool(true);

let contracts = await VotingApp(params);

let voterStore = new OffchainStorage<Member>(3);
let candidateStore = new OffchainStorage<Member>(3);
let votesStore = new OffchainStorage<Member>(3);

let initialRoot = voterStore.getRoot();
tx = await Mina.transaction(feePayer, () => {
  AccountUpdate.fundNewAccount(feePayer, 3);

  contracts.voting.deploy({ zkappKey: votingKey });
  contracts.voting.committedVotes.set(votesStore.getRoot());
  contracts.voting.accumulatedVotes.set(Reducer.initialActionState);

  contracts.candidateContract.deploy({ zkappKey: candidateKey });
  contracts.candidateContract.committedMembers.set(candidateStore.getRoot());
  contracts.candidateContract.accumulatedMembers.set(
    Reducer.initialActionState
  );

  contracts.voterContract.deploy({ zkappKey: voterKey });
  contracts.voterContract.committedMembers.set(voterStore.getRoot());
  contracts.voterContract.accumulatedMembers.set(Reducer.initialActionState);
});
await tx.sign([feePayerKey]).send();

let m: Member = Member.empty();
// lets register three voters
tx = await Mina.transaction(feePayer, () => {
  // creating and registering a new voter
  m = registerMember(
    /*
      NOTE: it isn't wise to use an incremented integer as an
      identifier for real world applications for your entries,
      but instead a public key
      */
    0n,
    Member.from(PrivateKey.random().toPublicKey(), UInt64.from(150)),
    voterStore
  );

  contracts.voting.voterRegistration(m);
  if (!params.doProofs) contracts.voting.sign(votingKey);
});
await tx.prove();
await tx.sign([feePayerKey]).send();

// lets register three voters
tx = await Mina.transaction(feePayer, () => {
  // creating and registering a new voter
  m = registerMember(
    /*
      NOTE: it isn't wise to use an incremented integer as an
      identifier for real world applications for your entries,
      but instead a public key
      */
    1n,
    Member.from(PrivateKey.random().toPublicKey(), UInt64.from(160)),
    voterStore
  );

  contracts.voting.voterRegistration(m);

  if (!params.doProofs) contracts.voting.sign(votingKey);
});
await tx.prove();
await tx.sign([feePayerKey]).send();

// lets register three voters
tx = await Mina.transaction(feePayer, () => {
  // creating and registering a new voter
  m = registerMember(
    /*
      NOTE: it isn't wise to use an incremented integer as an
      identifier for real world applications for your entries,
      but instead a public key
      */
    2n,
    Member.from(PrivateKey.random().toPublicKey(), UInt64.from(170)),
    voterStore
  );

  contracts.voting.voterRegistration(m);

  if (!params.doProofs) contracts.voting.sign(votingKey);
});
await tx.prove();
await tx.sign([feePayerKey]).send();

/*
 since the voting contract calls the voter membership contract via invoking voterRegister,
 the membership contract will then emit one event per new member
 we should have emitted three new members
 */
console.log(
  '3 events?? ',
  contracts.voterContract.reducer.getActions({}).length === 3
);

/*

   Lets register two candidates

 */
tx = await Mina.transaction(feePayer, () => {
  // creating and registering 1 new candidate
  let m = registerMember(
    /*
      NOTE: it isn't wise to use an incremented integer as an
      identifier for real world applications for your entries,
      but instead a public key
      */
    0n,
    Member.from(PrivateKey.random().toPublicKey(), UInt64.from(250)),
    candidateStore
  );

  contracts.voting.candidateRegistration(m);
  if (!params.doProofs) contracts.voting.sign(votingKey);
});

await tx.prove();
await tx.sign([feePayerKey]).send();

tx = await Mina.transaction(feePayer, () => {
  // creating and registering 1 new candidate
  let m = registerMember(
    /*
      NOTE: it isn't wise to use an incremented integer as an
      identifier for real world applications for your entries,
      but instead a public key
      */
    1n,
    Member.from(PrivateKey.random().toPublicKey(), UInt64.from(400)),
    candidateStore
  );

  contracts.voting.candidateRegistration(m);
  if (!params.doProofs) contracts.voting.sign(votingKey);
});

await tx.prove();
await tx.sign([feePayerKey]).send();

/*
  since the voting contact calls the candidate membership contract via invoking candidateRegister,
  the membership contract will then emit one event per new member
  we should have emitted 2 new members, because we registered 2 new candidates
  */
console.log(
  '2 events?? ',
  contracts.candidateContract.reducer.getActions({}).length === 2
);

/*
  we only emitted sequence events,
  so the merkel roots of both membership contract should still be the initial ones
  because the committed state should only change after publish has been invoked
  */

console.log(
  'still initial root? ',
  contracts.candidateContract.committedMembers
    .get()
    .equals(initialRoot)
    .toBoolean()
);
console.log(
  'still initial root? ',
  contracts.voterContract.committedMembers.get().equals(initialRoot).toBoolean()
);

/*
  if we now call approveVoters, which invokes publish on both membership contracts,
  we will also update the committed members!
  and since we keep track of voters and candidates in our off-chain storage,
  both the on-chain committedMembers variable and the off-chain merkle tree root need to be equal
  */

tx = await Mina.transaction(feePayer, () => {
  contracts.voting.approveRegistrations();
  if (!params.doProofs) contracts.voting.sign(votingKey);
});

await tx.prove();
await tx.sign([feePayerKey]).send();

for (let a of candidateStore.values()) {
  console.log(a.publicKey.toBase58());
}
```

```
console.log(
  'candidate root? ',
  contracts.candidateContract.committedMembers
    .get()
    .equals(candidateStore.getRoot())
    .toBoolean()
);
console.log(
  'voter root? ',
  contracts.voterContract.committedMembers
    .get()
    .equals(voterStore.getRoot())
    .toBoolean()
);

/*
    lets vote for the one candidate we have
  */
// we have to up the slot so we are within our election period
Local.incrementGlobalSlot(5);
tx = await Mina.transaction(feePayer, () => {
  let c = candidateStore.get(0n)!;
  c.witness = new MyMerkleWitness(candidateStore.getWitness(0n));
  c.votesWitness = new MyMerkleWitness(votesStore.getWitness(0n));
  // we are voting for candidate c, 0n, with voter 2n
  contracts.voting.vote(c, voterStore.get(2n)!);
  if (!params.doProofs) contracts.voting.sign(votingKey);
});

await tx.prove();
await tx.sign([feePayerKey]).send();
// after the transaction went through, we have to update our off chain store as well
vote(0n);

// vote dispatches a new sequence events, so we should have one

console.log(
  '1 vote sequence event? ',
  contracts.voting.reducer.getActions({}).length === 1
);

/*
    counting the votes
  */
tx = await Mina.transaction(feePayer, () => {
  contracts.voting.countVotes();
  if (!params.doProofs) contracts.voting.sign(votingKey);
});

await tx.prove();
await tx.sign([feePayerKey]).send();
```

```
// vote dispatches a new sequence events, so we should have one

console.log(
  'votes roots equal? ',
  votesStore.getRoot().equals(contracts.voting.committedVotes.get()).toBoolean()
);

printResult();

function registerMember(
  i: bigint,
  m: Member,
  store: OffchainStorage<Member>
): Member {
  Local.addAccount(m.publicKey, m.balance.toString());

  // we will also have to keep track of new voters and candidates within our off-chain merkle tree
  store.set(i, m); // setting voter 0n
  // setting the merkle witness
  m.witness = new MyMerkleWitness(store.getWitness(i));
  return m;
}

function vote(i: bigint) {
  let c_ = votesStore.get(i)!;
  if (!c_) {
    votesStore.set(i, candidateStore.get(i)!);
    c_ = votesStore.get(i)!;
  }
  c_ = c_.addVote();
  votesStore.set(i, c_);
  return c_;
}

function printResult() {
  if (
    !contracts.voting.committedVotes
      .get()
      .equals(votesStore.getRoot())
      .toBoolean()
  ) {
    throw new Error('On-chain root is not up to date with the off-chain tree');
  }

  let result: any = [];
  votesStore.forEach((m, i) => {
    result.push({
      [m.publicKey.toBase58()]: m.votes.toString(),
    });
  });
  console.log(result);
}
```

</file>

<file>

## path: /src/examples/zkapps/voting/deployContracts.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/voting/deployContracts.ts

```
import {
  DeployArgs,
  Field,
  Permissions,
  Mina,
  AccountUpdate,
  PrivateKey,
  SmartContract,
  Reducer,
} from 'o1js';
import { VotingAppParams } from './factory.js';

import { Membership_ } from './membership.js';

import { Voting_ } from './voting.js';

class InvalidContract extends SmartContract {
  deploy(args: DeployArgs) {
    super.deploy(args);
    this.account.permissions.set({
      ...Permissions.default(),
      editState: Permissions.none(),
      editActionState: Permissions.none(),
    });
  }
}

/**
 * Function used to deploy a set of contracts for a given set of preconditions
 * @param feePayer the private key used to pay the fees
 * @param contracts A set of contracts to deploy
 * @param params A set of preconditions and parameters
 * @param voterRoot the initial root of the voter store
 * @param candidateRoot the initial root of the voter store
 * @param votesRoot the initial root of the votes store
 */
export async function deployContracts(
  contracts: {
    voterContract: Membership_;
    candidateContract: Membership_;
    voting: Voting_;
  },
  params: VotingAppParams,
  voterRoot: Field,
  candidateRoot: Field,
  votesRoot: Field,
  proofsEnabled: boolean = false
): Promise<{
  voterContract: Membership_;
  candidateContract: Membership_;
  voting: Voting_;
  Local: any;
  feePayer: PrivateKey;
}> {
  let Local = Mina.LocalBlockchain({
    proofsEnabled,
    enforceTransactionLimits: true,
  });
  Mina.setActiveInstance(Local);

  let feePayerKey = Local.testAccounts[0].privateKey;
  let feePayer = Local.testAccounts[0].publicKey;
  let { voterContract, candidateContract, voting } = contracts;

  console.log('deploying set of 3 contracts');
  let tx = await Mina.transaction(feePayer, () => {
    AccountUpdate.fundNewAccount(feePayer, 3);

    voting.deploy({ zkappKey: params.votingKey });
    voting.committedVotes.set(votesRoot);
    voting.accumulatedVotes.set(Reducer.initialActionState);

    candidateContract.deploy({ zkappKey: params.candidateKey });
    candidateContract.committedMembers.set(candidateRoot);
    candidateContract.accumulatedMembers.set(Reducer.initialActionState);

    voterContract.deploy({ zkappKey: params.voterKey });
    voterContract.committedMembers.set(voterRoot);
    voterContract.accumulatedMembers.set(Reducer.initialActionState);
  });
  await tx.sign([feePayerKey]).send();

  console.log('successfully deployed contracts');
  return {
    voterContract,
    candidateContract,
    voting,
    feePayer: feePayerKey,
    Local,
  };
}

/**
 * Function used to deploy a set of **invalid** membership contracts for a given set of preconditions
 * @param feePayer the private key used to pay the fees
 * @param contracts A set of contracts to deploy
 * @param params A set of preconditions and parameters
 * @param voterRoot the initial root of the voter store
 * @param candidateRoot the initial root of the voter store
 * @param votesRoot the initial root of the votes store
 */
export async function deployInvalidContracts(
  contracts: {
    voterContract: Membership_;
    candidateContract: Membership_;
    voting: Voting_;
  },
  params: VotingAppParams,
  voterRoot: Field,
  candidateRoot: Field,
  votesRoot: Field
): Promise<{
  voterContract: Membership_;
  candidateContract: Membership_;
  voting: Voting_;
  Local: any;
  feePayer: PrivateKey;
}> {
  let Local = Mina.LocalBlockchain({
    proofsEnabled: false,
    enforceTransactionLimits: false,
  });
  Mina.setActiveInstance(Local);

  let feePayerKey = Local.testAccounts[0].privateKey;
  let feePayer = Local.testAccounts[0].publicKey;
  let { voterContract, candidateContract, voting } = contracts;

  console.log('deploying set of 3 contracts');
  let tx = await Mina.transaction(feePayer, () => {
    AccountUpdate.fundNewAccount(feePayer, 3);

    voting.deploy({ zkappKey: params.votingKey });
    voting.committedVotes.set(votesRoot);
    voting.accumulatedVotes.set(Reducer.initialActionState);

    // invalid contracts

    let invalidCandidateContract = new InvalidContract(
      params.candidateKey.toPublicKey()
    );

    invalidCandidateContract.deploy({ zkappKey: params.candidateKey });

    candidateContract = invalidCandidateContract as Membership_;
    let invalidVoterContract = new InvalidContract(
      params.voterKey.toPublicKey()
    );

    invalidVoterContract.deploy({ zkappKey: params.voterKey });

    voterContract = invalidVoterContract as Membership_;
  });
  await tx.sign([feePayerKey]).send();

  console.log('successfully deployed contracts');
  return {
    voterContract,
    candidateContract,
    voting,
    feePayer: feePayerKey,
    Local,
  };
}
```

</file>

<file>

## path: /src/examples/zkapps/voting/dummyContract.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/voting/dummyContract.ts

```
import {
  Field,
  SmartContract,
  state,
  State,
  method,
  DeployArgs,
  Permissions,
} from 'o1js';

export class DummyContract extends SmartContract {
  @state(Field) sum = State<Field>();

  deploy(args: DeployArgs) {
    super.deploy(args);
    this.account.permissions.set({
      ...Permissions.default(),
      editState: Permissions.proofOrSignature(),
      editActionState: Permissions.proofOrSignature(),
      setPermissions: Permissions.proofOrSignature(),
      setVerificationKey: Permissions.proofOrSignature(),
      incrementNonce: Permissions.proofOrSignature(),
    });
    this.sum.set(Field(0));
  }

  /**
   * Method used to add two variables together.
   */
  @method add(x: Field, y: Field) {
    this.sum.set(x.add(y));
  }
}
```

</file>

<file>

## path: /src/examples/zkapps/voting/election_preconditions.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/voting/election_preconditions.ts

```
import { CircuitValue, prop, UInt32 } from 'o1js';

export default class ElectionPreconditions extends CircuitValue {
  @prop startElection: UInt32;
  @prop endElection: UInt32;

  constructor(startElection: UInt32, endElection: UInt32) {
    super();
    this.startElection = startElection;
    this.endElection = endElection;
  }
}
```

</file>

<file>

## path: /src/examples/zkapps/voting/factory.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/voting/factory.ts

```
/**
 * Produces a set of three contracts, Voting, Voter Membership and Candidate Membership SCs.
 * Requires a set of preconditions.
 */

import { PrivateKey } from 'o1js';
import { Membership, Membership_ } from './membership.js';
import {
  ElectionPreconditions,
  ParticipantPreconditions,
} from './preconditions.js';
import { Voting, Voting_ } from './voting.js';

export interface VotingAppParams {
  candidatePreconditions: ParticipantPreconditions;
  voterPreconditions: ParticipantPreconditions;
  electionPreconditions: ElectionPreconditions;
  voterKey: PrivateKey;
  candidateKey: PrivateKey;
  votingKey: PrivateKey;
  doProofs: boolean;
}

function defaultParams(): VotingAppParams {
  return {
    candidatePreconditions: ParticipantPreconditions.default,
    voterPreconditions: ParticipantPreconditions.default,
    electionPreconditions: ElectionPreconditions.default,
    candidateKey: PrivateKey.random(),
    voterKey: PrivateKey.random(),
    votingKey: PrivateKey.random(),
    doProofs: true,
  };
}

/**
 * ! This is the only workaround that I found works with how our contracts compiled
 * ! Maybe we can figure out a more elegant factory pattern for our integration tests
 * This function takes a set of preconditions and produces a set of contract instances.
 * @param params {@link VotingAppParams}
 * @returns
 */
export async function VotingApp(
  params: VotingAppParams = defaultParams()
): Promise<{
  voterContract: Membership_;
  candidateContract: Membership_;
  voting: Voting_;
}> {
  let Voter = await Membership({
    participantPreconditions: params.voterPreconditions,
    contractAddress: params.voterKey.toPublicKey(),
    doProofs: params.doProofs,
  });

  let Candidate = await Membership({
    participantPreconditions: params.candidatePreconditions,
    contractAddress: params.candidateKey.toPublicKey(),
    doProofs: params.doProofs,
  });

  let VotingContract = await Voting({
    electionPreconditions: params.electionPreconditions,
    voterPreconditions: params.voterPreconditions,
    candidatePreconditions: params.candidatePreconditions,
    candidateAddress: params.candidateKey.toPublicKey(),
    voterAddress: params.voterKey.toPublicKey(),
    contractAddress: params.votingKey.toPublicKey(),
    doProofs: params.doProofs,
  });

  return {
    voterContract: Voter,
    candidateContract: Candidate,
    voting: VotingContract,
  };
}
```

</file>

<file>

## path: /src/examples/zkapps/voting/member.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/voting/member.ts

```
import {
  Bool,
  CircuitValue,
  Field,
  prop,
  PublicKey,
  UInt64,
  Poseidon,
  MerkleWitness,
} from 'o1js';

export class MyMerkleWitness extends MerkleWitness(3) {}
let w = {
  isLeft: false,
  sibling: Field(0),
};
let dummyWitness = Array.from(Array(MyMerkleWitness.height - 1).keys()).map(
  () => w
);

export class Member extends CircuitValue {
  @prop publicKey: PublicKey;
  @prop balance: UInt64;

  // will need this to keep track of votes for candidates
  @prop votes: Field;

  @prop witness: MyMerkleWitness;
  @prop votesWitness: MyMerkleWitness;

  constructor(publicKey: PublicKey, balance: UInt64) {
    super();
    this.publicKey = publicKey;
    this.balance = balance;
    this.votes = Field(0);

    this.witness = new MyMerkleWitness(dummyWitness);
    this.votesWitness = new MyMerkleWitness(dummyWitness);
  }

  getHash(): Field {
    return Poseidon.hash(
      this.publicKey
        .toFields()
        .concat(this.balance.toFields())
        .concat(this.votes.toFields())
    );
  }

  addVote(): Member {
    this.votes = this.votes.add(1);
    return this;
  }

  static empty() {
    return new Member(PublicKey.empty(), UInt64.zero);
  }

  static from(publicKey: PublicKey, balance: UInt64) {
    return new Member(publicKey, balance);
  }
}
```

</file>

<file>

## path: /src/examples/zkapps/voting/membership.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/voting/membership.ts

```
import {
  Field,
  SmartContract,
  state,
  State,
  method,
  DeployArgs,
  Permissions,
  Bool,
  PublicKey,
  Reducer,
  provablePure,
  AccountUpdate,
  Provable,
} from 'o1js';
import { Member } from './member.js';
import { ParticipantPreconditions } from './preconditions.js';

let participantPreconditions = ParticipantPreconditions.default;

interface MembershipParams {
  participantPreconditions: ParticipantPreconditions;
  contractAddress: PublicKey;
  doProofs: boolean;
}

/**
```

```
 * Returns a new contract instance that based on a set of preconditions.
 * @param params {@link MembershipParams}
 */
export async function Membership(
  params: MembershipParams
): Promise<Membership_> {
  participantPreconditions = params.participantPreconditions;

  let contract = new Membership_(params.contractAddress);

  params.doProofs = true;
  if (params.doProofs) {
    await Membership_.compile();
  }

  return contract;
}

/**
 * The Membership contract keeps track of a set of members.
 * The contract can either be of type Voter or Candidate.
 */
export class Membership_ extends SmartContract {
  /**
   * Root of the merkle tree that stores all committed members.
   */
  @state(Field) committedMembers = State<Field>();

  /**
   * Accumulator of all emitted members.
   */
  @state(Field) accumulatedMembers = State<Field>();

  reducer = Reducer({ actionType: Member });

  events = {
    newMemberState: provablePure({
      accumulatedMembersRoot: Field,
      committedMembersRoot: Field,
    }),
  };

  deploy(args: DeployArgs) {
    super.deploy(args);
    this.account.permissions.set({
      ...Permissions.default(),
      editState: Permissions.proofOrSignature(),
      editActionState: Permissions.proofOrSignature(),
      setPermissions: Permissions.proofOrSignature(),
      setVerificationKey: Permissions.proofOrSignature(),
      incrementNonce: Permissions.proofOrSignature(),
    });
```

```
}

/**
 * Method used to add a new member.
 * Dispatches a new member sequence event.
 * @param member
 */
@method addEntry(member: Member): Bool {
  // Emit event that indicates adding this item
  // Preconditions: Restrict who can vote or who can be a candidate

  // since we need to keep this contract "generic", we always assert within a range
  // even tho voters cant have a maximum balance, only candidates
  // but for a voter we simply use UInt64.MAXINT() as the maximum

  let accountUpdate = AccountUpdate.create(member.publicKey);

  accountUpdate.account.balance.assertEquals(
    accountUpdate.account.balance.get()
  );

  let balance = accountUpdate.account.balance.get();

  balance.assertGreaterThanOrEqual(
    participantPreconditions.minMina,
    'Balance not high enough!'
  );
  balance.assertLessThanOrEqual(
    participantPreconditions.maxMina,
    'Balance too high!'
  );

  let accumulatedMembers = this.accumulatedMembers.get();
  this.accumulatedMembers.assertEquals(accumulatedMembers);

  // checking if the member already exists within the accumulator
  let { state: exists } = this.reducer.reduce(
    this.reducer.getActions({
      fromActionState: accumulatedMembers,
    }),
    Bool,
    (state: Bool, action: Member) => {
      return action.equals(member).or(state);
    },
    // initial state
    { state: Bool(false), actionState: accumulatedMembers }
  );

  /*
  we cant really branch the control flow - we will always have to emit an event no matter what,
  so we emit an empty event if the member already exists
  it the member doesn't exist, emit the "real" member
```

```
    */

    let toEmit = Provable.if(exists, Member.empty(), member);

    this.reducer.dispatch(toEmit);

    return exists;
  }

  /**
   * Method used to check whether a member exists within the committed storage.
   * @param accountId
   * @returns true if member exists
   */
  @method isMember(member: Member): Bool {
    // Verify membership (voter or candidate) with the accountId via merkle tree committed to by the
sequence events and returns a boolean
    // Preconditions: Item exists in committed storage

    let committedMembers = this.committedMembers.get();
    this.committedMembers.assertEquals(committedMembers);

    return member.witness
      .calculateRootSlow(member.getHash())
      .equals(committedMembers);
  }

  /**
   * Method used to commit to the accumulated list of members.
   */
  @method publish() {
    // Commit to the items accumulated so far. This is a periodic update

    let accumulatedMembers = this.accumulatedMembers.get();
    this.accumulatedMembers.assertEquals(accumulatedMembers);

    let committedMembers = this.committedMembers.get();
    this.committedMembers.assertEquals(committedMembers);

    let pendingActions = this.reducer.getActions({
      fromActionState: accumulatedMembers,
    });

    let { state: newCommittedMembers, actionState: newAccumulatedMembers } =
      this.reducer.reduce(
        pendingActions,
        Field,
        (state: Field, action: Member) => {
          // because we inserted empty members, we need to check if a member is empty or "real"
          let isRealMember = Provable.if(
            action.publicKey.equals(PublicKey.empty()),
            Bool(false),
            Bool(true)
          );

          // if the member is real and not empty, we calculate and return the new merkle root
          // otherwise, we simply return the unmodified state - this is our way of branching
          return Provable.if(
            isRealMember,
            action.witness.calculateRootSlow(action.getHash()),
            state
          );
        },
        // initial state
        { state: committedMembers, actionState: accumulatedMembers },
        { maxTransactionsWithActions: 2 }
      );

    this.committedMembers.set(newCommittedMembers);
    this.accumulatedMembers.set(newAccumulatedMembers);

    this.emitEvent('newMemberState', {
      committedMembersRoot: newCommittedMembers,
      accumulatedMembersRoot: newAccumulatedMembers,
    });
  }
}
```

</file>

<file>

## path: /src/examples/zkapps/voting/off_chain_storage.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/voting/off_chain_storage.ts

```
// Merkle Tree and off chain storage

import { Field, MerkleTree } from 'o1js';

export { OffchainStorage };

class OffchainStorage<
  V extends {
    getHash(): Field;
  }
> extends Map<bigint, V> {
  private merkleTree;

  constructor(public readonly height: number) {
    super();
    this.merkleTree = new MerkleTree(height);
  }

  set(key: bigint, value: V): this {
    super.set(key, value);
    this.merkleTree.setLeaf(key, value.getHash());
    return this;
  }

  get(key: bigint): V | undefined {
    return super.get(key);
  }

  getWitness(key: bigint): { isLeft: boolean; sibling: Field }[] {
    return this.merkleTree.getWitness(key);
  }

  getRoot(): Field {
    return this.merkleTree.getRoot();
  }
}
```

</file>

<file>

## path: /src/examples/zkapps/voting/participant_preconditions.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/voting/participant_preconditions.ts

```
import { CircuitValue, prop, UInt64 } from 'o1js';

export default class ParticipantPreconditions extends CircuitValue {
  @prop minMinaVote: UInt64;
  @prop minMinaCandidate: UInt64;
  @prop maxMinaCandidate: UInt64;

  constructor(
    minMinaVote: UInt64,
    minMinaCandidate: UInt64,
    maxMinaCandidate: UInt64
  ) {
    super();
    this.minMinaVote = minMinaVote;
    this.minMinaCandidate = minMinaCandidate;
    this.maxMinaCandidate = maxMinaCandidate;
  }
}
```

</file>

<file>

## path: /src/examples/zkapps/voting/preconditions.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/voting/preconditions.ts

```
import { Bool, UInt32, UInt64 } from 'o1js';

export class ElectionPreconditions {
  startElection: UInt32;
  endElection: UInt32;
  enforce: Bool;
  static get default(): ElectionPreconditions {
    return new ElectionPreconditions(UInt32.zero, UInt32.MAXINT(), Bool(false));
  }

  constructor(startElection: UInt32, endElection: UInt32, enforce: Bool) {
    this.startElection = startElection;
    this.endElection = endElection;
    this.enforce = enforce;
  }
}

export class ParticipantPreconditions {
  minMina: UInt64;
  maxMina: UInt64; // have to make this "generic" so it applys for both candidate and voter instances

  static get default(): ParticipantPreconditions {
    return new ParticipantPreconditions(UInt64.zero, UInt64.MAXINT());
  }

  constructor(minMina: UInt64, maxMina: UInt64) {
    this.minMina = minMina;
    this.maxMina = maxMina;
  }
}
```

</file>

<file>

## path: /src/examples/zkapps/voting/run.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/voting/run.ts

```
import { Bool, PrivateKey, UInt32, UInt64 } from 'o1js';
import { VotingApp, VotingAppParams } from './factory.js';
import {
  ElectionPreconditions,
  ParticipantPreconditions,
} from './preconditions.js';

import { OffchainStorage } from './off_chain_storage.js';
import { Member } from './member.js';
import { testSet } from './test.js';
import { getProfiler } from '../../utils/profiler.js';
```

```
console.log('Running Voting script...');

// I really hope this factory pattern works with o1js' contracts
// one voting instance always consists of three contracts: two membership contracts and one voting contract
// this pattern will hopefully help us deploy multiple sets of voting apps
// with different preconditions efficiently for integration tests
// ! the VotingApp() factory returns a set of compiled contract instances

// dummy set to demonstrate how the script will function
console.log('Starting set 1...');

let params_set1: VotingAppParams = {
  candidatePreconditions: new ParticipantPreconditions(
    UInt64.from(10),
    UInt64.from(5000)
  ),
  voterPreconditions: new ParticipantPreconditions(
    UInt64.from(10),
    UInt64.from(50)
  ),
  electionPreconditions: new ElectionPreconditions(
    UInt32.from(5),
    UInt32.from(15),
    Bool(true)
  ),
  voterKey: PrivateKey.random(),
  candidateKey: PrivateKey.random(),
  votingKey: PrivateKey.random(),
  doProofs: false,
};

let storage_set1 = {
  votesStore: new OffchainStorage<Member>(3),
  candidatesStore: new OffchainStorage<Member>(3),
  votersStore: new OffchainStorage<Member>(3),
};

console.log('Building contracts for set 1...');
let contracts_set1 = await VotingApp(params_set1);

console.log('Testing set 1...');
const VotingProfiler = getProfiler('Voting profiler set 1');
VotingProfiler.start('Voting test flow');
await testSet(contracts_set1, params_set1, storage_set1);
VotingProfiler.stop().store();
// ..
```

</file>

<file>

## path: /src/examples/zkapps/voting/run_berkeley.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/voting/run_berkeley.ts

```
import {
  AccountUpdate,
  Bool,
  fetchAccount,
  isReady,
  Mina,
  PrivateKey,
  PublicKey,
  Reducer,
  shutdown,
  SmartContract,
  UInt32,
  UInt64,
} from 'o1js';
import { VotingApp, VotingAppParams } from './factory.js';
import { Member, MyMerkleWitness } from './member.js';
import { OffchainStorage } from './off_chain_storage.js';
import {
  ParticipantPreconditions,
  ElectionPreconditions,
} from './preconditions.js';
import { getResults, vote } from './voting_lib.js';
await isReady;

const Berkeley = Mina.Network({
  mina: 'https://proxy.berkeley.minaexplorer.com/graphql',
  archive: 'https://archive-node-api.p42.xyz/',
});
Mina.setActiveInstance(Berkeley);

let feePayerKey = PrivateKey.random();
let feePayerAddress = feePayerKey.toPublicKey();

let voterKey = PrivateKey.random();
let candidateKey = PrivateKey.random();
let votingKey = PrivateKey.random();

console.log('waiting for accounts to receive funds...');
await Mina.faucet(feePayerAddress);
console.log('funds received');

console.log(&nbsp;using the following addressed:
feePayer: ${feePayerAddress.toBase58()}
voting manager contract: ${votingKey.toPublicKey().toBase58()}
candidate membership contract: ${candidateKey.toPublicKey().toBase58()}
voter membership contract: ${voterKey.toPublicKey().toBase58()}&nbsp;);
```

```
let params: VotingAppParams = {
  candidatePreconditions: new ParticipantPreconditions(
    UInt64.from(0),
    UInt64.from(100_000_000_000)
  ),
  voterPreconditions: new ParticipantPreconditions(
    UInt64.from(0),
    UInt64.from(100_000_000_000)
  ),
  electionPreconditions: new ElectionPreconditions(
    UInt32.from(0),
    UInt32.MAXINT(),
    Bool(false)
  ),
  voterKey,
  candidateKey,
  votingKey,
  doProofs: false,
};

// we are using pre-funded voters here
const members = [
  PublicKey.fromBase58(
    'B62qqzhd5U54JafhR4CB8NLWQM8PRfiCZ4TuoTT5UQHzGwdR2f5RLnK'
  ),
  PublicKey.fromBase58(
    'B62qnScMYfgSUWwtzB1r6fB8i23YFXgA25rzcSXVCtYVfUxLHkMLr3G'
  ),
];

let storage = {
  votesStore: new OffchainStorage<Member>(3),
  candidatesStore: new OffchainStorage<Member>(3),
  votersStore: new OffchainStorage<Member>(3),
};

console.log('building contracts');
let contracts = await VotingApp(params);

console.log('deploying set of 3 contracts');
let tx = await Mina.transaction(
  {
    sender: feePayerAddress,
    fee: 10_000_000,
    memo: 'Deploying contracts',
  },
  () => {
    AccountUpdate.fundNewAccount(feePayerAddress, 3);

    contracts.voting.deploy({ zkappKey: params.votingKey });
    contracts.voting.committedVotes.set(storage.votesStore.getRoot());
    contracts.voting.accumulatedVotes.set(Reducer.initialActionState);
    contracts.candidateContract.deploy({ zkappKey: params.candidateKey });
    contracts.candidateContract.committedMembers.set(
      storage.candidatesStore.getRoot()
    );
    contracts.candidateContract.accumulatedMembers.set(
      Reducer.initialActionState
    );

    contracts.voterContract.deploy({ zkappKey: params.voterKey });
    contracts.voterContract.committedMembers.set(storage.votersStore.getRoot());
    contracts.voterContract.accumulatedMembers.set(Reducer.initialActionState);
  }
);
await tx.prove();
await (await tx.sign([feePayerKey]).send()).wait();

console.log('successfully deployed contracts');

await fetchAllAccounts();

console.log('registering one voter');
tx = await Mina.transaction(
  {
    sender: feePayerAddress,
    fee: 10_000_000,
    memo: 'Registering a voter',
  },
  () => {
    let m = registerMember(
      0n,
      Member.from(members[0], UInt64.from(150)),
      storage.votersStore
    );
    contracts.voting.voterRegistration(m);
  }
);
await tx.prove();
await (await tx.sign([feePayerKey]).send()).wait();
console.log('voter registered');

await fetchAllAccounts();

console.log('registering one candidate');
tx = await Mina.transaction(
  {
    sender: feePayerAddress,
    fee: 10_000_000,
    memo: 'Registering a candidate',
  },
  () => {
    let m = registerMember(
    0n,
    Member.from(members[1], UInt64.from(150)),
    storage.candidatesStore
  );
  contracts.voting.candidateRegistration(m);
 }
);
await tx.prove();
await (await tx.sign([feePayerKey]).send()).wait();
console.log('candidate registered');
// we have to wait a few seconds before continuing, otherwise we might not get the actions from the archive,
we if continue too fast
await new Promise((resolve) => setTimeout(resolve, 20000));

await fetchAllAccounts();

console.log('approving registrations');
tx = await Mina.transaction(
 {
  sender: feePayerAddress,
  fee: 10_000_000,
  memo: 'Approving registrations',
 },
 () => {
  contracts.voting.approveRegistrations();
 }
);
await tx.prove();
await (await tx.sign([feePayerKey]).send()).wait();
console.log('registrations approved');

await fetchAllAccounts();

console.log('voting for a candidate');
tx = await Mina.transaction(
 { sender: feePayerAddress, fee: 10_000_000, memo: 'Casting vote' },
 () => {
  let currentCandidate = storage.candidatesStore.get(0n)!;

  currentCandidate.witness = new MyMerkleWitness(
    storage.candidatesStore.getWitness(0n)
  );
  currentCandidate.votesWitness = new MyMerkleWitness(
    storage.votesStore.getWitness(0n)
  );

  let v = storage.votersStore.get(0n)!;
  v.witness = new MyMerkleWitness(storage.votersStore.getWitness(0n));
  console.log(v.witness.calculateRoot(v.getHash()).toString());
  console.log(contracts.voting.committedVotes.get().toString());
  contracts.voting.vote(currentCandidate, v);
 }
```

```
);
await tx.prove();
await (await tx.sign([feePayerKey]).send()).wait();
vote(0n, storage.votesStore, storage.candidatesStore);
console.log('voted for a candidate');
await new Promise((resolve) => setTimeout(resolve, 20000));

await fetchAllAccounts();

console.log('counting votes');
tx = await Mina.transaction(
  { sender: feePayerAddress, fee: 10_000_000, memo: 'Counting votes' },
  () => {
    contracts.voting.countVotes();
  }
);
await tx.prove();
await (await tx.sign([feePayerKey]).send()).wait();
console.log('votes counted');
await new Promise((resolve) => setTimeout(resolve, 20000));

await fetchAllAccounts();
let results = getResults(contracts.voting, storage.votesStore);

if (results[members[1].toBase58()] !== 1) {
  throw Error(
    &nbsp;Candidate ${members[1].toBase58()} should have one vote, but has ${
      results[members[1].toBase58()]
    } &nbsp;
  );
}
console.log('final result', results);

console.log('The following events were emitted during the voting process:');

await displayEvents(contracts.voting);
await displayEvents(contracts.candidateContract);
await displayEvents(contracts.voterContract);

async function displayEvents(contract: SmartContract) {
  let events = await contract.fetchEvents();
  console.log(
    &nbsp;events on ${contract.address.toBase58()}&nbsp;,
    events.map((e) => {
      return { type: e.type, data: JSON.stringify(e.event) };
    })
  );
}

async function fetchAllAccounts() {
  await Promise.all(
    [
    feePayerAddress,
    params.voterKey.toPublicKey(),
    params.candidateKey.toPublicKey(),
    params.votingKey.toPublicKey(),
    ...members,
  ].map((publicKey) => fetchAccount({ publicKey }))
 );
}

function registerMember(
 i: bigint,
 m: Member,
 store: OffchainStorage<Member>
): Member {
 // we will also have to keep track of new voters and candidates within our off-chain merkle tree
 store.set(i, m); // setting voter 0n
 // setting the merkle witness
 m.witness = new MyMerkleWitness(store.getWitness(i));
 return m;
}

shutdown();
```

</file>

<file>

## path: /src/examples/zkapps/voting/test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/voting/test.ts

```
import {
 Mina,
 AccountUpdate,
 Field,
 PrivateKey,
 UInt64,
 UInt32,
 Permissions,
} from 'o1js';
import { deployContracts, deployInvalidContracts } from './deployContracts.js';
import { DummyContract } from './dummyContract.js';
import { VotingAppParams } from './factory.js';
import { Member, MyMerkleWitness } from './member.js';
import { Membership_ } from './membership.js';
import { OffchainStorage } from './off_chain_storage.js';
import { Voting_ } from './voting.js';
import {
 assertValidTx,
 getResults,
 registerMember,
  vote,
} from './voting_lib.js';

type Votes = OffchainStorage<Member>;
type Candidates = OffchainStorage<Member>;
type Voters = OffchainStorage<Member>;

/**
 * Function used to test a set of contracts and precondition
 * @param contracts A set of contracts
 * @param params A set of preconditions and parameters
 * @param storage A set of off-chain storage
 */
export async function testSet(
  contracts: {
    voterContract: Membership_;
    candidateContract: Membership_;
    voting: Voting_;
  },
  params: VotingAppParams,
  storage: {
    votesStore: Votes;
    candidatesStore: Candidates;
    votersStore: Voters;
  }
) {
  let { votersStore, candidatesStore, votesStore } = storage;

  // toggle these to only run a subset for debugging
  let runTestingPhases = { 1: true, 2: true, 3: true, 4: true, 5: true };

  if (runTestingPhases[1]) {
    /*
    test case description:
      change verification key of a deployed zkapp

    preconditions:
      - contracts are deployed and valid
      - verification key changes

    tested cases:
      - deploy contract and make sure they are valid
      - change verification key
      - proofs should fail since verification key is outdated

    expected results:
      - transaction fails if verification key does not match the proof

  */
    console.log('deploying testing phase 1 contracts');

    let verificationKeySet = await deployContracts(
      contracts,
      params,
      Field(0),
      Field(0),
      Field(0),
      true
    );
    console.log('checking that the tx is valid using default verification key');

    let m = Member.from(PrivateKey.random().toPublicKey(), UInt64.from(15));
    verificationKeySet.Local.addAccount(m.publicKey, m.balance.toString());

    await assertValidTx(
      true,
      () => {
        verificationKeySet.voting.voterRegistration(m);
      },
      verificationKeySet.feePayer
    );

    console.log('changing verification key');
    let { verificationKey } = await DummyContract.compile();

    await assertValidTx(
      true,
      () => {
        let vkUpdate = AccountUpdate.createSigned(params.votingKey);
        vkUpdate.account.verificationKey.set({
          ...verificationKey,
          hash: Field(verificationKey.hash),
        });
      },
      verificationKeySet.feePayer
    );

    m = Member.from(PrivateKey.random().toPublicKey(), UInt64.from(15));
    verificationKeySet.Local.addAccount(m.publicKey, m.balance.toString());

    await assertValidTx(
      false,
      () => {
        verificationKeySet.voting.voterRegistration(m);
      },
      verificationKeySet.feePayer,
      'Invalid proof'
    );
  }

  if (runTestingPhases[2]) {
    /*
    test case description:
      permissions of the zkapp account change in the middle of a transaction
    preconditions:
      - set of voting contracts deployed
      - permissions allow the transaction to pass
      - trying to register a valid member
      - changing permissions mid-transaction

    tested cases:
      - making sure a transaction passed with default permissions -> tx success
      - changing the permissions to disallow the transaction to pass -> tx failure
      - changing permissions back to default that allows the transaction to pass -> tx success
      - changing permissions back to default on its own -> tx success
      - invoking a method on its own -> success

    expected results:
      - transaction fails or succeeds, depending on the ordering of permissions changes

  */
    console.log('deploying testing phase 2 contracts');

    let permissionedSet = await deployContracts(
      contracts,
      params,
      Field(0),
      Field(0),
      Field(0)
    );
    console.log('checking that the tx is valid using default permissions');

    let m = Member.from(PrivateKey.random().toPublicKey(), UInt64.from(15));
    permissionedSet.Local.addAccount(m.publicKey, m.balance.toString());

    await assertValidTx(
      true,
      () => {
        permissionedSet.voting.voterRegistration(m);
      },
      permissionedSet.feePayer
    );

    console.log('trying to change permissions...');

    await assertValidTx(
      true,
      () => {
        let permUpdate = AccountUpdate.createSigned(params.voterKey);

        permUpdate.account.permissions.set({
          ...Permissions.default(),
          setPermissions: Permissions.none(),
          editActionState: Permissions.impossible(),
        });
      },
      permissionedSet.feePayer
    );

    console.log('trying to invoke method with invalid permissions...');

    m = Member.from(PrivateKey.random().toPublicKey(), UInt64.from(15));
    permissionedSet.Local.addAccount(m.publicKey, m.balance.toString());

    await assertValidTx(
      false,
      () => {
        permissionedSet.voting.voterRegistration(m);
      },
      permissionedSet.feePayer,
      'actions'
    );
  }

  if (runTestingPhases[3]) {
    /*
    test case description:
      voting contract is trying to call methods of an invalid contract

    preconditions:
      - real voting contract deployed
      - voter and candidate membership contracts are faulty (empty/dummy contracts)
      - trying to register a valid voter member

    tested cases:
      - deploying set of invalid contracts
        - trying to invoke a non-existent method -> failure

    expected results:
      - throws an error

  */

    console.log('deploying testing phase 3 contracts');

    let invalidSet = await deployInvalidContracts(
      contracts,
      params,
      votersStore.getRoot(),
      candidatesStore.getRoot(),
      votesStore.getRoot()
    );

    console.log('trying to invoke invalid contract method...');

    let m = Member.from(PrivateKey.random().toPublicKey(), UInt64.from(15));
    invalidSet.Local.addAccount(m.publicKey, m.balance.toString());
    try {
      let tx = await Mina.transaction(invalidSet.feePayer.toPublicKey(), () => {
        invalidSet.voting.voterRegistration(m);
      });
      await tx.prove();
      await tx.sign([invalidSet.feePayer]).send();
    } catch (err: any) {
      if (!err.toString().includes('fromActionState not found')) {
        throw Error(
          &nbsp;Transaction should have failed, but failed with an unexpected error! ${err}&nbsp;
        );
      }
    }
  }

  const initialRoot = votersStore.getRoot();

  if (runTestingPhases[4]) {
    console.log('deploying testing phase 4 contracts');

    let sequenceOverflowSet = await deployContracts(
      contracts,
      params,
      votersStore.getRoot(),
      candidatesStore.getRoot(),
      votesStore.getRoot()
    );

    /*
    test case description:
      overflowing maximum amount of sequence events allowed in the reducer (2)

    preconditions:
      - x

    tested cases:
      - emitted 3 sequence events and trying to reduce them

    expected results:
      - throws an error

  */

    console.log('trying to overflow actions (custom max: 2)');

    console.log(
      'emitting more than 2 actions without periodically updating them'
    );
    for (let index = 0; index <= 3; index++) {
      try {
        let tx = await Mina.transaction(
          sequenceOverflowSet.feePayer.toPublicKey(),
          () => {
            let m = Member.from(
              PrivateKey.random().toPublicKey(),

              UInt64.from(15)
            );
            sequenceOverflowSet.Local.addAccount(
              m.publicKey,
              m.balance.toString()
            );

            sequenceOverflowSet.voting.voterRegistration(m);
          }
        );
        await tx.prove();
        await tx.sign([sequenceOverflowSet.feePayer]).send();
      } catch (error) {
        throw new Error('Transaction failed!');
      }
    }

    if (sequenceOverflowSet.voterContract.reducer.getActions().length < 3) {
      throw Error(
        &nbsp;Did not emit expected actions! Only emitted ${
          sequenceOverflowSet.voterContract.reducer.getActions().length
        }&nbsp;
      );
    }

    try {
      let tx = await Mina.transaction(
        sequenceOverflowSet.feePayer.toPublicKey(),
        () => {
          sequenceOverflowSet.voting.approveRegistrations();
        }
      );
      await tx.prove();
      await tx.sign([sequenceOverflowSet.feePayer]).send();
    } catch (err: any) {
      if (!err.toString().includes('the maximum number of lists of actions')) {
        throw Error(
          &nbsp;Transaction should have failed but went through! Error: ${err}&nbsp;
        );
      }
    }
  }

  if (runTestingPhases[5]) {
    console.log('deploying testing phase 5 contracts');

    let { voterContract, candidateContract, voting, feePayer, Local } =
      await deployContracts(
        contracts,
        params,
        votersStore.getRoot(),
        candidatesStore.getRoot(),
        votesStore.getRoot()
      );

    /*
    test case description:
      Happy path - invokes addEntry on voter membership SC

    preconditions:
      - no such member exists within the accumulator
      - the member passed in is a valid voter that passes the required preconditions
      - time window is before election has started

    tested cases:
      - voter is valid and can be registered

    expected results:
      - no state change at all
      - voter SC emits one sequence event
      - -> invoked addEntry method on voter SC

  */

    let initialAccumulatedMembers = voterContract.accumulatedMembers.get();
    let initialCommittedMembers = voterContract.committedMembers.get();

    console.log(
      &nbsp;setting slot to ${params.electionPreconditions.startElection
        .sub(1)
        .toString()}, before election has started&nbsp;
    );
    Local.setGlobalSlot(
      UInt32.from(params.electionPreconditions.startElection.sub(1))
    );

    console.log('attempting to register a valid voter... ');

    // register new member
    let newVoter1 = registerMember(
      0n,
      Member.from(PrivateKey.random().toPublicKey(), UInt64.from(15)),
      votersStore,
      Local
    );

    await assertValidTx(
      true,
      () => {
       voting.voterRegistration(newVoter1);
    },
    feePayer
);

if (voterContract.reducer.getActions().length !== 1) {
  throw Error(
    'Should have emitted 1 event after registering only one valid voter'
  );
}

if (
  !initialAccumulatedMembers
    .equals(voterContract.accumulatedMembers.get())
    .toBoolean() ||
  !initialCommittedMembers
    .equals(voterContract.committedMembers.get())
    .toBoolean()
) {
  throw Error('State changed, but should not have!');
}

/*
test case description:
  checking the methods failure, depending on different predefined preconditions
  (voterPreconditions - minimum balance and maximum balance)

preconditions:
  - voter has not enough balance
  - voter has a too high balance
  - voter already exists within the sequence state
  - .. ??

tested cases:
  - voter has not enough balance -> failure
  - voter has too high balance -> failure
  - voter registered twice -> failure


expected results:
  - no state change at all
  - voter SC emits one sequence event
*/
function addAccount(member: Member) {
  Local.addAccount(member.publicKey, member.balance.toString());
}
console.log('attempting to register a voter with not enough balance...');
let newVoterLow = Member.from(
  PrivateKey.random().toPublicKey(),
  params.voterPreconditions.minMina.sub(1)
);
addAccount(newVoterLow);
```

```
await assertValidTx(
  false,
  () => {
    voting.voterRegistration(newVoterLow);
  },
  feePayer,
  'Balance not high enough!'
);

console.log('attempting to register a voter with too high balance...');
let newVoterHigh = Member.from(
  PrivateKey.random().toPublicKey(),
  params.voterPreconditions.maxMina.add(1)
);
addAccount(newVoterHigh);

await assertValidTx(
  false,
  () => {
    voting.voterRegistration(newVoterHigh);
  },
  feePayer,
  'Balance too high!'
);

console.log('attempting to register the same voter twice...');

await assertValidTx(
  false,
  () => {
    voting.voterRegistration(newVoter1);
  },
  feePayer,
  'Member already exists!'
);

if (voterContract.reducer.getActions().length !== 1) {
  throw Error(
    'Should have emitted 1 event after registering only one valid voter'
  );
}

/*

test case description:
  Happy path - invokes addEntry on candidate membership SC
  (similar to voter contract)

preconditions:
  - no such member exists within the accumulator
  - the member passed in is a valid candidate that passes the required preconditions
```

```
    - time window is before election has started

  tested cases:
    - candidate is valid and can be registered -> success

  expected results:
    - no state change at all
    - voter SC emits one sequence event
    - -> invoked addEntry method on voter SC
*/
  console.log('attempting to register a candidate...');

  await assertValidTx(
    true,
    () => {
      let newCandidate = registerMember(
        0n,
        Member.from(
          PrivateKey.random().toPublicKey(),

          params.candidatePreconditions.minMina.add(1)
        ),
        candidatesStore,
        Local
      );

      // register new candidate
      voting.candidateRegistration(newCandidate);
    },
    feePayer
  );

  console.log('attempting to register another candidate...');

  await assertValidTx(
    true,
    () => {
      let newCandidate = registerMember(
        1n,
        Member.from(
          PrivateKey.random().toPublicKey(),

          params.candidatePreconditions.minMina.add(1)
        ),
        candidatesStore,
        Local
      );

      // register new candidate
      voting.candidateRegistration(newCandidate);
    },
    feePayer
  );

  let numberOfEvents = candidateContract.reducer.getActions().length;
  if (candidateContract.reducer.getActions().length !== 2) {
    throw Error(
      &nbsp;Should have emitted 2 event after registering 2 candidates. ${numberOfEvents} emitted&nbsp;
    );
  }

  // the merkle roots of both membership contract should still be the initial ones because publish hasn't been
invoked
  // therefor the state should not have changes
  if (
    !candidateContract.committedMembers.get().equals(initialRoot).toBoolean()
  ) {
    throw Error('candidate merkle root is not the initialroot');
  }

  if (!voterContract.committedMembers.get().equals(initialRoot).toBoolean()) {
    throw Error('voter merkle root is not the initialroot');
  }

  /*
  test case description:
    approve registrations, invoked publish on both membership SCs

  preconditions:
    - votes and candidates were registered previously

  tested cases:
    - authorizing all members -> success

  expected results:
    - publish invoked
    - sequence events executed and committed state updates on both membership contracts
      - committed state should now equal off-chain state
    - voting contract state unchanged
 */
  console.log('authorizing registrations...');

  await assertValidTx(
    true,
    () => {
      // register new candidate
      voting.approveRegistrations();
    },
    feePayer
  );

  // approve updates the committed members on both contracts by invoking the publish method.
  // We check if offchain storage merkle roots match both on-chain committedMembers for voters and
candidates
```

```
if (!voting.committedVotes.get().equals(initialRoot).toBoolean()) {
  throw Error('voter contract state changed, but should not have');
}

if (
  !candidateContract.committedMembers
    .get()
    .equals(candidatesStore.getRoot())
    .toBoolean()
) {
  throw Error(
    'candidatesStore merkle root does not match on-chain committed members'
  );
}

if (
  !voterContract.committedMembers
    .get()
    .equals(votersStore.getRoot())
    .toBoolean()
) {
  throw Error(
    'votersStore merkle root does not match on-chain committed members'
  );
}

/*
test case description:
  registering candidate within the election period

preconditions:
  - slot has been set to within the election period

tested cases:
  - registering candidate -> failure
  - registering voter -> failure

expected results:
  - no new events emitted
  - no state changes
*/

console.log(
  'attempting to register a candidate within the election period ...'
);
Local.setGlobalSlot(params.electionPreconditions.startElection.add(1));

let previousEventsVoter = voterContract.reducer.getActions().length;
let previousEventsCandidate = candidateContract.reducer.getActions().length;

let lateCandidate = Member.from(
  PrivateKey.random().toPublicKey(),
  UInt64.from(200)
);
addAccount(lateCandidate);

await assertValidTx(
  false,
  () => {
    // register late candidate
    voting.candidateRegistration(lateCandidate);
  },
  feePayer,
  'Outside of election period!'
);

console.log(
  'attempting to register a voter within the election period ...'
);

let lateVoter = Member.from(
  PrivateKey.random().toPublicKey(),
  UInt64.from(50)
);
addAccount(lateVoter);

await assertValidTx(
  false,
  () => {
    // register late voter
    voting.voterRegistration(lateVoter);
  },
  feePayer,
  'Outside of election period!'
);

if (previousEventsVoter !== voterContract.reducer.getActions().length) {
  throw Error('events emitted but should not have been');
}
if (
  previousEventsCandidate !== candidateContract.reducer.getActions().length
) {
  throw Error('events emitted but should not have been');
}

if (
  !candidateContract.committedMembers
    .get()
    .equals(candidatesStore.getRoot())
    .toBoolean()
) {
  throw Error(
    'candidatesStore merkle root does not match on-chain committed members'
    );
  }

  if (
    !voterContract.committedMembers
      .get()
      .equals(votersStore.getRoot())
      .toBoolean()
  ) {
    throw Error(
      'votersStore merkle root does not match on-chain committed members'
    );
  }

  /*
  test case description:
    attempting to count votes before any votes were casted

  preconditions:
    - no votes have been casted

  tested cases:
    - count votes -> success, but no state change

  expected results:
    - no state change
*/
  console.log('attempting to count votes but no votes were casted...');

  let beforeAccumulator = voting.accumulatedVotes.get();
  let beforeCommitted = voting.committedVotes.get();
  await assertValidTx(
    true,
    () => {
      voting.countVotes();
    },
    feePayer
  );

  if (!beforeAccumulator.equals(voting.accumulatedVotes.get()).toBoolean()) {
    throw Error('state changed but it should not have!');
  }
  if (!beforeCommitted.equals(voting.committedVotes.get()).toBoolean()) {
    throw Error('state changed but it should not have!');
  }

  /*
  test case description:
    happy path voting for candidate

  preconditions:
    - slot is within predefine precondition slot
    - voters and candidates have been registered previously

  tested cases:
    - voting for candidate -> success

  expected results:
    - isMember check on voter and candidate
    - vote invoked
    - vote sequence event emitted
    - state unchanged
*/
  console.log('attempting to vote for the candidate...');

  let currentCandidate: Member;

  await assertValidTx(
    true,
    () => {
      // attempting to vote for the registered candidate
      currentCandidate = candidatesStore.get(0n)!;
      currentCandidate.witness = new MyMerkleWitness(
        candidatesStore.getWitness(0n)
      );
      currentCandidate.votesWitness = new MyMerkleWitness(
        votesStore.getWitness(0n)
      );

      let v = votersStore.get(0n)!;
      v.witness = new MyMerkleWitness(votersStore.getWitness(0n));

      voting.vote(currentCandidate, v);
    },
    feePayer
  );

  vote(0n, votesStore, candidatesStore);

  numberOfEvents = voting.reducer.getActions().length;
  if (numberOfEvents !== 1) {
    throw Error('Should have emitted 1 event after voting for a candidate');
  }

  /*
  test case description:
    voting for invalid candidate

  preconditions:
    - slot is within predefine precondition slot
    - candidate is invalid (not registered)
    - voting for voter
    - unregistered voter
  tested cases:
    - voting for fake candidate -> failure
    - unregistered voter voting for candidate -> failure
    - voter voting for voter -> failure

  expected results:
    - isMember check on voter and candidate -> fails and tx fails
    - no state changes and no emitted events

*/
  console.log('attempting to vote for a fake candidate...');

  let fakeCandidate = Member.from(
    PrivateKey.random().toPublicKey(),
    params.candidatePreconditions.minMina.add(1)
  );
  addAccount(fakeCandidate);

  await assertValidTx(
    false,
    () => {
      // attempting to vote for the registered candidate

      voting.vote(fakeCandidate, votersStore.get(0n)!);
    },
    feePayer,
    'Member is not a candidate!'
  );

  console.log('unregistered voter attempting to vote');

  let fakeVoter = Member.from(
    PrivateKey.random().toPublicKey(),
    UInt64.from(50)
  );
  addAccount(fakeVoter);

  await assertValidTx(
    false,
    () => {
      voting.vote(fakeVoter, votersStore.get(0n)!);
    },
    feePayer,
    'Member is not a candidate!'
  );

  console.log('attempting to vote for voter...');

  await assertValidTx(
    false,
    () => {
      const voter = votersStore.get(0n)!;
      voting.vote(voter, votersStore.get(0n)!);
    },
    feePayer,
    'Member is not a candidate!'
  );

  /*
  test case description:
    happy path - vote counting

  preconditions:
    - votes were emitted

  tested cases:
    - counting votes -> success

  expected results:
    - counts all emitted votes through sequence events
    - updates on-chain state to equal off-chain state
    - prints final result (helper function)

*/
  console.log('counting votes...');

  await assertValidTx(
    true,
    () => {
      voting.countVotes();
    },
    feePayer
  );

  if (!voting.committedVotes.get().equals(votesStore.getRoot()).toBoolean()) {
    throw Error(
      'votesStore merkle root does not match on-chain committed votes'
    );
  }

  console.log('election is over, printing results');

  let results = getResults(voting, votesStore);
  console.log(results);

  if (results[currentCandidate!.publicKey.toBase58()] !== 1) {
    throw Error(
      &nbsp;Candidate ${currentCandidate!.publicKey.toBase58()} should have one vote, but has ${
        results[currentCandidate!.publicKey.toBase58()]
      } &nbsp;
    );
  }

  console.log('testing after election state');
```

```
  /*
  test case description:
    registering voter and candidates AFTER election has ended

  preconditions:
    - election ended

  tested cases:
    - registering voter -> failure
    - registering candidate -> failure

  expected results:
    - no state changes
    - no events emitted

*/
  console.log('attempting to register voter after election has ended');

  let voter = Member.from(
    PrivateKey.random().toPublicKey(),
    params.voterPreconditions.minMina.add(1)
  );
  addAccount(voter);

  await assertValidTx(
    false,
    () => {
      voting.voterRegistration(voter);
    },
    feePayer,
    'Outside of election period!'
  );

  console.log('attempting to register candidate after election has ended');

  let candidate = Member.from(
    PrivateKey.random().toPublicKey(),
    params.candidatePreconditions.minMina.add(1)
  );
  addAccount(candidate);

  await assertValidTx(
    false,
    () => {
      voting.candidateRegistration(candidate);
    },
    feePayer,
    'Outside of election period!'
  );
}
```

```
  console.log('test successful!');
}
```

</file>

<file>

## path: /src/examples/zkapps/voting/voting.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/voting/voting.ts

```
import {
  Field,
  SmartContract,
  state,
  State,
  method,
  DeployArgs,
  Permissions,
  PublicKey,
  Bool,
  Reducer,
  provablePure,
  AccountUpdate,
  Provable,
} from 'o1js';

import { Member } from './member.js';
import {
  ElectionPreconditions,
  ParticipantPreconditions,
} from './preconditions.js';
import { Membership_ } from './membership.js';

/**
 * Address to the Membership instance that keeps track of Candidates.
 */
let candidateAddress = PublicKey.empty();

/**
 * Address to the Membership instance that keeps track of Voters.
 */
let voterAddress = PublicKey.empty();

/**
 * Requirements in order for a Member to participate in the election as a Candidate.
 */
let candidatePreconditions = ParticipantPreconditions.default;

/**
 * Requirements in order for a Member to participate in the election as a Voter.
 */
let voterPreconditions = ParticipantPreconditions.default;

/**
 * Defines the preconditions of an election.
 */
let electionPreconditions = ElectionPreconditions.default;

interface VotingParams {
  electionPreconditions: ElectionPreconditions;
  voterPreconditions: ParticipantPreconditions;
  candidatePreconditions: ParticipantPreconditions;
  candidateAddress: PublicKey;
  voterAddress: PublicKey;
  contractAddress: PublicKey;
  doProofs: boolean;
}

/**
 * Returns a new contract instance that based on a set of preconditions.
 * @param params {@link Voting_}
 */
export async function Voting(params: VotingParams): Promise<Voting_> {
  electionPreconditions = params.electionPreconditions;
  voterPreconditions = params.voterPreconditions;
  candidatePreconditions = params.candidatePreconditions;
  candidateAddress = params.candidateAddress;
  voterAddress = params.voterAddress;

  let contract = new Voting_(params.contractAddress);
  params.doProofs = true;
  if (params.doProofs) {
    await Voting_.compile();
  }
  return contract;
}

export class Voting_ extends SmartContract {
  /**
   * Root of the merkle tree that stores all committed votes.
   */
  @state(Field) committedVotes = State<Field>();

  /**
   * Accumulator of all emitted votes.
   */
  @state(Field) accumulatedVotes = State<Field>();

  reducer = Reducer({ actionType: Member });

  events = {
    newVoteFor: PublicKey,
    newVoteState: provablePure({
      accumulatedVotesRoot: Field,
      committedVotesRoot: Field,
    }),
  };

  deploy(args: DeployArgs) {
    super.deploy(args);
    this.account.permissions.set({
      ...Permissions.default(),
      editState: Permissions.proofOrSignature(),
      editActionState: Permissions.proofOrSignature(),
      incrementNonce: Permissions.proofOrSignature(),
      setVerificationKey: Permissions.none(),
      setPermissions: Permissions.proofOrSignature(),
    });
    this.accumulatedVotes.set(Reducer.initialActionState);
  }

  /**
   * Method used to register a new voter. Calls the &nbsp;addEntry(member)&nbsp; method of the Voter-
  Membership contract.
   * @param member
   */
  @method
  voterRegistration(member: Member) {
    let currentSlot = this.network.globalSlotSinceGenesis.get();
    this.network.globalSlotSinceGenesis.assertBetween(
      currentSlot,
      currentSlot.add(10)
    );

    // can only register voters before the election has started
    Provable.if(
      electionPreconditions.enforce,
      currentSlot.lessThanOrEqual(electionPreconditions.startElection),
      Bool(true)
    ).assertTrue('Outside of election period!');

    // can only register voters if their balance is gte the minimum amount required
    // this snippet pulls the account data of an address from the network

    let accountUpdate = AccountUpdate.create(member.publicKey);

    accountUpdate.account.balance.assertEquals(
      accountUpdate.account.balance.get()
    );

    let balance = accountUpdate.account.balance.get();

    balance.assertGreaterThanOrEqual(
      voterPreconditions.minMina,
      'Balance not high enough!'
    );
    balance.assertLessThanOrEqual(
      voterPreconditions.maxMina,
      'Balance too high!'
    );

    let VoterContract: Membership_ = new Membership_(voterAddress);
    let exists = VoterContract.addEntry(member);

    // the check happens here because we want to see if the other contract returns a value
    // if exists is true, that means the member already exists within the accumulated state
    // if its false, its a new entry
    exists.assertFalse('Member already exists!');
  }

  /**
   * Method used to register a new candidate.
   * Calls the &nbsp;addEntry(member)&nbsp; method of the Candidate-Membership contract.
   * @param member
   */
  @method
  candidateRegistration(member: Member) {
    let currentSlot = this.network.globalSlotSinceGenesis.get();
    this.network.globalSlotSinceGenesis.assertBetween(
      currentSlot,
      currentSlot.add(10)
    );

    // can only register candidates before the election has started
    Provable.if(
      electionPreconditions.enforce,
      currentSlot.lessThanOrEqual(electionPreconditions.startElection),
      Bool(true)
    ).assertTrue('Outside of election period!');

    // can only register candidates if their balance is gte the minimum amount required
    // and lte the maximum amount
    // this snippet pulls the account data of an address from the network

    let accountUpdate = AccountUpdate.create(member.publicKey);
    accountUpdate.account.balance.assertEquals(
      accountUpdate.account.balance.get()
    );

    let balance = accountUpdate.account.balance.get();

    balance.assertGreaterThanOrEqual(
      candidatePreconditions.minMina,
      'Balance not high enough!'
    );
    balance.assertLessThanOrEqual(
      candidatePreconditions.maxMina,
      'Balance too high!'
    );

    let CandidateContract: Membership_ = new Membership_(candidateAddress);
    let exists = CandidateContract.addEntry(member);

    // the check happens here because we want to see if the other contract returns a value
    // if exists is true, that means the member already exists within the accumulated state
    // if its false, its a new entry
    exists.assertEquals(false);
  }

  /**
   * Method used to register update all pending member registrations.
   * Calls the &nbsp;publish()&nbsp; method of the Candidate-Membership and Voter-Membership contract.
   */
  @method
  approveRegistrations() {
    // Invokes the publish method of both Voter and Candidate Membership contracts.
    let VoterContract: Membership_ = new Membership_(voterAddress);
    VoterContract.publish();

    let CandidateContract: Membership_ = new Membership_(candidateAddress);
    CandidateContract.publish();
  }

  /**
   * Method used to cast a vote to a specific candidate.
   * Dispatches a new vote sequence event.
   * @param candidate
   * @param voter
   */
  @method
  vote(candidate: Member, voter: Member) {
    let currentSlot = this.network.globalSlotSinceGenesis.get();
    this.network.globalSlotSinceGenesis.assertBetween(
      currentSlot,
      currentSlot.add(10)
    );

    // we can only vote in the election period time frame
    Provable.if(
      electionPreconditions.enforce,
      currentSlot
        .greaterThanOrEqual(electionPreconditions.startElection)
        .and(currentSlot.lessThanOrEqual(electionPreconditions.endElection)),
      Bool(true)
    ).assertTrue('Not in voting period!');

    // verifying that both the voter and the candidate are actually part of our member set
    // ideally we would also verify a signature here, but ignoring that for now
```

```
    let VoterContract: Membership_ = new Membership_(voterAddress);
    VoterContract.isMember(voter).assertTrue('Member is not a voter!');

    let CandidateContract: Membership_ = new Membership_(candidateAddress);
    CandidateContract.isMember(candidate).assertTrue(
      'Member is not a candidate!'
    );

    // emits a sequence event with the information about the candidate
    this.reducer.dispatch(candidate);

    this.emitEvent('newVoteFor', candidate.publicKey);
  }

  /**
   * Method used to accumulate all pending votes from sequence events
   * and applies state changes to the votes merkle tree.
   */
  @method
  countVotes() {
    let accumulatedVotes = this.accumulatedVotes.get();
    this.accumulatedVotes.assertEquals(accumulatedVotes);

    let committedVotes = this.committedVotes.get();
    this.committedVotes.assertEquals(committedVotes);

    let { state: newCommittedVotes, actionState: newAccumulatedVotes } =
      this.reducer.reduce(
        this.reducer.getActions({ fromActionState: accumulatedVotes }),
        Field,
        (state: Field, action: Member) => {
          // apply one vote
          action = action.addVote();
          // this is the new root after we added one vote
          return action.votesWitness.calculateRootSlow(action.getHash());
        },
        // initial state
        { state: committedVotes, actionState: accumulatedVotes }
      );

    this.committedVotes.set(newCommittedVotes);
    this.accumulatedVotes.set(newAccumulatedVotes);

    this.emitEvent('newVoteState', {
      committedVotesRoot: newCommittedVotes,
      accumulatedVotesRoot: newAccumulatedVotes,
    });
  }
}
```

</file>

<file>

## path: /src/examples/zkapps/voting/voting_lib.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/voting/voting_lib.ts

```
import { Member, MyMerkleWitness } from './member.js';
import { OffchainStorage } from './off_chain_storage.js';
import { Voting_ } from './voting.js';
import { Mina, PrivateKey } from 'o1js';
import { Printer } from 'prettier';
/**
 * Updates off-chain storage when registering a member or candidate
 * @param {bigint} i index of memberStore or candidatesStore
 * @param {Member} m member to register
 * @param {OffchainStorage<Member>} store  off-chain store which should be used when registering a new member
 * @param {any} Local  local blockchain instance in use
 */
export function registerMember(
  i: bigint,
  m: Member,
  store: OffchainStorage<Member>,
  Local: any
): Member {
  Local.addAccount(m.publicKey, m.balance.toString());

  // we will also have to keep track of new voters and candidates within our off-chain merkle tree
  store.set(i, m); // setting voter 0n
  // setting the merkle witness
  m.witness = new MyMerkleWitness(store.getWitness(i));

  return m;
}

/**
 * Updates off-chain storage after voting
 * @param {bigint} i                          index of candidateStore and votesStore
 * @param {OffchainStorage<Member>} votesStore  votes off-chain storage
 * @param {OffchainStorage<Member>} votesStore  candidates off-chain storage
 */
export function vote(
  i: bigint,
  votesStore: OffchainStorage<Member>,
  candidateStore: OffchainStorage<Member>
) {
  let c_ = votesStore.get(i)!;
  if (!c_) {
    votesStore.set(i, candidateStore.get(i)!);
    c_ = votesStore.get(i)!;
  }
  c_ = c_.addVote();
  votesStore.set(i, c_);
  return c_;
}

/**
 * Prints the voting results of an election
 */
export function getResults(
  voting: Voting_,
  votesStore: OffchainStorage<Member>
) {
  if (!voting.committedVotes.get().equals(votesStore.getRoot()).toBoolean()) {
    throw new Error('On-chain root is not up to date with the off-chain tree');
  }

  let result: Record<string, number> = {};
  votesStore.forEach((m, i) => {
    result[m.publicKey.toBase58()] = Number(m.votes.toString());
  });
  return result;
}

/**
 * Checks if a transaction is valid.
 * If it is expected to fail, an expected error message needs to be provided
 * @boolean expectedToBeValid - true if the transaction is expected to pass without error
 */
export async function assertValidTx(
  expectToBeValid: boolean,
  cb: () => void,
  feePayer: PrivateKey,
  msg?: string
) {
  let failed = false;
  let err;
  try {
    let tx = await Mina.transaction(feePayer.toPublicKey(), cb);
    await tx.prove();
    await tx.sign([feePayer]).send();
  } catch (e: any) {
    failed = true;
    err = e;
  }

  if (!failed && expectToBeValid) {
    console.log('> transaction valid!');
  } else if (failed && expectToBeValid) {
    console.error('transaction failed but should have passed');
    console.log(cb.toString());
    console.error('with error message: ');
    throw Error(err);
  } else if (failed && !expectToBeValid) {
    if (err.message.includes(msg ?? 'NO__EXPECTED_ERROR_MESSAGE_SET')) {
      console.log('> transaction failed, as expected!');
    } else {
      console.log(err);
      throw Error('transaction failed, but got a different error message!');
    }
  } else if (!failed && !expectToBeValid) {
    throw Error('transaction passed but should have failed');
  } else {
    throw Error('transaction was expected to fail but it passed');
  }
}
```

</file>

<file>

## path: /src/examples/zkapps/zkapp-self-update.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkapps/zkapp-self-update.ts

```
/**
 * This example deploys a zkApp and then updates its verification key via proof, self-replacing the zkApp
 */
import {
  SmartContract,
  VerificationKey,
  method,
  Permissions,
  isReady,
  PrivateKey,
  Mina,
  AccountUpdate,
  Circuit,
  Provable,
} from 'o1js';

class Foo extends SmartContract {
  init() {
    super.init();
    this.account.permissions.set({
      ...Permissions.default(),
      setVerificationKey: Permissions.proof(),
    });
  }

  @method replaceVerificationKey(verificationKey: VerificationKey) {
    this.account.verificationKey.set(verificationKey);
  }
}
```

```
class Bar extends SmartContract {
  @method call() {
    Provable.log('Bar');
  }
}

// setup

await isReady;

const Local = Mina.LocalBlockchain({ proofsEnabled: true });
Mina.setActiveInstance(Local);

const zkAppPrivateKey = PrivateKey.random();
const zkAppAddress = zkAppPrivateKey.toPublicKey();
const zkApp = new Foo(zkAppAddress);

const { privateKey: deployerKey, publicKey: deployerAccount } =
  Local.testAccounts[0];

// deploy first verification key

await Foo.compile();

const tx = await Mina.transaction(deployerAccount, () => {
  AccountUpdate.fundNewAccount(deployerAccount);
  zkApp.deploy();
});
await tx.prove();
await tx.sign([deployerKey, zkAppPrivateKey]).send();

const fooVerificationKey = Mina.getAccount(zkAppAddress).zkapp?.verificationKey;
Provable.log('original verification key', fooVerificationKey);

// update verification key

const { verificationKey: barVerificationKey } = await Bar.compile();

const tx2 = await Mina.transaction(deployerAccount, () => {
  zkApp.replaceVerificationKey(barVerificationKey);
});
await tx2.prove();
await tx2.sign([deployerKey]).send();

const updatedVerificationKey =
  Mina.getAccount(zkAppAddress).zkapp?.verificationKey;

// should be different from Foo
Provable.log('updated verification key', updatedVerificationKey);
```

</file>

<file>

## path: /src/examples/zkprogram/README.md

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkprogram/README.md

# ZkProgram

These examples focus on how to use ZkProgram, our main API for creating proofs outside of smart contract.
</file>

<file>

## path: /src/examples/zkprogram/gadgets.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkprogram/gadgets.ts

```
import { Field, Provable, Gadgets, ZkProgram } from 'o1js';

let cs = Provable.constraintSystem(() => {
  let f = Provable.witness(Field, () => Field(12));

  let res1 = Gadgets.rotate(f, 2, 'left');
  let res2 = Gadgets.rotate(f, 2, 'right');

  res1.assertEquals(Field(48));
  res2.assertEquals(Field(3));

  Provable.log(res1);
  Provable.log(res2);
});
console.log('constraint system: ', cs);

const BitwiseProver = ZkProgram({
  name: 'bitwise',
  methods: {
    rot: {
      privateInputs: [],
      method: () => {
        let a = Provable.witness(Field, () => Field(48));
        let actualLeft = Gadgets.rotate(a, 2, 'left');
        let actualRight = Gadgets.rotate(a, 2, 'right');

        let expectedLeft = Field(192);
        actualLeft.assertEquals(expectedLeft);

        let expectedRight = Field(12);
        actualRight.assertEquals(expectedRight);
      },
    },
    xor: {
      privateInputs: [],
      method: () => {
        let a = Provable.witness(Field, () => Field(5));
        let b = Provable.witness(Field, () => Field(2));
        let actual = Gadgets.xor(a, b, 4);
        let expected = Field(7);
        actual.assertEquals(expected);
      },
    },
    and: {
      privateInputs: [],
      method: () => {
        let a = Provable.witness(Field, () => Field(3));
        let b = Provable.witness(Field, () => Field(5));
        let actual = Gadgets.and(a, b, 4);
        let expected = Field(1);
        actual.assertEquals(expected);
      },
    },
  },
});

console.log('compiling..');

console.time('compile');
await BitwiseProver.compile();
console.timeEnd('compile');

console.log('proving..');

console.time('rotation prove');
let rotProof = await BitwiseProver.rot();
console.timeEnd('rotation prove');
if (!(await BitwiseProver.verify(rotProof))) throw Error('rot: Invalid proof');

console.time('xor prove');
let xorProof = await BitwiseProver.xor();
console.timeEnd('xor prove');
if (!(await BitwiseProver.verify(xorProof))) throw Error('xor: Invalid proof');

console.time('and prove');
let andProof = await BitwiseProver.and();
console.timeEnd('and prove');
if (!(await BitwiseProver.verify(andProof))) throw Error('and: Invalid proof');
```

</file>

<file>

## path: /src/examples/zkprogram/program-with-input.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkprogram/program-with-input.ts

```
import {
  SelfProof,
  Field,
  ZkProgram,
  verify,
  isReady,
  Proof,
  JsonProof,
  Provable,
} from 'o1js';

await isReady;

let MyProgram = ZkProgram({
  name: 'example-with-input',
  publicInput: Field,

  methods: {
    baseCase: {
      privateInputs: [],
      method(input: Field) {
        input.assertEquals(Field(0));
      },
    },

    inductiveCase: {
      privateInputs: [SelfProof],
      method(input: Field, earlierProof: SelfProof<Field, void>) {
        earlierProof.verify();
        earlierProof.publicInput.add(1).assertEquals(input);
      },
    },
  },
});
// type sanity checks
MyProgram.publicInputType satisfies typeof Field;
MyProgram.publicOutputType satisfies Provable<void>;

let MyProof = ZkProgram.Proof(MyProgram);

console.log('program digest', MyProgram.digest());

console.log('compiling MyProgram...');
let { verificationKey } = await MyProgram.compile();
console.log('verification key', verificationKey.slice(0, 10) + '..');

console.log('proving base case...');
```

```
let proof = await MyProgram.baseCase(Field(0));
proof = testJsonRoundtrip(MyProof, proof);

// type sanity check
proof satisfies Proof<Field, void>;

console.log('verify...');
let ok = await verify(proof.toJSON(), verificationKey);
console.log('ok?', ok);

console.log('verify alternative...');
ok = await MyProgram.verify(proof);
console.log('ok (alternative)?', ok);

console.log('proving step 1...');
proof = await MyProgram.inductiveCase(Field(1), proof);
proof = testJsonRoundtrip(MyProof, proof);

console.log('verify...');
ok = await verify(proof, verificationKey);
console.log('ok?', ok);

console.log('verify alternative...');
ok = await MyProgram.verify(proof);
console.log('ok (alternative)?', ok);

console.log('proving step 2...');
proof = await MyProgram.inductiveCase(Field(2), proof);
proof = testJsonRoundtrip(MyProof, proof);

console.log('verify...');
ok = await verify(proof.toJSON(), verificationKey);

console.log('ok?', ok && proof.publicInput.toString() === '2');

function testJsonRoundtrip<
  P extends Proof<any, any>,
  MyProof extends { fromJSON(jsonProof: JsonProof): P }
>(MyProof: MyProof, proof: P) {
  let jsonProof = proof.toJSON();
  console.log(
    'json proof',
    JSON.stringify({
      ...jsonProof,
      proof: jsonProof.proof.slice(0, 10) + '..',
    })
  );
  return MyProof.fromJSON(jsonProof);
}
```

</file>

<file>

## path: /src/examples/zkprogram/program.ts

url: https://github.com/o1-labs/o1js/blob/main/src/examples/zkprogram/program.ts

```
import {
  SelfProof,
  Field,
  ZkProgram,
  verify,
  isReady,
  Proof,
  JsonProof,
  Provable,
  Empty,
} from 'o1js';

await isReady;

let MyProgram = ZkProgram({
  name: 'example-with-output',
  publicOutput: Field,

  methods: {
    baseCase: {
      privateInputs: [],
      method() {
        return Field(0);
      },
    },

    inductiveCase: {
      privateInputs: [SelfProof],
      method(earlierProof: SelfProof<Empty, Field>) {
        earlierProof.verify();
        return earlierProof.publicOutput.add(1);
      },
    },
  },
});
// type sanity checks
MyProgram.publicInputType satisfies Provable<Empty>;
MyProgram.publicOutputType satisfies typeof Field;

let MyProof = ZkProgram.Proof(MyProgram);

console.log('program digest', MyProgram.digest());

console.log('compiling MyProgram...');
let { verificationKey } = await MyProgram.compile();
```

```
console.log('verification key', verificationKey.slice(0, 10) + '..');

console.log('proving base case...');
let proof = await MyProgram.baseCase();
proof = testJsonRoundtrip(MyProof, proof);

// type sanity check
proof satisfies Proof<undefined, Field>;

console.log('verify...');
let ok = await verify(proof.toJSON(), verificationKey);
console.log('ok?', ok);

console.log('verify alternative...');
ok = await MyProgram.verify(proof);
console.log('ok (alternative)?', ok);

console.log('proving step 1...');
proof = await MyProgram.inductiveCase(proof);
proof = testJsonRoundtrip(MyProof, proof);

console.log('verify...');
ok = await verify(proof, verificationKey);
console.log('ok?', ok);

console.log('verify alternative...');
ok = await MyProgram.verify(proof);
console.log('ok (alternative)?', ok);

console.log('proving step 2...');
proof = await MyProgram.inductiveCase(proof);
proof = testJsonRoundtrip(MyProof, proof);

console.log('verify...');
ok = await verify(proof.toJSON(), verificationKey);

console.log('ok?', ok && proof.publicOutput.toString() === '2');

function testJsonRoundtrip<
  P extends Proof<any, any>,
  MyProof extends { fromJSON(jsonProof: JsonProof): P }
>(MyProof: MyProof, proof: P) {
  let jsonProof = proof.toJSON();
  console.log(
    'json proof',
    JSON.stringify({
      ...jsonProof,
      proof: jsonProof.proof.slice(0, 10) + '..',
    })
  );
  return MyProof.fromJSON(jsonProof);
}
```

</file>

<file>

## path: /src/index.ts

url: https://github.com/o1-labs/o1js/blob/main/src/index.ts

```
export type { ProvablePure } from './snarky.js';
export { Ledger } from './snarky.js';
export { Field, Bool, Group, Scalar } from './lib/core.js';
export { Poseidon, TokenSymbol } from './lib/hash.js';
export * from './lib/signature.js';
export type {
  ProvableExtended,
  FlexibleProvable,
  FlexibleProvablePure,
  InferProvable,
} from './lib/circuit_value.js';
export {
  CircuitValue,
  prop,
  arrayProp,
  matrixProp,
  provable,
  provablePure,
  Struct,
} from './lib/circuit_value.js';
export { Provable } from './lib/provable.js';
export { Circuit, Keypair, public_, circuitMain } from './lib/circuit.js';
export { UInt32, UInt64, Int64, Sign } from './lib/int.js';
export { Gadgets } from './lib/gadgets/gadgets.js';
export { Types } from './bindings/mina-transaction/types.js';

export * as Mina from './lib/mina.js';
export type { DeployArgs } from './lib/zkapp.js';
export {
  SmartContract,
  method,
  declareMethods,
  Account,
  VerificationKey,
  Reducer,
} from './lib/zkapp.js';
export { state, State, declareState } from './lib/state.js';

export type { JsonProof } from './lib/proof_system.js';
export {
  Proof,
  SelfProof,
  verify,
  Empty,
  Undefined,
  Void,
} from './lib/proof_system.js';
export { Cache, CacheHeader } from './lib/proof-system/cache.js';

export {
  Token,
  TokenId,
  AccountUpdate,
  Permissions,
  ZkappPublicInput,
} from './lib/account_update.js';

export type { TransactionStatus } from './lib/fetch.js';
export {
  fetchAccount,
  fetchLastBlock,
  fetchTransactionStatus,
  checkZkappTransaction,
  fetchEvents,
  addCachedAccount,
  setGraphqlEndpoint,
  setGraphqlEndpoints,
  setArchiveGraphqlEndpoint,
  sendZkapp,
  Lightnet,
} from './lib/fetch.js';
export * as Encryption from './lib/encryption.js';
export * as Encoding from './bindings/lib/encoding.js';
export { Character, CircuitString } from './lib/string.js';
export { MerkleTree, MerkleWitness } from './lib/merkle_tree.js';
export { MerkleMap, MerkleMapWitness } from './lib/merkle_map.js';

export { Nullifier } from './lib/nullifier.js';

import { ExperimentalZkProgram, ZkProgram } from './lib/proof_system.js';
export { ZkProgram };

// experimental APIs
import { Callback } from './lib/zkapp.js';
import { createChildAccountUpdate } from './lib/account_update.js';
import { memoizeWitness } from './lib/provable.js';
export { Experimental };

const Experimental_ = {
  Callback,
  createChildAccountUpdate,
  memoizeWitness,
};
```

```
type Callback_<Result> = Callback<Result>;

/**
 * This module exposes APIs that are unstable, in the sense that the API surface is expected to change.
 * (Not unstable in the sense that they are less functional or tested than other parts.)
 */
namespace Experimental {
  /** @deprecated &nbsp;ZkProgram&nbsp; has moved out of the Experimental namespace and is now
directly available as a top-level import &nbsp;ZkProgram&nbsp;.
   * The old &nbsp;Experimental.ZkProgram&nbsp; API has been deprecated in favor of the new
&nbsp;ZkProgram&nbsp; top-level import.
   */
  export let ZkProgram = ExperimentalZkProgram;
  export let createChildAccountUpdate = Experimental_.createChildAccountUpdate;
  export let memoizeWitness = Experimental_.memoizeWitness;
  export let Callback = Experimental_.Callback;
  export type Callback<Result> = Callback_<Result>;
}

Error.stackTraceLimit = 1000;

// deprecated stuff
export { isReady, shutdown };

/**
 * @deprecated &nbsp;await isReady&nbsp; is no longer needed. Remove it from your code.
 */
let isReady = Promise.resolve();

/**
 * @deprecated &nbsp;shutdown()&nbsp; is no longer needed, and is a no-op. Remove it from your code.
 */
function shutdown() {}
```

</file>

<file>

## path: /src/lib/account_update.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/account_update.ts

```
import {
  cloneCircuitValue,
  FlexibleProvable,
  provable,
  provablePure,
} from './circuit_value.js';
import { memoizationContext, memoizeWitness, Provable } from './provable.js';
import { Field, Bool } from './core.js';
import { Pickles, Test } from '../snarky.js';
import { jsLayout } from '../bindings/mina-transaction/gen/js-layout.js';
import {
  Types,
  TypesBigint,
  toJSONEssential,
} from '../bindings/mina-transaction/types.js';
import { PrivateKey, PublicKey } from './signature.js';
import { UInt64, UInt32, Int64, Sign } from './int.js';
import * as Mina from './mina.js';
import { SmartContract } from './zkapp.js';
import * as Precondition from './precondition.js';
import { dummyBase64Proof, Empty, Proof, Prover } from './proof_system.js';
import { Memo } from '../mina-signer/src/memo.js';
import {
  Events,
  Actions,
} from '../bindings/mina-transaction/transaction-leaves.js';
import { TokenId as Base58TokenId } from './base58-encodings.js';
import { hashWithPrefix, packToFields } from './hash.js';
import { mocks, prefixes } from '../bindings/crypto/constants.js';
import { Context } from './global-context.js';
import { assert } from './errors.js';
import { MlArray } from './ml/base.js';
import { Signature, signFieldElement } from '../mina-signer/src/signature.js';
import { MlFieldConstArray } from './ml/fields.js';
import { transactionCommitments } from '../mina-signer/src/sign-zkapp-command.js';

// external API
export { AccountUpdate, Permissions, ZkappPublicInput };
// internal API
export {
  smartContractContext,
  SetOrKeep,
  Permission,
  Preconditions,
  Body,
  Authorization,
  FeePayerUnsigned,
  ZkappCommand,
  addMissingSignatures,
  addMissingProofs,
  ZkappStateLength,
  Events,
  Actions,
  TokenId,
  Token,
  CallForest,
  createChildAccountUpdate,
  AccountUpdatesLayout,
  zkAppProver,
  SmartContractContext,
  dummySignature,
};

const ZkappStateLength = 8;

type SmartContractContext = {
  this: SmartContract;
  methodCallDepth: number;
  selfUpdate: AccountUpdate;
};
let smartContractContext = Context.create<null | SmartContractContext>({
  default: null,
});

type ZkappProverData = {
  transaction: ZkappCommand;
  accountUpdate: AccountUpdate;
  index: number;
};
let zkAppProver = Prover<ZkappProverData>();

type AuthRequired = Types.Json.AuthRequired;

type AccountUpdateBody = Types.AccountUpdate['body'];
type Update = AccountUpdateBody['update'];

type MayUseToken = AccountUpdateBody['mayUseToken'];

/**
 * Preconditions for the network and accounts
 */
type Preconditions = AccountUpdateBody['preconditions'];

/**
 * Either set a value or keep it the same.
 */
type SetOrKeep<T> = { isSome: Bool; value: T };

const True = () => Bool(true);
const False = () => Bool(false);

/**
 * One specific permission value.
 *
 * A {@link Permission} tells one specific permission for our zkapp how it
 * should behave when presented with requested modifications.
 *
 * Use static factory methods on this class to use a specific behavior. See
 * documentation on those methods to learn more.
 */
type Permission = Types.AuthRequired;
let Permission = {
  /**
  * Modification is impossible.
  */
 impossible: (): Permission => ({
   constant: True(),
   signatureNecessary: True(),
   signatureSufficient: False(),
 }),

 /**
  * Modification is always permitted
  */
 none: (): Permission => ({
   constant: True(),
   signatureNecessary: False(),
   signatureSufficient: True(),
 }),

 /**
  * Modification is permitted by zkapp proofs only
  */
 proof: (): Permission => ({
   constant: False(),
   signatureNecessary: False(),
   signatureSufficient: False(),
 }),

 /**
  * Modification is permitted by signatures only, using the private key of the zkapp account
  */
 signature: (): Permission => ({
   constant: False(),
   signatureNecessary: True(),
   signatureSufficient: True(),
 }),

 /**
  * Modification is permitted by zkapp proofs or signatures
  */
 proofOrSignature: (): Permission => ({
   constant: False(),
   signatureNecessary: False(),
   signatureSufficient: True(),
 }),
};

// TODO: we could replace the interface below if we could bridge annotations from OCaml
type Permissions_ = Update['permissions']['value'];

/**
 * Permissions specify how specific aspects of the zkapp account are allowed
 * to be modified. All fields are denominated by a {@link Permission}.
 */
```

```
interface Permissions extends Permissions_ {
  /**
   * The {@link Permission} corresponding to the 8 state fields associated with
   * an account.
   */
  editState: Permission;

  /**
   * The {@link Permission} corresponding to the ability to send transactions
   * from this account.
   */
  send: Permission;

  /**
   * The {@link Permission} corresponding to the ability to receive transactions
   * to this account.
   */
  receive: Permission;

  /**
   * The {@link Permission} corresponding to the ability to set the delegate
   * field of the account.
   */
  setDelegate: Permission;

  /**
   * The {@link Permission} corresponding to the ability to set the permissions
   * field of the account.
   */
  setPermissions: Permission;

  /**
   * The {@link Permission} corresponding to the ability to set the verification
   * key associated with the circuit tied to this account. Effectively
   * "upgradeability" of the smart contract.
   */
  setVerificationKey: Permission;

  /**
   * The {@link Permission} corresponding to the ability to set the zkapp uri
   * typically pointing to the source code of the smart contract. Usually this
   * should be changed whenever the {@link Permissions.setVerificationKey} is
   * changed. Effectively "upgradeability" of the smart contract.
   */
  setZkappUri: Permission;

  /**
   * The {@link Permission} corresponding to the ability to emit actions to the account.
   */
  editActionState: Permission;

  /**
```

```
   * The {@link Permission} corresponding to the ability to set the token symbol
   * for this account.
   */
  setTokenSymbol: Permission;

  // TODO: doccomments
  incrementNonce: Permission;
  setVotingFor: Permission;
  setTiming: Permission;

  /**
   * Permission to control the ability to include _any_ account update for this
   * account in a transaction. Note that this is more restrictive than all other
   * permissions combined. For normal accounts it can safely be set to &nbsp;none&nbsp;,
   * but for token contracts this has to be more restrictive, to prevent
   * unauthorized token interactions -- for example, it could be
   * &nbsp;proofOrSignature&nbsp;.
   */
  access: Permission;
}
let Permissions = {
  ...Permission,
  /**
   * Default permissions are:
   *
   *   {@link Permissions.editState} = {@link Permission.proof}
   *
   *   {@link Permissions.send} = {@link Permission.signature}
   *
   *   {@link Permissions.receive} = {@link Permission.none}
   *
   *   {@link Permissions.setDelegate} = {@link Permission.signature}
   *
   *   {@link Permissions.setPermissions} = {@link Permission.signature}
   *
   *   {@link Permissions.setVerificationKey} = {@link Permission.signature}
   *
   *   {@link Permissions.setZkappUri} = {@link Permission.signature}
   *
   *   {@link Permissions.editActionState} = {@link Permission.proof}
   *
   *   {@link Permissions.setTokenSymbol} = {@link Permission.signature}
   *
   */
  default: (): Permissions => ({
    editState: Permission.proof(),
    send: Permission.proof(),
    receive: Permission.none(),
    setDelegate: Permission.signature(),
    setPermissions: Permission.signature(),
    setVerificationKey: Permission.signature(),
    setZkappUri: Permission.signature(),
    editActionState: Permission.proof(),
    setTokenSymbol: Permission.signature(),
    incrementNonce: Permission.signature(),
    setVotingFor: Permission.signature(),
    setTiming: Permission.signature(),
    access: Permission.none(),
  }),

  initial: (): Permissions => ({
    editState: Permission.signature(),
    send: Permission.signature(),
    receive: Permission.none(),
    setDelegate: Permission.signature(),
    setPermissions: Permission.signature(),
    setVerificationKey: Permission.signature(),
    setZkappUri: Permission.signature(),
    editActionState: Permission.signature(),
    setTokenSymbol: Permission.signature(),
    incrementNonce: Permission.signature(),
    setVotingFor: Permission.signature(),
    setTiming: Permission.signature(),
    access: Permission.none(),
  }),

  dummy: (): Permissions => ({
    editState: Permission.none(),
    send: Permission.none(),
    receive: Permission.none(),
    access: Permission.none(),
    setDelegate: Permission.none(),
    setPermissions: Permission.none(),
    setVerificationKey: Permission.none(),
    setZkappUri: Permission.none(),
    editActionState: Permission.none(),
    setTokenSymbol: Permission.none(),
    incrementNonce: Permission.none(),
    setVotingFor: Permission.none(),
    setTiming: Permission.none(),
  }),

  allImpossible: (): Permissions => ({
    editState: Permission.impossible(),
    send: Permission.impossible(),
    receive: Permission.impossible(),
    access: Permission.impossible(),
    setDelegate: Permission.impossible(),
    setPermissions: Permission.impossible(),
    setVerificationKey: Permission.impossible(),
    setZkappUri: Permission.impossible(),
    editActionState: Permission.impossible(),
    setTokenSymbol: Permission.impossible(),
    incrementNonce: Permission.impossible(),
    setVotingFor: Permission.impossible(),
    setTiming: Permission.impossible(),
   }),

   fromString: (permission: AuthRequired): Permission => {
    switch (permission) {
     case 'None':
      return Permission.none();
     case 'Either':
      return Permission.proofOrSignature();
     case 'Proof':
      return Permission.proof();
     case 'Signature':
      return Permission.signature();
     case 'Impossible':
      return Permission.impossible();
     default:
      throw Error(
       &nbsp;Cannot parse invalid permission. ${permission} does not exist.&nbsp;
      );
    }
   },

   fromJSON: (
    permissions: NonNullable<
     Types.Json.AccountUpdate['body']['update']['permissions']
    >
   ): Permissions => {
    return Object.fromEntries(
     Object.entries(permissions).map(([k, v]) => [
      k,
      Permissions.fromString(v),
     ])
    ) as unknown as Permissions;
   },
  };

  // TODO: get docstrings from OCaml and delete this interface

  /**
   * The body of describing how some [[ AccountUpdate ]] should change.
   */
  interface Body extends AccountUpdateBody {
   /**
    * The address for this body.
    */
   publicKey: PublicKey;

   /**
    * Specify {@link Update}s to tweakable pieces of the account record backing
    * this address in the ledger.
    */
```

```
update: Update;

/**
 * The TokenId for this account.
 */
tokenId: Field;

/**
 * By what {@link Int64} should the balance of this account change. All
 * balanceChanges must balance by the end of smart contract execution.
 */
balanceChange: { magnitude: UInt64; sgn: Sign };

/**
 * Recent events that have been emitted from this account.
 * Events can be collected by archive nodes.
 *
 * [Check out our documentation about
 * Events!](https://docs.minaprotocol.com/zkapps/advanced-o1js/events)
 */
events: Events;
/**
 * Recent {@link Action}s emitted from this account.
 * Actions can be collected by archive nodes and used in combination with
 * a {@link Reducer}.
 *
 * [Check out our documentation about
 * Actions!](https://docs.minaprotocol.com/zkapps/advanced-o1js/actions-and-reducer)
 */
actions: Events;
/**
 * The type of call.
 */
mayUseToken: MayUseToken;
callData: Field;
callDepth: number;
/**
 * A list of {@link Preconditions} that need to be fulfilled in order for
 * the {@link AccountUpdate} to be valid.
 */
preconditions: Preconditions;
/**
 * Defines if a full commitment is required for this transaction.
 */
useFullCommitment: Bool;
/**
 * Defines if the fee for creating this account should be paid out of this
 * account's balance change.
 *
 * This must only be true if the balance change is larger than the account
 * creation fee and the token ID is the default.
 */
```

```
  implicitAccountCreationFee: Bool;
  /**
   * Defines if the nonce should be incremented with this {@link AccountUpdate}.
   */
  incrementNonce: Bool;
  /**
   * Defines the type of authorization that is needed for this {@link
   * AccountUpdate}.
   *
   * A authorization can be one of three types: None, Proof or Signature
   */
  authorizationKind: AccountUpdateBody['authorizationKind'];
}
const Body = {
  /**
   * A body that doesn't change the underlying account record
   */
  keepAll(
    publicKey: PublicKey,
    tokenId?: Field,
    mayUseToken?: MayUseToken
  ): Body {
    let { body } = Types.AccountUpdate.emptyValue();
    body.publicKey = publicKey;
    if (tokenId) {
      body.tokenId = tokenId;
      body.mayUseToken = Provable.if(
        tokenId.equals(TokenId.default),
        AccountUpdate.MayUseToken.type,
        AccountUpdate.MayUseToken.No,
        AccountUpdate.MayUseToken.ParentsOwnToken
      );
    }
    if (mayUseToken) {
      body.mayUseToken = mayUseToken;
    }
    return body;
  },

  dummy(): Body {
    return Types.AccountUpdate.emptyValue().body;
  },
};

type FeePayer = Types.ZkappCommand['feePayer'];
type FeePayerBody = FeePayer['body'];
const FeePayerBody = {
  keepAll(publicKey: PublicKey, nonce: UInt32): FeePayerBody {
    return {
      publicKey,
      nonce,
      fee: UInt64.zero,
      validUntil: undefined,
    };
  },
};
type FeePayerUnsigned = FeePayer & {
  lazyAuthorization?: LazySignature | undefined;
};

/**
 * Either check a value or ignore it.
 *
 * Used within [[ AccountPredicate ]]s and [[ ProtocolStatePredicate ]]s.
 */
type OrIgnore<T> = { isSome: Bool; value: T };

/**
 * An interval representing all the values between &nbsp;lower&nbsp; and &nbsp;upper&nbsp; inclusive
 * of both the &nbsp;lower&nbsp; and &nbsp;upper&nbsp; values.
 *
 * @typeParam A something with an ordering where one can quantify a lower and
 *            upper bound.
 */
type ClosedInterval<T> = { lower: T; upper: T };

type NetworkPrecondition = Preconditions['network'];
let NetworkPrecondition = {
  ignoreAll(): NetworkPrecondition {
    let stakingEpochData = {
      ledger: { hash: ignore(Field(0)), totalCurrency: ignore(uint64()) },
      seed: ignore(Field(0)),
      startCheckpoint: ignore(Field(0)),
      lockCheckpoint: ignore(Field(0)),
      epochLength: ignore(uint32()),
    };
    let nextEpochData = cloneCircuitValue(stakingEpochData);
    return {
      snarkedLedgerHash: ignore(Field(0)),
      blockchainLength: ignore(uint32()),
      minWindowDensity: ignore(uint32()),
      totalCurrency: ignore(uint64()),
      globalSlotSinceGenesis: ignore(uint32()),
      stakingEpochData,
      nextEpochData,
    };
  },
};

/**
 * Ignores a &nbsp;dummy&nbsp;
 *
 * @param dummy The value to ignore
 * @returns Always an ignored value regardless of the input.
 */
function ignore<T>(dummy: T): OrIgnore<T> {
  return { isSome: Bool(false), value: dummy };
}

/**
 * Ranges between all uint32 values
 */
const uint32 = () => ({ lower: UInt32.from(0), upper: UInt32.MAXINT() });

/**
 * Ranges between all uint64 values
 */
const uint64 = () => ({ lower: UInt64.from(0), upper: UInt64.MAXINT() });

type AccountPrecondition = Preconditions['account'];
const AccountPrecondition = {
  ignoreAll(): AccountPrecondition {
    let appState: Array<OrIgnore<Field>> = [];
    for (let i = 0; i < ZkappStateLength; ++i) {
      appState.push(ignore(Field(0)));
    }
    return {
      balance: ignore(uint64()),
      nonce: ignore(uint32()),
      receiptChainHash: ignore(Field(0)),
      delegate: ignore(PublicKey.empty()),
      state: appState,
      actionState: ignore(Actions.emptyActionState()),
      provedState: ignore(Bool(false)),
      isNew: ignore(Bool(false)),
    };
  },
  nonce(nonce: UInt32): AccountPrecondition {
    let p = AccountPrecondition.ignoreAll();
    AccountUpdate.assertEquals(p.nonce, nonce);
    return p;
  },
};

type GlobalSlotPrecondition = Preconditions['validWhile'];
const GlobalSlotPrecondition = {
  ignoreAll(): GlobalSlotPrecondition {
    return ignore(uint32());
  },
};

const Preconditions = {
  ignoreAll(): Preconditions {
    return {
      account: AccountPrecondition.ignoreAll(),
      network: NetworkPrecondition.ignoreAll(),
    validWhile: GlobalSlotPrecondition.ignoreAll(),
  };
 },
};

type Control = Types.AccountUpdate['authorization'];
type LazyNone = {
 kind: 'lazy-none';
};
type LazySignature = {
 kind: 'lazy-signature';
 privateKey?: PrivateKey;
};
type LazyProof = {
 kind: 'lazy-proof';
 methodName: string;
 args: any[];
 previousProofs: Pickles.Proof[];
 ZkappClass: typeof SmartContract;
 memoized: { fields: Field[]; aux: any[] }[];
 blindingValue: Field;
};

const AccountId = provable({ tokenOwner: PublicKey, parentTokenId: Field });

const TokenId = {
 ...Types.TokenId,
 ...Base58TokenId,
 get default() {
  return Field(1);
 },
 derive(tokenOwner: PublicKey, parentTokenId = Field(1)): Field {
  let input = AccountId.toInput({ tokenOwner, parentTokenId });
  return hashWithPrefix(prefixes.deriveTokenId, packToFields(input));
 },
};

/**
 * @deprecated use &nbsp;TokenId&nbsp; instead of &nbsp;Token.Id&nbsp; and
&nbsp;TokenId.derive()&nbsp; instead of &nbsp;Token.getId()&nbsp;
 */
class Token {
 static Id = TokenId;

 static getId(tokenOwner: PublicKey, parentTokenId = TokenId.default) {
  return TokenId.derive(tokenOwner, parentTokenId);
 }

 readonly id: Field;
 readonly parentTokenId: Field;
 readonly tokenOwner: PublicKey;
 constructor({
  tokenOwner,
  parentTokenId = TokenId.default,
 }: {
  tokenOwner: PublicKey;
  parentTokenId?: Field;
 }) {
  this.parentTokenId = parentTokenId;
  this.tokenOwner = tokenOwner;
  try {
    this.id = TokenId.derive(tokenOwner, parentTokenId);
  } catch (e) {
    throw new Error(
      &nbsp;Could not create a custom token id:\nError: ${(e as Error).message}&nbsp;
    );
  }
 }
}

/**
 * An {@link AccountUpdate} is a set of instructions for the Mina network.
 * It includes {@link Preconditions} and a list of state updates, which need to
 * be authorized by either a {@link Signature} or {@link Proof}.
 */
class AccountUpdate implements Types.AccountUpdate {
 id: number;
 /**
  * A human-readable label for the account update, indicating how that update
  * was created. Can be modified by applications to add richer information.
  */
 label: string = '';
 body: Body;
 authorization: Control;
 lazyAuthorization: LazySignature | LazyProof | LazyNone | undefined =
  undefined;
 account: Precondition.Account;
 network: Precondition.Network;
 currentSlot: Precondition.CurrentSlot;
 children: {
  callsType:
    | { type: 'None' }
    | { type: 'Witness' }
    | { type: 'Equals'; value: Field };
  accountUpdates: AccountUpdate[];
 } = {
  callsType: { type: 'None' },
  accountUpdates: [],
 };
 parent: AccountUpdate | undefined = undefined;

 private isSelf: boolean;

 static Actions = Actions;
```

```
constructor(body: Body, authorization?: Control);
constructor(body: Body, authorization: Control = {}, isSelf = false) {
  this.id = Math.random();
  this.body = body;
  this.authorization = authorization;
  let { account, network, currentSlot } = Precondition.preconditions(
    this,
    isSelf
  );
  this.account = account;
  this.network = network;
  this.currentSlot = currentSlot;
  this.isSelf = isSelf;
}

/**
 * Clones the {@link AccountUpdate}.
 */
static clone(accountUpdate: AccountUpdate) {
  let body = cloneCircuitValue(accountUpdate.body);
  let authorization = cloneCircuitValue(accountUpdate.authorization);
  let cloned: AccountUpdate = new (AccountUpdate as any)(
    body,
    authorization,
    accountUpdate.isSelf
  );
  cloned.lazyAuthorization = accountUpdate.lazyAuthorization;
  cloned.children.callsType = accountUpdate.children.callsType;
  cloned.children.accountUpdates = accountUpdate.children.accountUpdates.map(
    AccountUpdate.clone
  );
  cloned.id = accountUpdate.id;
  cloned.label = accountUpdate.label;
  cloned.parent = accountUpdate.parent;
  return cloned;
}

token() {
  let thisAccountUpdate = this;
  let tokenOwner = this.publicKey;
  let parentTokenId = this.tokenId;
  let id = TokenId.derive(tokenOwner, parentTokenId);

  function getApprovedAccountUpdate(
    accountLike: PublicKey | AccountUpdate | SmartContract,
    label: string
  ) {
    if (accountLike instanceof SmartContract) {
      accountLike = accountLike.self;
    }
    if (accountLike instanceof AccountUpdate) {
      accountLike.tokenId.assertEquals(id);
      thisAccountUpdate.approve(accountLike);
    }
    if (accountLike instanceof PublicKey) {
      accountLike = AccountUpdate.defaultAccountUpdate(accountLike, id);
      makeChildAccountUpdate(thisAccountUpdate, accountLike);
    }
    if (!accountLike.label)
      accountLike.label = &nbsp;${
        thisAccountUpdate.label ?? 'Unlabeled'
      }.${label}&nbsp;;
    return accountLike;
  }

  return {
    id,
    parentTokenId,
    tokenOwner,

    /**
     * Mints token balance to &nbsp;address&nbsp;. Returns the mint account update.
     */
    mint({
      address,
      amount,
    }: {
      address: PublicKey | AccountUpdate | SmartContract;
      amount: number | bigint | UInt64;
    }) {
      let receiver = getApprovedAccountUpdate(address, 'token.mint()');
      receiver.balance.addInPlace(amount);
      return receiver;
    },

    /**
     * Burn token balance on &nbsp;address&nbsp;. Returns the burn account update.
     */
    burn({
      address,
      amount,
    }: {
      address: PublicKey | AccountUpdate | SmartContract;
      amount: number | bigint | UInt64;
    }) {
      let sender = getApprovedAccountUpdate(address, 'token.burn()');

      // Sub the amount to burn from the sender's account
      sender.balance.subInPlace(amount);

      // Require signature from the sender account being deducted
      sender.body.useFullCommitment = Bool(true);
      Authorization.setLazySignature(sender);
      return sender;
    },

    /**
     * Move token balance from &nbsp;from&nbsp; to &nbsp;to&nbsp;. Returns the &nbsp;to&nbsp; account
update.
     */
    send({
      from,
      to,
      amount,
    }: {
      from: PublicKey | AccountUpdate | SmartContract;
      to: PublicKey | AccountUpdate | SmartContract;
      amount: number | bigint | UInt64;
    }) {
      let sender = getApprovedAccountUpdate(from, 'token.send() (sender)');
      sender.balance.subInPlace(amount);
      sender.body.useFullCommitment = Bool(true);
      Authorization.setLazySignature(sender);

      let receiver = getApprovedAccountUpdate(to, 'token.send() (receiver)');
      receiver.balance.addInPlace(amount);

      return receiver;
    },
  };
}

get tokenId() {
  return this.body.tokenId;
}

/**
 * @deprecated use &nbsp;this.account.tokenSymbol&nbsp;
 */
get tokenSymbol() {
  let accountUpdate = this;

  return {
    set(tokenSymbol: string) {
      accountUpdate.account.tokenSymbol.set(tokenSymbol);
    },
  };
}

send({
  to,
  amount,
}: {
  to: PublicKey | AccountUpdate | SmartContract;
  amount: number | bigint | UInt64;
```

```
}) {
  let receiver: AccountUpdate;
  if (to instanceof AccountUpdate) {
    receiver = to;
    receiver.body.tokenId.assertEquals(this.body.tokenId);
  } else if (to instanceof SmartContract) {
    receiver = to.self;
    receiver.body.tokenId.assertEquals(this.body.tokenId);
  } else {
    receiver = AccountUpdate.defaultAccountUpdate(to, this.body.tokenId);
    receiver.label = &nbsp;${this.label ?? 'Unlabeled'}.send()&nbsp;;
    this.approve(receiver);
  }

  // Sub the amount from the sender's account
  this.body.balanceChange = Int64.fromObject(this.body.balanceChange).sub(
    amount
  );
  // Add the amount to the receiver's account
  receiver.body.balanceChange = Int64.fromObject(
    receiver.body.balanceChange
  ).add(amount);
  return receiver;
}

/**
 * Makes an {@link AccountUpdate} a child-{@link AccountUpdate} of this and
 * approves it.
 */
approve(
  childUpdate: AccountUpdate,
  layout: AccountUpdatesLayout = AccountUpdate.Layout.NoChildren
) {
  makeChildAccountUpdate(this, childUpdate);
  AccountUpdate.witnessChildren(childUpdate, layout, { skipCheck: true });
}

get balance() {
  let accountUpdate = this;

  return {
    addInPlace(x: Int64 | UInt32 | UInt64 | string | number | bigint) {
      let { magnitude, sgn } = accountUpdate.body.balanceChange;
      accountUpdate.body.balanceChange = new Int64(magnitude, sgn).add(x);
    },
    subInPlace(x: Int64 | UInt32 | UInt64 | string | number | bigint) {
      let { magnitude, sgn } = accountUpdate.body.balanceChange;
      accountUpdate.body.balanceChange = new Int64(magnitude, sgn).sub(x);
    },
  };
}
```

```
get update(): Update {
  return this.body.update;
}

static setValue<T>(maybeValue: SetOrKeep<T>, value: T) {
  maybeValue.isSome = Bool(true);
  maybeValue.value = value;
}

/**
 * Constrain a property to lie between lower and upper bounds.
 *
 * @param property The property to constrain
 * @param lower The lower bound
 * @param upper The upper bound
 *
 * Example: To constrain the account balance of a SmartContract to lie between
 * 0 and 20 MINA, you can use
 *
 * &nbsp;&nbsp;&nbsp;ts
 * \@method onlyRunsWhenBalanceIsLow() {
 *   let lower = UInt64.zero;
 *   let upper = UInt64.from(20e9);
 *   AccountUpdate.assertBetween(this.self.body.preconditions.account.balance, lower, upper);
 *   // ...
 * }
 * &nbsp;&nbsp;&nbsp;
 */
static assertBetween<T>(
  property: OrIgnore<ClosedInterval<T>>,
  lower: T,
  upper: T
) {
  property.isSome = Bool(true);
  property.value.lower = lower;
  property.value.upper = upper;
}

// TODO: assertGreaterThan, assertLowerThan?

/**
 * Fix a property to a certain value.
 *
 * @param property The property to constrain
 * @param value The value it is fixed to
 *
 * Example: To fix the account nonce of a SmartContract to 0, you can use
 *
 * &nbsp;&nbsp;&nbsp;ts
 * \@method onlyRunsWhenNonceIsZero() {
 *   AccountUpdate.assertEquals(this.self.body.preconditions.account.nonce, UInt32.zero);
 *   // ...
```

```
  * }
  * &nbsp;&nbsp;&nbsp;
  */
 static assertEquals<T extends object>(
   property: OrIgnore<ClosedInterval<T> | T>,
   value: T
 ) {
   property.isSome = Bool(true);
   if ('lower' in property.value && 'upper' in property.value) {
     property.value.lower = value;
     property.value.upper = value;
   } else {
     property.value = value;
   }
 }

 get publicKey(): PublicKey {
   return this.body.publicKey;
 }

 /**
  * Use this command if this account update should be signed by the account
  * owner, instead of not having any authorization.
  *
  * If you use this and are not relying on a wallet to sign your transaction,
  * then you should use the following code before sending your transaction:
  *
  * &nbsp;&nbsp;&nbsp;ts
  * let tx = Mina.transaction(...); // create transaction as usual, using &nbsp;requireSignature()&nbsp;
somewhere
  * tx.sign([privateKey]); // pass the private key of this account to &nbsp;sign()&nbsp;!
  * &nbsp;&nbsp;&nbsp;
  *
  * Note that an account's {@link Permissions} determine which updates have to
  * be (can be) authorized by a signature.
  */
 requireSignature() {
   this.sign();
 }
 /**
  * @deprecated &nbsp;.sign()&nbsp; is deprecated in favor of &nbsp;.requireSignature()&nbsp;
  */
 sign(privateKey?: PrivateKey) {
   let { nonce, isSameAsFeePayer } = AccountUpdate.getSigningInfo(this);
   // if this account is the same as the fee payer, we use the "full commitment" for replay protection
   this.body.useFullCommitment = isSameAsFeePayer;
   this.body.implicitAccountCreationFee = Bool(false);
   // otherwise, we increment the nonce
   let doIncrementNonce = isSameAsFeePayer.not();
   this.body.incrementNonce = doIncrementNonce;
   // in this case, we also have to set a nonce precondition
   let lower = Provable.if(doIncrementNonce, UInt32, nonce, UInt32.zero);
  let upper = Provable.if(doIncrementNonce, UInt32, nonce, UInt32.MAXINT());
  this.body.preconditions.account.nonce.isSome = doIncrementNonce;
  this.body.preconditions.account.nonce.value.lower = lower;
  this.body.preconditions.account.nonce.value.upper = upper;
  // set lazy signature
  Authorization.setLazySignature(this, { privateKey });
}

static signFeePayerInPlace(
  feePayer: FeePayerUnsigned,
  privateKey?: PrivateKey
) {
  feePayer.body.nonce = this.getNonce(feePayer);
  feePayer.authorization = dummySignature();
  feePayer.lazyAuthorization = { kind: 'lazy-signature', privateKey };
}

static getNonce(accountUpdate: AccountUpdate | FeePayerUnsigned) {
  return AccountUpdate.getSigningInfo(accountUpdate).nonce;
}

private static signingInfo = provable({
  isSameAsFeePayer: Bool,
  nonce: UInt32,
});

private static getSigningInfo(
  accountUpdate: AccountUpdate | FeePayerUnsigned
) {
  return memoizeWitness(AccountUpdate.signingInfo, () =>
    AccountUpdate.getSigningInfoUnchecked(accountUpdate)
  );
}

private static getSigningInfoUnchecked(
  update: AccountUpdate | FeePayerUnsigned
) {
  let publicKey = update.body.publicKey;
  let tokenId =
    update instanceof AccountUpdate ? update.body.tokenId : TokenId.default;
  let nonce = Number(
    Precondition.getAccountPreconditions(update.body).nonce.toString()
  );
  // if the fee payer is the same account update as this one, we have to start
  // the nonce predicate at one higher, bc the fee payer already increases its
  // nonce
  let isFeePayer = Mina.currentTransaction()?.sender?.equals(publicKey);
  let isSameAsFeePayer = !!isFeePayer
    ?.and(tokenId.equals(TokenId.default))
    .toBoolean();
  if (isSameAsFeePayer) nonce++;
  // now, we check how often this account update already updated its nonce in
  // this tx, and increase nonce from &nbsp;getAccount&nbsp; by that amount
  CallForest.forEachPredecessor(
    Mina.currentTransaction.get().accountUpdates,
    update as AccountUpdate,
    (otherUpdate) => {
      let shouldIncreaseNonce = otherUpdate.publicKey
        .equals(publicKey)
        .and(otherUpdate.tokenId.equals(tokenId))
        .and(otherUpdate.body.incrementNonce);
      if (shouldIncreaseNonce.toBoolean()) nonce++;
    }
  );
  return {
    nonce: UInt32.from(nonce),
    isSameAsFeePayer: Bool(isSameAsFeePayer),
  };
}

toJSON() {
  return Types.AccountUpdate.toJSON(this);
}
static toJSON(a: AccountUpdate) {
  return Types.AccountUpdate.toJSON(a);
}
static fromJSON(json: Types.Json.AccountUpdate) {
  let accountUpdate = Types.AccountUpdate.fromJSON(json);
  return new AccountUpdate(accountUpdate.body, accountUpdate.authorization);
}

hash(): Field {
  // these two ways of hashing are (and have to be) consistent / produce the same hash
  // TODO: there's no reason anymore to use two different hashing methods here!
  // -- the "inCheckedComputation" branch works in all circumstances now
  // we just leave this here for a couple more weeks, because it checks
  // consistency between JS & OCaml hashing on *every single account update
  // proof* we create. It will give us 100% confidence that the two
  // implementations are equivalent, and catch regressions quickly
  if (Provable.inCheckedComputation()) {
    let input = Types.AccountUpdate.toInput(this);
    return hashWithPrefix(prefixes.body, packToFields(input));
  } else {
    let json = Types.AccountUpdate.toJSON(this);
    return Field(Test.hashFromJson.accountUpdate(JSON.stringify(json)));
  }
}

toPublicInput(): ZkappPublicInput {
  let accountUpdate = this.hash();
  let calls = CallForest.hashChildren(this);
  return { accountUpdate, calls };
}
```

```
static defaultAccountUpdate(address: PublicKey, tokenId?: Field) {
  return new AccountUpdate(Body.keepAll(address, tokenId));
}
static dummy() {
  return new AccountUpdate(Body.dummy());
}
isDummy() {
  return this.body.publicKey.isEmpty();
}

static defaultFeePayer(address: PublicKey, nonce: UInt32): FeePayerUnsigned {
  let body = FeePayerBody.keepAll(address, nonce);
  return {
    body,
    authorization: dummySignature(),
    lazyAuthorization: { kind: 'lazy-signature' },
  };
}

static dummyFeePayer(): FeePayerUnsigned {
  let body = FeePayerBody.keepAll(PublicKey.empty(), UInt32.zero);
  return { body, authorization: dummySignature() };
}

/**
 * Creates an account update. If this is inside a transaction, the account
 * update becomes part of the transaction. If this is inside a smart contract
 * method, the account update will not only become part of the transaction,
 * but also becomes available for the smart contract to modify, in a way that
 * becomes part of the proof.
 */
static create(publicKey: PublicKey, tokenId?: Field) {
  let accountUpdate = AccountUpdate.defaultAccountUpdate(publicKey, tokenId);
  let insideContract = smartContractContext.get();
  if (insideContract) {
    let self = insideContract.this.self;
    self.approve(accountUpdate);
    accountUpdate.label = &nbsp;${
      self.label || 'Unlabeled'
    } > AccountUpdate.create()&nbsp;;
  } else {
    Mina.currentTransaction()?.accountUpdates.push(accountUpdate);
    accountUpdate.label = &nbsp;Mina.transaction > AccountUpdate.create()&nbsp;;
  }
  return accountUpdate;
}
/**
 * Attach account update to the current transaction
 * -- if in a smart contract, to its children
 */
static attachToTransaction(accountUpdate: AccountUpdate) {
  let insideContract = smartContractContext.get();
  if (insideContract) {
    let selfUpdate = insideContract.this.self;
    // avoid redundant attaching & cycle in account update structure, happens
    // when calling attachToTransaction(this.self) inside a @method
    // TODO avoid account update cycles more generally
    if (selfUpdate === accountUpdate) return;
    insideContract.this.self.approve(accountUpdate);
  } else {
    if (!Mina.currentTransaction.has()) return;
    let updates = Mina.currentTransaction.get().accountUpdates;
    if (!updates.find((update) => update.id === accountUpdate.id)) {
      updates.push(accountUpdate);
    }
  }
}
/**
 * Disattach an account update from where it's currently located in the transaction
 */
static unlink(accountUpdate: AccountUpdate) {
  let siblings =
    accountUpdate.parent?.children.accountUpdates ??
    Mina.currentTransaction()?.accountUpdates;
  if (siblings === undefined) return;
  let i = siblings?.findIndex((update) => update.id === accountUpdate.id);
  if (i !== undefined && i !== -1) {
    siblings!.splice(i, 1);
  }
  accountUpdate.parent === undefined;
}

/**
 * Creates an account update, like {@link AccountUpdate.create}, but also
 * makes sure this account update will be authorized with a signature.
 *
 * If you use this and are not relying on a wallet to sign your transaction,
 * then you should use the following code before sending your transaction:
 *
 * &nbsp;&nbsp;&nbsp;ts
 * let tx = Mina.transaction(...); // create transaction as usual, using &nbsp;createSigned()&nbsp;
somewhere
 * tx.sign([privateKey]); // pass the private key of this account to &nbsp;sign()&nbsp;!
 * &nbsp;&nbsp;&nbsp;
 *
 * Note that an account's {@link Permissions} determine which updates have to
 * be (can be) authorized by a signature.
 */
static createSigned(signer: PublicKey, tokenId?: Field): AccountUpdate;
/**
 * @deprecated in favor of calling this function with a &nbsp;PublicKey&nbsp; as &nbsp;signer&nbsp;
 */
static createSigned(signer: PrivateKey, tokenId?: Field): AccountUpdate;
static createSigned(signer: PrivateKey | PublicKey, tokenId?: Field) {
  let publicKey =
    signer instanceof PrivateKey ? signer.toPublicKey() : signer;
  let accountUpdate = AccountUpdate.create(publicKey, tokenId);
  accountUpdate.label = accountUpdate.label.replace(
    '.create()',
    '.createSigned()'
  );
  if (signer instanceof PrivateKey) {
    accountUpdate.sign(signer);
  } else {
    accountUpdate.requireSignature();
  }
  return accountUpdate;
}

/**
 * Use this method to pay the account creation fee for another account (or, multiple accounts using the
optional second argument).
 *
 * Beware that you _don't_ need to specify the account that is created!
 * Instead, the protocol will automatically identify that accounts need to be created,
 * and require that the net balance change of the transaction covers the account creation fee.
 *
 * @param feePayer the address of the account that pays the fee
 * @param numberOfAccounts the number of new accounts to fund (default: 1)
 * @returns they {@link AccountUpdate} for the account which pays the fee
 */
static fundNewAccount(
  feePayer: PublicKey,
  numberOfAccounts?: number
): AccountUpdate;
/**
 * @deprecated Call this function with a &nbsp;PublicKey&nbsp; as &nbsp;feePayer&nbsp;, and remove
the &nbsp;initialBalance&nbsp; option.
 * To send an initial balance to the new account, you can use the returned account update:
 * &nbsp;&nbsp;&nbsp;
 * let feePayerUpdate = AccountUpdate.fundNewAccount(feePayer);
 * feePayerUpdate.send({ to: receiverAddress, amount: initialBalance });
 * &nbsp;&nbsp;&nbsp;
 */
static fundNewAccount(
  feePayer: PrivateKey | PublicKey,
  options?: { initialBalance: number | string | UInt64 } | number
): AccountUpdate;
static fundNewAccount(
  feePayer: PrivateKey | PublicKey,
  numberOfAccounts?: number | { initialBalance: number | string | UInt64 }
) {
  let accountUpdate = AccountUpdate.createSigned(feePayer as PrivateKey);
  accountUpdate.label = 'AccountUpdate.fundNewAccount()';
  let fee = Mina.accountCreationFee();
  numberOfAccounts ??= 1;
  if (typeof numberOfAccounts === 'number') fee = fee.mul(numberOfAccounts);
  else fee = fee.add(UInt64.from(numberOfAccounts.initialBalance ?? 0));
  accountUpdate.balance.subInPlace(fee);
  return accountUpdate;
}

// static methods that implement Provable<AccountUpdate>
static sizeInFields = Types.AccountUpdate.sizeInFields;
static toFields = Types.AccountUpdate.toFields;
static toAuxiliary(a?: AccountUpdate) {
  let aux = Types.AccountUpdate.toAuxiliary(a);
  let children: AccountUpdate['children'] = {
    callsType: { type: 'None' },
    accountUpdates: [],
  };
  let lazyAuthorization = a && a.lazyAuthorization;
  if (a) {
    children.callsType = a.children.callsType;
    children.accountUpdates = a.children.accountUpdates.map(
      AccountUpdate.clone
    );
  }
  let parent = a?.parent;
  let id = a?.id ?? Math.random();
  let label = a?.label ?? '';
  return [{ lazyAuthorization, children, parent, id, label }, aux];
}
static toInput = Types.AccountUpdate.toInput;
static check = Types.AccountUpdate.check;
static fromFields(fields: Field[], [other, aux]: any[]): AccountUpdate {
  let accountUpdate = Types.AccountUpdate.fromFields(fields, aux);
  return Object.assign(
    new AccountUpdate(accountUpdate.body, accountUpdate.authorization),
    other
  );
}

static witness<T>(
  type: FlexibleProvable<T>,
  compute: () => { accountUpdate: AccountUpdate; result: T },
  { skipCheck = false } = {}
) {
  // construct the circuit type for a accountUpdate + other result
  let accountUpdateType = skipCheck
    ? { ...provable(AccountUpdate), check() {} }
    : AccountUpdate;
  let combinedType = provable({
    accountUpdate: accountUpdateType,
    result: type as any,
  });
  return Provable.witness(combinedType, compute);
}
```

```
static witnessChildren(
  accountUpdate: AccountUpdate,
  childLayout: AccountUpdatesLayout,
  options?: { skipCheck: boolean }
) {
  // just witness children's hash if childLayout === null
  if (childLayout === AccountUpdate.Layout.AnyChildren) {
    accountUpdate.children.callsType = { type: 'Witness' };
    return;
  }
  if (childLayout === AccountUpdate.Layout.NoDelegation) {
    accountUpdate.children.callsType = { type: 'Witness' };
    accountUpdate.body.mayUseToken.parentsOwnToken.assertFalse();
    accountUpdate.body.mayUseToken.inheritFromParent.assertFalse();
    return;
  }
  let childArray: AccountUpdatesLayout[] =
    typeof childLayout === 'number'
      ? Array(childLayout).fill(AccountUpdate.Layout.NoChildren)
      : childLayout;
  let n = childArray.length;
  for (let i = 0; i < n; i++) {
    accountUpdate.children.accountUpdates[i] = AccountUpdate.witnessTree(
      provable(null),
      childArray[i],
      () => ({
        accountUpdate:
          accountUpdate.children.accountUpdates[i] ?? AccountUpdate.dummy(),
        result: null,
      }),
      options
    ).accountUpdate;
  }
  if (n === 0) {
    accountUpdate.children.callsType = {
      type: 'Equals',
      value: CallForest.emptyHash(),
    };
  }
}

/**
 * Like AccountUpdate.witness, but lets you specify a layout for the
 * accountUpdate's children, which also get witnessed
 */
static witnessTree<T>(
  resultType: FlexibleProvable<T>,
  childLayout: AccountUpdatesLayout,
  compute: () => {
    accountUpdate: AccountUpdate;
    result: T;
  },
  options?: { skipCheck: boolean }
 ) {
  // witness the root accountUpdate
  let { accountUpdate, result } = AccountUpdate.witness(
    resultType,
    compute,
    options
  );
  // witness child account updates
  AccountUpdate.witnessChildren(accountUpdate, childLayout, options);
  return { accountUpdate, result };
 }

 /**
  * Describes the children of an account update, which are laid out in a tree.
  *
  * The tree layout is described recursively by using a combination of
&nbsp;AccountUpdate.Layout.NoChildren&nbsp;, &nbsp;AccountUpdate.Layout.StaticChildren(...)&nbsp;
and &nbsp;AccountUpdate.Layout.AnyChildren&nbsp;.
  * - &nbsp;NoChildren&nbsp; means an account update that can't have children
  * - &nbsp;AnyChildren&nbsp; means an account update can have an arbitrary amount of children, which
means you can't access those children in your circuit (because the circuit is static).
  * - &nbsp;StaticChildren&nbsp; means the account update must have a certain static amount of children
and expects as arguments a description of each of those children.
  *   As a shortcut, you can also pass &nbsp;StaticChildren&nbsp; a number, which means it has that
amount of children but no grandchildren.
  *
  * This is best understood by examples:
  *
  * &nbsp;&nbsp;&nbsp;ts
  * let { NoChildren, AnyChildren, StaticChildren } = AccounUpdate.Layout;
  *
  * NoChildren                 // an account update with no children
  * AnyChildren                // an account update with arbitrary children
  * StaticChildren(NoChildren) // an account update with 1 child, which doesn't have children itself
  * StaticChildren(1)          // shortcut for StaticChildren(NoChildren)
  * StaticChildren(2)          // shortcut for StaticChildren(NoChildren, NoChildren)
  * StaticChildren(0)          // equivalent to NoChildren
  *
  * // an update with 2 children, of which one has arbitrary children and the other has exactly 1 descendant
  * StaticChildren(AnyChildren, StaticChildren(1))
  * &nbsp;&nbsp;&nbsp;
  */
 static Layout = {
  StaticChildren: ((...args: any[]) => {
    if (args.length === 1 && typeof args[0] === 'number') return args[0];
    if (args.length === 0) return 0;
    return args;
  }) as {
    (n: number): AccountUpdatesLayout;
    (...args: AccountUpdatesLayout[]): AccountUpdatesLayout;
  },
  NoChildren: 0,
  AnyChildren: 'AnyChildren' as const,
  NoDelegation: 'NoDelegation' as const,
};

static get MayUseToken() {
  return {
    type: provablePure({ parentsOwnToken: Bool, inheritFromParent: Bool }),
    No: { parentsOwnToken: Bool(false), inheritFromParent: Bool(false) },
    ParentsOwnToken: {
      parentsOwnToken: Bool(true),
      inheritFromParent: Bool(false),
    },
    InheritFromParent: {
      parentsOwnToken: Bool(false),
      inheritFromParent: Bool(true),
    },
    isNo({
      body: {
        mayUseToken: { parentsOwnToken, inheritFromParent },
      },
    }: AccountUpdate) {
      return parentsOwnToken.or(inheritFromParent).not();
    },
    isParentsOwnToken(a: AccountUpdate) {
      return a.body.mayUseToken.parentsOwnToken;
    },
    isInheritFromParent(a: AccountUpdate) {
      return a.body.mayUseToken.inheritFromParent;
    },
  };
}

/**
 * Returns a JSON representation of only the fields that differ from the
 * default {@link AccountUpdate}.
 */
toPretty() {
  function short(s: string) {
    return '..' + s.slice(-4);
  }
  let jsonUpdate: Partial<Types.Json.AccountUpdate> = toJSONEssential(
    jsLayout.AccountUpdate as any,
    this
  );
  let body: Partial<Types.Json.AccountUpdate['body']> =
    jsonUpdate.body as any;
  delete body.callData;
  body.publicKey = short(body.publicKey!);
  if (body.balanceChange?.magnitude === '0') delete body.balanceChange;
  if (body.tokenId === TokenId.toBase58(TokenId.default)) {
```

```
  delete body.tokenId;
} else {
  body.tokenId = short(body.tokenId!);
}
if (body.callDepth === 0) delete body.callDepth;
if (body.incrementNonce === false) delete body.incrementNonce;
if (body.useFullCommitment === false) delete body.useFullCommitment;
if (body.implicitAccountCreationFee === false)
  delete body.implicitAccountCreationFee;
if (body.events?.length === 0) delete body.events;
if (body.actions?.length === 0) delete body.actions;
if (body.preconditions?.account) {
  body.preconditions.account = JSON.stringify(
    body.preconditions.account
  ) as any;
}
if (body.preconditions?.network) {
  body.preconditions.network = JSON.stringify(
    body.preconditions.network
  ) as any;
}
if (body.preconditions?.validWhile) {
  body.preconditions.validWhile = JSON.stringify(
    body.preconditions.validWhile
  ) as any;
}
if (jsonUpdate.authorization?.proof) {
  jsonUpdate.authorization.proof = short(jsonUpdate.authorization.proof);
}
if (jsonUpdate.authorization?.signature) {
  jsonUpdate.authorization.signature = short(
    jsonUpdate.authorization.signature
  );
}
if (body.update?.verificationKey) {
  body.update.verificationKey = JSON.stringify({
    data: short(body.update.verificationKey.data),
    hash: short(body.update.verificationKey.hash),
  }) as any;
}
for (let key of ['permissions', 'appState', 'timing'] as const) {
  if (body.update?.[key]) {
    body.update[key] = JSON.stringify(body.update[key]) as any;
  }
}
for (let key of ['events', 'actions'] as const) {
  if (body[key]) {
    body[key] = JSON.stringify(body[key]) as any;
  }
}
if (
  jsonUpdate.authorization !== undefined ||
      body.authorizationKind?.isProved === true ||
      body.authorizationKind?.isSigned === true
    ) {
      (body as any).authorization = jsonUpdate.authorization;
    }
    body.mayUseToken = {
      parentsOwnToken: this.body.mayUseToken.parentsOwnToken.toBoolean(),
      inheritFromParent: this.body.mayUseToken.inheritFromParent.toBoolean(),
    };
    let pretty: any = { ...body };
    let withId = false;
    if (withId) pretty = { id: Math.floor(this.id * 1000), ...pretty };
    if (this.label) pretty = { label: this.label, ...pretty };
    return pretty;
  }
}

type AccountUpdatesLayout =
  | number
  | 'AnyChildren'
  | 'NoDelegation'
  | AccountUpdatesLayout[];

type WithCallers = {
  accountUpdate: AccountUpdate;
  caller: Field;
  children: WithCallers[];
};

const CallForest = {
  // similar to Mina_base.ZkappCommand.Call_forest.to_account_updates_list
  // takes a list of accountUpdates, which each can have children, so they form a "forest" (list of trees)
  // returns a flattened list, with &nbsp;accountUpdate.body.callDepth&nbsp; specifying positions in the forest
  // also removes any "dummy" accountUpdates
  toFlatList(
    forest: AccountUpdate[],
    mutate = true,
    depth = 0
  ): AccountUpdate[] {
    let accountUpdates = [];
    for (let accountUpdate of forest) {
      if (accountUpdate.isDummy().toBoolean()) continue;
      if (mutate) accountUpdate.body.callDepth = depth;
      let children = accountUpdate.children.accountUpdates;
      accountUpdates.push(
        accountUpdate,
        ...CallForest.toFlatList(children, mutate, depth + 1)
      );
    }
    return accountUpdates;
  },
// Mina_base.Zkapp_command.Digest.Forest.empty
emptyHash() {
  return Field(0);
},

// similar to Mina_base.Zkapp_command.Call_forest.accumulate_hashes
// hashes a accountUpdate's children (and their children, and ...) to compute
// the &nbsp;calls&nbsp; field of ZkappPublicInput
hashChildren(update: AccountUpdate): Field {
  let { callsType } = update.children;
  // compute hash outside the circuit if callsType is "Witness"
  // i.e., allowing accountUpdates with arbitrary children
  if (callsType.type === 'Witness') {
    return Provable.witness(Field, () => CallForest.hashChildrenBase(update));
  }
  let calls = CallForest.hashChildrenBase(update);
  if (callsType.type === 'Equals' && Provable.inCheckedComputation()) {
    calls.assertEquals(callsType.value);
  }
  return calls;
},

hashChildrenBase({ children }: AccountUpdate) {
  let stackHash = CallForest.emptyHash();
  for (let accountUpdate of [...children.accountUpdates].reverse()) {
    let calls = CallForest.hashChildren(accountUpdate);
    let nodeHash = hashWithPrefix(prefixes.accountUpdateNode, [
      accountUpdate.hash(),
      calls,
    ]);
    let newHash = hashWithPrefix(prefixes.accountUpdateCons, [
      nodeHash,
      stackHash,
    ]);
    // skip accountUpdate if it's a dummy
    stackHash = Provable.if(accountUpdate.isDummy(), stackHash, newHash);
  }
  return stackHash;
},

// Mina_base.Zkapp_command.Call_forest.add_callers
// TODO: currently unused, but could come back when we add caller to the
// public input
addCallers(
  updates: AccountUpdate[],
  context: { self: Field; caller: Field } = {
    self: TokenId.default,
    caller: TokenId.default,
  }
): WithCallers[] {
  let withCallers: WithCallers[] = [];
  for (let update of updates) {
    let { mayUseToken } = update.body;
    let caller = Provable.if(
      mayUseToken.parentsOwnToken,
      context.self,
      Provable.if(
        mayUseToken.inheritFromParent,
        context.caller,
        TokenId.default
      )
    );
    let self = TokenId.derive(update.body.publicKey, update.body.tokenId);
    let childContext = { caller, self };
    withCallers.push({
      accountUpdate: update,
      caller,
      children: CallForest.addCallers(
        update.children.accountUpdates,
        childContext
      ),
    });
  }
  return withCallers;
},
/**
 * Used in the prover to witness the context from which to compute its caller
 *
 * TODO: currently unused, but could come back when we add caller to the
 * public input
 */
computeCallerContext(update: AccountUpdate) {
  // compute the line of ancestors
  let current = update;
  let ancestors = [];
  while (true) {
    let parent = current.parent;
    if (parent === undefined) break;
    ancestors.unshift(parent);
    current = parent;
  }
  let context = { self: TokenId.default, caller: TokenId.default };
  for (let update of ancestors) {
    if (update.body.mayUseToken.parentsOwnToken.toBoolean()) {
      context.caller = context.self;
    } else if (!update.body.mayUseToken.inheritFromParent.toBoolean()) {
      context.caller = TokenId.default;
    }
    context.self = TokenId.derive(update.body.publicKey, update.body.tokenId);
  }
  return context;
},
callerContextType: provablePure({ self: Field, caller: Field }),
  computeCallDepth(update: AccountUpdate) {
    for (let callDepth = 0; ; callDepth++) {
      if (update.parent === undefined) return callDepth;
      update = update.parent;
    }
  },

  map(updates: AccountUpdate[], map: (update: AccountUpdate) => AccountUpdate) {
    let newUpdates: AccountUpdate[] = [];
    for (let update of updates) {
      let newUpdate = map(update);
      newUpdate.children.accountUpdates = CallForest.map(
        update.children.accountUpdates,
        map
      );
      newUpdates.push(newUpdate);
    }
    return newUpdates;
  },

  forEach(updates: AccountUpdate[], callback: (update: AccountUpdate) => void) {
    for (let update of updates) {
      callback(update);
      CallForest.forEach(update.children.accountUpdates, callback);
    }
  },

  forEachPredecessor(
    updates: AccountUpdate[],
    update: AccountUpdate,
    callback: (update: AccountUpdate) => void
  ) {
    let isPredecessor = true;
    CallForest.forEach(updates, (otherUpdate) => {
      if (otherUpdate.id === update.id) isPredecessor = false;
      if (isPredecessor) callback(otherUpdate);
    });
  },
};

function createChildAccountUpdate(
  parent: AccountUpdate,
  childAddress: PublicKey,
  tokenId?: Field
) {
  let child = AccountUpdate.defaultAccountUpdate(childAddress, tokenId);
  makeChildAccountUpdate(parent, child);
  return child;
}
function makeChildAccountUpdate(parent: AccountUpdate, child: AccountUpdate) {
  child.body.callDepth = parent.body.callDepth + 1;
  let wasChildAlready = parent.children.accountUpdates.find(
    (update) => update.id === child.id
  );
  // add to our children if not already here
  if (!wasChildAlready) {
    parent.children.accountUpdates.push(child);
    // remove the child from the top level list / its current parent
    AccountUpdate.unlink(child);
  }
  child.parent = parent;
}

// authorization

type ZkappCommand = {
  feePayer: FeePayerUnsigned;
  accountUpdates: AccountUpdate[];
  memo: string;
};
type ZkappCommandSigned = {
  feePayer: FeePayer;
  accountUpdates: (AccountUpdate & { lazyAuthorization?: LazyProof })[];
  memo: string;
};
type ZkappCommandProved = {
  feePayer: FeePayerUnsigned;
  accountUpdates: (AccountUpdate & { lazyAuthorization?: LazySignature })[];
  memo: string;
};

const ZkappCommand = {
  toPretty(transaction: ZkappCommand) {
    let feePayer = ZkappCommand.toJSON(transaction).feePayer as any;
    feePayer.body.publicKey = '..' + feePayer.body.publicKey.slice(-4);
    feePayer.body.authorization = '..' + feePayer.authorization.slice(-4);
    if (feePayer.body.validUntil === null) delete feePayer.body.validUntil;
    return [
      { label: 'feePayer', ...feePayer.body },
      ...transaction.accountUpdates.map((a) => a.toPretty()),
    ];
  },
  fromJSON(json: Types.Json.ZkappCommand): ZkappCommand {
    let { feePayer } = Types.ZkappCommand.fromJSON({
      feePayer: json.feePayer,
      accountUpdates: [],
      memo: json.memo,
    });
    let memo = Memo.toString(Memo.fromBase58(json.memo));
    let accountUpdates = json.accountUpdates.map(AccountUpdate.fromJSON);
    return { feePayer, accountUpdates, memo };
  },
  toJSON({ feePayer, accountUpdates, memo }: ZkappCommand) {
    memo = Memo.toBase58(Memo.fromString(memo));
    return Types.ZkappCommand.toJSON({ feePayer, accountUpdates, memo });
  },
};

type AccountUpdateProved = AccountUpdate & {
  lazyAuthorization?: LazySignature;
};

const Authorization = {
  hasLazyProof(accountUpdate: AccountUpdate) {
    return accountUpdate.lazyAuthorization?.kind === 'lazy-proof';
  },
  hasAny(accountUpdate: AccountUpdate) {
    let { authorization: auth, lazyAuthorization: lazyAuth } = accountUpdate;
    return !!(lazyAuth || 'proof' in auth || 'signature' in auth);
  },
  setSignature(accountUpdate: AccountUpdate, signature: string) {
    accountUpdate.authorization = { signature };
    accountUpdate.lazyAuthorization = undefined;
  },
  setProof(accountUpdate: AccountUpdate, proof: string): AccountUpdateProved {
    accountUpdate.authorization = { proof };
    accountUpdate.lazyAuthorization = undefined;
    return accountUpdate as AccountUpdateProved;
  },
  setLazySignature(
    accountUpdate: AccountUpdate,
    signature?: Omit<LazySignature, 'kind'>
  ) {
    signature ??= {};
    accountUpdate.body.authorizationKind.isSigned = Bool(true);
    accountUpdate.body.authorizationKind.isProved = Bool(false);
    accountUpdate.body.authorizationKind.verificationKeyHash = Field(
      mocks.dummyVerificationKeyHash
    );
    accountUpdate.authorization = {};
    accountUpdate.lazyAuthorization = { ...signature, kind: 'lazy-signature' };
  },
  setProofAuthorizationKind(
    { body, id }: AccountUpdate,
    priorAccountUpdates?: AccountUpdate[]
  ) {
    body.authorizationKind.isSigned = Bool(false);
    body.authorizationKind.isProved = Bool(true);
    let hash = Provable.witness(Field, () => {
      let proverData = zkAppProver.getData();
      let isProver = proverData !== undefined;
      assert(
        isProver || priorAccountUpdates !== undefined,
        'Called &nbsp;setProofAuthorizationKind()&nbsp; outside the prover without passing in
&nbsp;priorAccountUpdates&nbsp;.'
      );
      let myAccountUpdateId = isProver ? proverData.accountUpdate.id : id;
      priorAccountUpdates ??= proverData.transaction.accountUpdates;
      priorAccountUpdates = priorAccountUpdates.filter(
        (a) => a.id !== myAccountUpdateId
      );
      let priorAccountUpdatesFlat = CallForest.toFlatList(
        priorAccountUpdates,
        false
      );
      let accountUpdate = [...priorAccountUpdatesFlat]
        .reverse()
        .find((body_) =>
          body_.update.verificationKey.isSome
            .and(body_.tokenId.equals(body.tokenId))
            .and(body_.publicKey.equals(body.publicKey))
            .toBoolean()
        );
      if (accountUpdate !== undefined) {
        return accountUpdate.body.update.verificationKey.value.hash;
      }
      try {
        let account = Mina.getAccount(body.publicKey, body.tokenId);
        return account.zkapp?.verificationKey?.hash ?? Field(0);
      } catch {
        return Field(0);
      }
    });
    body.authorizationKind.verificationKeyHash = hash;
  },
  setLazyProof(
    accountUpdate: AccountUpdate,
    proof: Omit<LazyProof, 'kind'>,
    priorAccountUpdates: AccountUpdate[]
  ) {
    Authorization.setProofAuthorizationKind(accountUpdate, priorAccountUpdates);
    accountUpdate.authorization = {};
    accountUpdate.lazyAuthorization = { ...proof, kind: 'lazy-proof' };
  },
  setLazyNone(accountUpdate: AccountUpdate) {
    accountUpdate.body.authorizationKind.isSigned = Bool(false);
    accountUpdate.body.authorizationKind.isProved = Bool(false);
    accountUpdate.body.authorizationKind.verificationKeyHash = Field(
      mocks.dummyVerificationKeyHash
    );
    accountUpdate.authorization = {};
    accountUpdate.lazyAuthorization = { kind: 'lazy-none' };
  },
};

function addMissingSignatures(
  zkappCommand: ZkappCommand,
  additionalKeys = [] as PrivateKey[]
```

```
): ZkappCommandSigned {
  let additionalPublicKeys = additionalKeys.map((sk) => sk.toPublicKey());
  let { commitment, fullCommitment } = transactionCommitments(
    TypesBigint.ZkappCommand.fromJSON(ZkappCommand.toJSON(zkappCommand))
  );

  function addFeePayerSignature(accountUpdate: FeePayerUnsigned): FeePayer {
    let { body, authorization, lazyAuthorization } =
      cloneCircuitValue(accountUpdate);
    if (lazyAuthorization === undefined) return { body, authorization };
    let { privateKey } = lazyAuthorization;
    if (privateKey === undefined) {
      let i = additionalPublicKeys.findIndex((pk) =>
        pk.equals(accountUpdate.body.publicKey).toBoolean()
      );
      if (i === -1) {
        // private key is missing, but we are not throwing an error here
        // there is a change signature will be added by the wallet
        // if not, error will be thrown by verifyAccountUpdate
        // while .send() execution
        return { body, authorization: dummySignature() };
      }
      privateKey = additionalKeys[i];
    }
    let signature = signFieldElement(
      fullCommitment,
      privateKey.toBigInt(),
      'testnet'
    );
    return { body, authorization: Signature.toBase58(signature) };
  }

  function addSignature(accountUpdate: AccountUpdate) {
    accountUpdate = AccountUpdate.clone(accountUpdate);
    if (accountUpdate.lazyAuthorization?.kind !== 'lazy-signature') {
      return accountUpdate as AccountUpdate & { lazyAuthorization?: LazyProof };
    }
    let { privateKey } = accountUpdate.lazyAuthorization;
    if (privateKey === undefined) {
      let i = additionalPublicKeys.findIndex((pk) =>
        pk.equals(accountUpdate.body.publicKey).toBoolean()
      );
      if (i === -1) {
        // private key is missing, but we are not throwing an error here
        // there is a change signature will be added by the wallet
        // if not, error will be thrown by verifyAccountUpdate
        // while .send() execution
        Authorization.setSignature(accountUpdate, dummySignature());
        return accountUpdate as AccountUpdate & {
          lazyAuthorization: undefined;
        };
      }
      privateKey = additionalKeys[i];
    }
    let transactionCommitment = accountUpdate.body.useFullCommitment.toBoolean()
      ? fullCommitment
      : commitment;
    let signature = signFieldElement(
      transactionCommitment,
      privateKey.toBigInt(),
      'testnet'
    );
    Authorization.setSignature(accountUpdate, Signature.toBase58(signature));
    return accountUpdate as AccountUpdate & { lazyAuthorization: undefined };
  }
  let { feePayer, accountUpdates, memo } = zkappCommand;
  return {
    feePayer: addFeePayerSignature(feePayer),
    accountUpdates: accountUpdates.map(addSignature),
    memo,
  };
}

function dummySignature() {
  return Signature.toBase58(Signature.dummy());
}

/**
 * The public input for zkApps consists of certain hashes of the proving
 AccountUpdate (and its child accountUpdates) which is constructed during method
 execution.

  For SmartContract proving, a method is run twice: First outside the proof, to
 obtain the public input, and once in the prover, which takes the public input
 as input. The current transaction is hashed again inside the prover, which
 asserts that the result equals the input public input, as part of the snark
 circuit. The block producer will also hash the transaction they receive and
 pass it as a public input to the verifier. Thus, the transaction is fully
 constrained by the proof - the proof couldn't be used to attest to a different
 transaction.
 */
type ZkappPublicInput = {
  accountUpdate: Field;
  calls: Field;
};
let ZkappPublicInput = provablePure({ accountUpdate: Field, calls: Field });

async function addMissingProofs(
  zkappCommand: ZkappCommand,
  { proofsEnabled = true }
): Promise<{
  zkappCommand: ZkappCommandProved;
  proofs: (Proof<ZkappPublicInput, Empty> | undefined)[];
}> {
  let { feePayer, accountUpdates, memo } = zkappCommand;
  // compute proofs serially. in parallel would clash with our global variable
  // hacks
  let accountUpdatesProved: AccountUpdateProved[] = [];
  let proofs: (Proof<ZkappPublicInput, Empty> | undefined)[] = [];
  for (let i = 0; i < accountUpdates.length; i++) {
    let { accountUpdateProved, proof } = await addProof(
      zkappCommand,
      i,
      proofsEnabled
    );
    accountUpdatesProved.push(accountUpdateProved);
    proofs.push(proof);
  }
  return {
    zkappCommand: { feePayer, accountUpdates: accountUpdatesProved, memo },
    proofs,
  };
}

async function addProof(
  transaction: ZkappCommand,
  index: number,
  proofsEnabled: boolean
) {
  let accountUpdate = transaction.accountUpdates[index];
  accountUpdate = AccountUpdate.clone(accountUpdate);

  if (accountUpdate.lazyAuthorization?.kind !== 'lazy-proof') {
    return {
      accountUpdateProved: accountUpdate as AccountUpdateProved,
      proof: undefined,
    };
  }
  if (!proofsEnabled) {
    Authorization.setProof(accountUpdate, await dummyBase64Proof());
    return {
      accountUpdateProved: accountUpdate as AccountUpdateProved,
      proof: undefined,
    };
  }

  let lazyProof: LazyProof = accountUpdate.lazyAuthorization;
  let prover = getZkappProver(lazyProof);
  let proverData = { transaction, accountUpdate, index };
  let proof = await createZkappProof(prover, lazyProof, proverData);

  let accountUpdateProved = Authorization.setProof(
    accountUpdate,
    Pickles.proofToBase64Transaction(proof.proof)
  );
  return { accountUpdateProved, proof };
```

```
}

async function createZkappProof(
  prover: Pickles.Prover,
  {
    methodName,
    args,
    previousProofs,
    ZkappClass,
    memoized,
    blindingValue,
  }: LazyProof,
  { transaction, accountUpdate, index }: ZkappProverData
): Promise<Proof<ZkappPublicInput, Empty>> {
  let publicInput = accountUpdate.toPublicInput();
  let publicInputFields = MlFieldConstArray.to(
    ZkappPublicInput.toFields(publicInput)
  );

  let [, , proof] = await zkAppProver.run(
    [accountUpdate.publicKey, accountUpdate.tokenId, ...args],
    { transaction, accountUpdate, index },
    async () => {
      let id = memoizationContext.enter({
        memoized,
        currentIndex: 0,
        blindingValue,
      });
      try {
        return await prover(publicInputFields, MlArray.to(previousProofs));
      } catch (err) {
        console.error(&nbsp;Error when proving ${ZkappClass.name}.${methodName}()&nbsp;);
        throw err;
      } finally {
        memoizationContext.leave(id);
      }
    }
  );

  let maxProofsVerified = ZkappClass._maxProofsVerified!;
  const Proof = ZkappClass.Proof();
  return new Proof({
    publicInput,
    publicOutput: undefined,
    proof,
    maxProofsVerified,
  });
}

function getZkappProver({ methodName, ZkappClass }: LazyProof) {
  if (ZkappClass._provers === undefined)
    throw Error(
    &nbsp;Cannot prove execution of ${methodName}(), no prover found. &nbsp; +
      &nbsp;Try calling \&nbsp;await ${ZkappClass.name}.compile()\&nbsp; first, this will cache provers in
the background.&nbsp;
  );
 let provers = ZkappClass._provers;
 let methodError =
   &nbsp;Error when computing proofs: Method ${methodName} not found. &nbsp; +
   &nbsp;Make sure your environment supports decorators, and annotate with \&nbsp;@method
${methodName}\&nbsp;.&nbsp;;
 if (ZkappClass._methods === undefined) throw Error(methodError);
 let i = ZkappClass._methods.findIndex((m) => m.methodName === methodName);
 if (i === -1) throw Error(methodError);
 return provers[i];
}
```

</file>

<file>

## path: /src/lib/account_update.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/account_update.unit-test.ts

```
import { mocks } from '../bindings/crypto/constants.js';
import {
 AccountUpdate,
 PrivateKey,
 Field,
 Mina,
 Int64,
 Types,
 Provable,
} from '../index.js';
import { Test } from '../snarky.js';
import { expect } from 'expect';

let address = PrivateKey.random().toPublicKey();

function createAccountUpdate() {
 let accountUpdate = AccountUpdate.defaultAccountUpdate(address);
 accountUpdate.body.balanceChange = Int64.from(1e9).neg();
 return accountUpdate;
}

// can convert account update to fields consistently
{
 let accountUpdate = createAccountUpdate();

 // convert accountUpdate to fields in OCaml, going via AccountUpdate.of_json
 let json = JSON.stringify(accountUpdate.toJSON().body);
 let [, ...fields1_] = Test.fieldsFromJson.accountUpdate(json);
```

```
  let fields1 = fields1_.map(Field);
  // convert accountUpdate to fields in pure JS, leveraging generated code
  let fields2 = Types.AccountUpdate.toFields(accountUpdate);

  // this is useful console output in the case the test should fail
  if (fields1.length !== fields2.length) {
    console.log(
      &nbsp;unequal length. expected ${fields1.length}, actual: ${fields2.length}&nbsp;
    );
  }
  for (let i = 0; i < fields1.length; i++) {
    if (fields1[i].toString() !== fields2[i].toString()) {
      console.log('unequal at', i);
      console.log(&nbsp;expected: ${fields1[i]} actual: ${fields2[i]}&nbsp;);
    }
  }

  expect(fields1.length).toEqual(fields2.length);
  expect(fields1.map(String)).toEqual(fields2.map(String));
  expect(fields1).toEqual(fields2);
}

// can hash an account update
{
  let accountUpdate = createAccountUpdate();

  // TODO remove restriction "This function can't be run outside of a checked computation."
  Provable.runAndCheck(() => {
    let hash = accountUpdate.hash();

    // if we clone the accountUpdate, hash should be the same
    let accountUpdate2 = AccountUpdate.clone(accountUpdate);
    expect(accountUpdate2.hash()).toEqual(hash);

    // if we change something on the cloned accountUpdate, the hash should become different
    AccountUpdate.setValue(accountUpdate2.update.appState[0], Field(1));
    expect(accountUpdate2.hash()).not.toEqual(hash);
  });
}

// converts account update to a public input that's consistent with the ocaml implementation
{
  let otherAddress = PrivateKey.random().toPublicKey();

  let accountUpdate = AccountUpdate.create(address);
  accountUpdate.approve(AccountUpdate.create(otherAddress));

  let publicInput = accountUpdate.toPublicInput();

  // create transaction JSON with the same accountUpdate structure, for ocaml version
  let tx = await Mina.transaction(() => {
    let accountUpdate = AccountUpdate.create(address);
    accountUpdate.approve(AccountUpdate.create(otherAddress));
  });
  let publicInputOcaml = Test.hashFromJson.zkappPublicInput(tx.toJSON(), 0);

  expect(publicInput).toEqual({
    accountUpdate: Field(publicInputOcaml.accountUpdate),
    calls: Field(publicInputOcaml.calls),
  });
}

// creates the right empty sequence state
{
  let accountUpdate = createAccountUpdate();
  expect(
    accountUpdate.body.preconditions.account.actionState.value.toString()
  ).toEqual(
    '25079927036070901246064867767436987657692091363973573142121686150614948079097'
  );
}

// creates the right empty vk hash
{
  let accountUpdate = createAccountUpdate();
  expect(
    accountUpdate.body.authorizationKind.verificationKeyHash.toString()
  ).toEqual(mocks.dummyVerificationKeyHash);
}

// does not throw an error if private key is missing unless if .send is executed
{
  let Local = Mina.LocalBlockchain({ proofsEnabled: false });
  Mina.setActiveInstance(Local);

  const feePayer = Local.testAccounts[0].publicKey;

  let tx = await Mina.transaction(feePayer, () => {
    AccountUpdate.fundNewAccount(feePayer);
  });
  tx.sign();
  await expect(tx.send()).rejects.toThrow(
    'Check signature: Invalid signature on fee payer for key'
  );
}
```

</file>

<file>

## path: /src/lib/base58-encodings.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/base58-encodings.ts

```
import { fieldEncodings } from './base58.js';
import { Field } from './core.js';

export { TokenId, ReceiptChainHash, LedgerHash, EpochSeed, StateHash };

const { TokenId, ReceiptChainHash, EpochSeed, LedgerHash, StateHash } =
  fieldEncodings(Field);

```

</file>

<file>

## path: /src/lib/base58.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/base58.ts

```
import { versionBytes } from '../bindings/crypto/constants.js';
import { Binable, withVersionNumber } from '../bindings/lib/binable.js';
import { sha256 } from 'js-sha256';
import { changeBase } from '../bindings/crypto/bigint-helpers.js';

export {
  toBase58Check,
  fromBase58Check,
  base58,
  withBase58,
  fieldEncodings,
  Base58,
  alphabet,
};

const alphabet =
  '123456789ABCDEFGHJKLMNPQRSTUVWXYZabcdefghijkmnopqrstuvwxyz'.split('');
let inverseAlphabet: Record<string, number> = {};
alphabet.forEach((c, i) => {
  inverseAlphabet[c] = i;
});

function toBase58Check(input: number[] | Uint8Array, versionByte: number) {
  let withVersion = [versionByte, ...input];
  let checksum = computeChecksum(withVersion);
  let withChecksum = withVersion.concat(checksum);
  return toBase58(withChecksum);
}

function fromBase58Check(base58: string, versionByte: number) {
  // throws on invalid character
  let bytes = fromBase58(base58);
  // check checksum
  let checksum = bytes.slice(-4);
  let originalBytes = bytes.slice(0, -4);
  let actualChecksum = computeChecksum(originalBytes);
  if (!arrayEqual(checksum, actualChecksum))
    throw Error('fromBase58Check: invalid checksum');
  // check version byte
  if (originalBytes[0] !== versionByte)
    throw Error(
      &nbsp;fromBase58Check: input version byte ${versionByte} does not match encoded version byte
${originalBytes[0]}&nbsp;
    );
  // return result
  return originalBytes.slice(1);
}

function toBase58(bytes: number[] | Uint8Array) {
  // count the leading zeroes. these get turned into leading zeroes in the output
  let z = 0;
  while (bytes[z] === 0) z++;
  // for some reason, this is big-endian, so we need to reverse
  let digits = [...bytes].map(BigInt).reverse();
  // change base and reverse
  let base58Digits = changeBase(digits, 256n, 58n).reverse();
  // add leading zeroes, map into alphabet
  base58Digits = Array(z).fill(0n).concat(base58Digits);
  return base58Digits.map((x) => alphabet[Number(x)]).join('');
}

function fromBase58(base58: string) {
  let base58Digits = [...base58].map((c) => {
    let digit = inverseAlphabet[c];
    if (digit === undefined) throw Error('fromBase58: invalid character');
    return BigInt(digit);
  });
  let z = 0;
  while (base58Digits[z] === 0n) z++;
  let digits = changeBase(base58Digits.reverse(), 58n, 256n).reverse();
  digits = Array(z).fill(0n).concat(digits);
  return digits.map(Number);
}

function computeChecksum(input: number[] | Uint8Array) {
  let hash1 = sha256.create();
  hash1.update(input);
  let hash2 = sha256.create();
  hash2.update(hash1.array());
  return hash2.array().slice(0, 4);
}

type Base58<T> = {
  toBase58(t: T): string;
  fromBase58(base58: string): T;
};
```

```
function base58<T>(binable: Binable<T>, versionByte: number): Base58<T> {
  return {
    toBase58(t) {
      let bytes = binable.toBytes(t);
      return toBase58Check(bytes, versionByte);
    },
    fromBase58(base58) {
      let bytes = fromBase58Check(base58, versionByte);
      return binable.fromBytes(bytes);
    },
  };
}

function withBase58<T>(
  binable: Binable<T>,
  versionByte: number
): Binable<T> & Base58<T> {
  return { ...binable, ...base58(binable, versionByte) };
}

// encoding of fields as base58, compatible with ocaml encodings (provided the versionByte and
versionNumber are the same)

function customEncoding<Field>(
  Field: Binable<Field>,
  versionByte: number,
  versionNumber?: number
) {
  let customField =
    versionNumber !== undefined
      ? withVersionNumber(Field, versionNumber)
      : Field;
  return base58(customField, versionByte);
}

const RECEIPT_CHAIN_HASH_VERSION = 1;
const LEDGER_HASH_VERSION = 1;
const EPOCH_SEED_VERSION = 1;
const STATE_HASH_VERSION = 1;

function fieldEncodings<Field>(Field: Binable<Field>) {
  const TokenId = customEncoding(Field, versionBytes.tokenIdKey);
  const ReceiptChainHash = customEncoding(
    Field,
    versionBytes.receiptChainHash,
    RECEIPT_CHAIN_HASH_VERSION
  );
  const LedgerHash = customEncoding(
    Field,
    versionBytes.ledgerHash,
    LEDGER_HASH_VERSION
  );
  const EpochSeed = customEncoding(
    Field,
    versionBytes.epochSeed,
    EPOCH_SEED_VERSION
  );
  const StateHash = customEncoding(
    Field,
    versionBytes.stateHash,
    STATE_HASH_VERSION
  );
  return { TokenId, ReceiptChainHash, LedgerHash, EpochSeed, StateHash };
}

function arrayEqual(a: unknown[], b: unknown[]) {
  if (a.length !== b.length) return false;
  for (let i = 0; i < a.length; i++) {
    if (a[i] !== b[i]) return false;
  }
  return true;
}
```

</file>

<file>

## path: /src/lib/base58.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/base58.unit-test.ts

```
import { fromBase58Check, toBase58Check } from './base58.js';
import { Test } from '../snarky.js';
import { expect } from 'expect';
import { test, Random, withHardCoded } from './testing/property.js';

let bytes = withHardCoded(
  Random.bytes(Random.nat(100)),
  [0, 0, 0, 0] // definitely test some zero bytes
);
let version = Random.nat(100);

test(bytes, version, (bytes, version, assert) => {
  let binaryString = String.fromCharCode(...bytes);
  let ocamlBytes = { t: 9, c: binaryString, l: bytes.length };
  let base58Ocaml = Test.encoding.toBase58(ocamlBytes, version);

  // check consistency with OCaml result
  let base58 = toBase58Check(bytes, version);
  assert(base58 === base58Ocaml, 'base58 agrees with ocaml');

  // check roundtrip
  let recoveredBytes = fromBase58Check(base58, version);
  expect(recoveredBytes).toEqual(bytes);
});

let goodExample = 'AhgX24Hr3v';
expect(toBase58Check([0, 1, 2], 1)).toEqual(goodExample);

// negative tests

// throws on invalid character
expect(() => fromBase58Check('@hgX24Hr3v', 1)).toThrow('invalid character');

// throws on invalid checksum
expect(() => fromBase58Check('AhgX24Hr3u', 1)).toThrow('invalid checksum');

// throws on invalid version byte
expect(() => fromBase58Check('AhgX24Hr3v', 2)).toThrow(
  '2 does not match encoded version byte 1'
);
```

</file>

<file>

## path: /src/lib/bool.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/bool.ts

```
import { Snarky } from '../snarky.js';
import {
  Field,
  FieldConst,
  FieldType,
  FieldVar,
  readVarMessage,
} from './field.js';
import { Bool as B } from '../provable/field-bigint.js';
import { defineBinable } from '../bindings/lib/binable.js';
import { NonNegativeInteger } from '../bindings/crypto/non-negative.js';
import { asProver } from './provable-context.js';

export { BoolVar, Bool, isBool };

// same representation, but use a different name to communicate intent / constraints
type BoolVar = FieldVar;

type ConstantBoolVar = [FieldType.Constant, FieldConst];
type ConstantBool = Bool & { value: ConstantBoolVar };

/**
 * A boolean value. You can use it like this:
 *
 * &nbsp;&nbsp;&nbsp;
 * const x = new Bool(true);
 * &nbsp;&nbsp;&nbsp;
 *
 * You can also combine multiple booleans via [[&nbsp;not&nbsp;]], [[&nbsp;and&nbsp;]],
[[&nbsp;or&nbsp;]].
 *
 * Use [[assertEquals]] to enforce the value of a Bool.
 */
class Bool {
  value: BoolVar;

  constructor(x: boolean | Bool | BoolVar) {
    if (Bool.#isBool(x)) {
      this.value = x.value;
      return;
    }
    if (Array.isArray(x)) {
      this.value = x;
      return;
    }
    this.value = FieldVar.constant(B(x));
  }

  isConstant(): this is { value: ConstantBoolVar } {
    return this.value[0] === FieldType.Constant;
  }

  /**
   * Converts a {@link Bool} to a {@link Field}. &nbsp;false&nbsp; becomes 0 and &nbsp;true&nbsp;
```

```
becomes 1.
  */
 toField(): Field {
   return Bool.toField(this);
 }

 /**
  * @returns a new {@link Bool} that is the negation of this {@link Bool}.
  */
 not(): Bool {
   if (this.isConstant()) {
     return new Bool(!this.toBoolean());
   }
   return new Bool(Snarky.bool.not(this.value));
 }

 /**
  * @param y A {@link Bool} to AND with this {@link Bool}.
  * @returns a new {@link Bool} that is set to true only if
  * this {@link Bool} and &nbsp;y&nbsp; are also true.
  */
 and(y: Bool | boolean): Bool {
   if (this.isConstant() && isConstant(y)) {
     return new Bool(this.toBoolean() && toBoolean(y));
   }
   return new Bool(Snarky.bool.and(this.value, Bool.#toVar(y)));
 }

 /**
  * @param y a {@link Bool} to OR with this {@link Bool}.
  * @returns a new {@link Bool} that is set to true if either
  * this {@link Bool} or &nbsp;y&nbsp; is true.
  */
 or(y: Bool | boolean): Bool {
   if (this.isConstant() && isConstant(y)) {
     return new Bool(this.toBoolean() || toBoolean(y));
   }
   return new Bool(Snarky.bool.or(this.value, Bool.#toVar(y)));
 }

 /**
  * Proves that this {@link Bool} is equal to &nbsp;y&nbsp;.
  * @param y a {@link Bool}.
  */
 assertEquals(y: Bool | boolean, message?: string): void {
   try {
     if (this.isConstant() && isConstant(y)) {
       if (this.toBoolean() !== toBoolean(y)) {
         throw Error(&nbsp;Bool.assertEquals(): ${this} != ${y}&nbsp;);
       }
       return;
     }
```

```
    Snarky.bool.assertEqual(this.value, Bool.#toVar(y));
  } catch (err) {
    throw withMessage(err, message);
  }
}

/**
 * Proves that this {@link Bool} is &nbsp;true&nbsp;.
 */
assertTrue(message?: string): void {
  try {
    if (this.isConstant() && !this.toBoolean()) {
      throw Error(&nbsp;Bool.assertTrue(): ${this} != ${true}&nbsp;);
    }
    this.assertEquals(true);
  } catch (err) {
    throw withMessage(err, message);
  }
}

/**
 * Proves that this {@link Bool} is &nbsp;false&nbsp;.
 */
assertFalse(message?: string): void {
  try {
    if (this.isConstant() && this.toBoolean()) {
      throw Error(&nbsp;Bool.assertFalse(): ${this} != ${false}&nbsp;);
    }
    this.assertEquals(false);
  } catch (err) {
    throw withMessage(err, message);
  }
}

/**
 * Returns true if this {@link Bool} is equal to &nbsp;y&nbsp;.
 * @param y a {@link Bool}.
 */
equals(y: Bool | boolean): Bool {
  if (this.isConstant() && isConstant(y)) {
    return new Bool(this.toBoolean() === toBoolean(y));
  }
  return new Bool(Snarky.bool.equals(this.value, Bool.#toVar(y)));
}

/**
 * Returns the size of this type.
 */
sizeInFields(): number {
  return 1;
}
```

```
/**
 * Serializes this {@link Bool} into {@link Field} elements.
 */
toFields(): Field[] {
  return Bool.toFields(this);
}

/**
 * Serialize the {@link Bool} to a string, e.g. for printing.
 * This operation does _not_ affect the circuit and can't be used to prove anything about the string
representation of the Field.
 */
toString(): string {
  return this.toBoolean().toString();
}

/**
 * Serialize the {@link Bool} to a JSON string.
 * This operation does _not_ affect the circuit and can't be used to prove anything about the string
representation of the Field.
 */
toJSON(): boolean {
  return this.toBoolean();
}

/**
 * This converts the {@link Bool} to a javascript [[boolean]].
 * This can only be called on non-witness values.
 */
toBoolean(): boolean {
  let value: FieldConst;
  if (this.isConstant()) {
    value = this.value[1];
  } else if (Snarky.run.inProverBlock()) {
    value = Snarky.field.readVar(this.value);
  } else {
    throw Error(readVarMessage('toBoolean', 'b', 'Bool'));
  }
  return FieldConst.equal(value, FieldConst[1]);
}

static toField(x: Bool | boolean): Field {
  return new Field(Bool.#toVar(x));
}

/**
 * Boolean negation.
 */
static not(x: Bool | boolean): Bool {
  if (Bool.#isBool(x)) {
    return x.not();
  }
  return new Bool(!x);
}

/**
 * Boolean AND operation.
 */
static and(x: Bool | boolean, y: Bool | boolean): Bool {
  if (Bool.#isBool(x)) {
    return x.and(y);
  }
  return new Bool(x).and(y);
}

/**
 * Boolean OR operation.
 */
static or(x: Bool | boolean, y: Bool | boolean): Bool {
  if (Bool.#isBool(x)) {
    return x.or(y);
  }
  return new Bool(x).or(y);
}

/**
 * Asserts if both {@link Bool} are equal.
 */
static assertEqual(x: Bool, y: Bool | boolean): void {
  if (Bool.#isBool(x)) {
    x.assertEquals(y);
    return;
  }
  new Bool(x).assertEquals(y);
}

/**
 * Checks two {@link Bool} for equality.
 */
static equal(x: Bool | boolean, y: Bool | boolean): Bool {
  if (Bool.#isBool(x)) {
    return x.equals(y);
  }
  return new Bool(x).equals(y);
}

/**
 * Static method to serialize a {@link Bool} into an array of {@link Field} elements.
 */
static toFields(x: Bool): Field[] {
  return [Bool.toField(x)];
}

/**
```

```
  * Static method to serialize a {@link Bool} into its auxiliary data.
  */
 static toAuxiliary(_?: Bool): [] {
   return [];
 }

 /**
  * Creates a data structure from an array of serialized {@link Field} elements.
  */
 static fromFields(fields: Field[]): Bool {
   if (fields.length !== 1) {
     throw Error(&nbsp;Bool.fromFields(): expected 1 field, got ${fields.length}&nbsp;);
   }
   return new Bool(fields[0].value);
 }

 /**
  * Serialize a {@link Bool} to a JSON string.
  * This operation does _not_ affect the circuit and can't be used to prove anything about the string
representation of the Field.
  */
 static toJSON(x: Bool): boolean {
   return x.toBoolean();
 }

 /**
  * Deserialize a JSON structure into a {@link Bool}.
  * This operation does _not_ affect the circuit and can't be used to prove anything about the string
representation of the Field.
  */
 static fromJSON(b: boolean): Bool {
   return new Bool(b);
 }

 /**
  * Returns the size of this type.
  */
 static sizeInFields() {
   return 1;
 }

 static toInput(x: Bool): { packed: [Field, number][] } {
   return { packed: [[x.toField(), 1] as [Field, number]] };
 }

 static toBytes(b: Bool): number[] {
   return BoolBinable.toBytes(b);
 }

 static fromBytes(bytes: number[]): Bool {
   return BoolBinable.fromBytes(bytes);
 }
```

```
  static readBytes<N extends number>(
    bytes: number[],
    offset: NonNegativeInteger<N>
  ): [value: Bool, offset: number] {
    return BoolBinable.readBytes(bytes, offset);
  }

  static sizeInBytes() {
    return 1;
  }

  static check(x: Bool): void {
    Snarky.field.assertBoolean(x.value);
  }

  static Unsafe = {
    /**
     * Converts a {@link Field} into a {@link Bool}. This is a **dangerous** operation
     * as it assumes that the field element is either 0 or 1 (which might not be true).
     *
     * Only use this with constants or if you have already constrained the Field element to be 0 or 1.
     *
     * @param x a {@link Field}
     */
    ofField(x: Field) {
      asProver(() => {
        let x0 = x.toBigInt();
        if (x0 !== 0n && x0 !== 1n)
          throw Error(&nbsp;Bool.Unsafe.ofField(): Expected 0 or 1, got ${x0}&nbsp;);
      });
      return new Bool(x.value);
    },
  };

  static #isBool(x: boolean | Bool | BoolVar): x is Bool {
    return x instanceof Bool;
  }

  static #toVar(x: boolean | Bool): BoolVar {
    if (Bool.#isBool(x)) return x.value;
    return FieldVar.constant(B(x));
  }
}

const BoolBinable = defineBinable({
  toBytes(b: Bool) {
    return [Number(b.toBoolean())];
  },
  readBytes(bytes, offset) {
    return [new Bool(!!bytes[offset]), offset + 1];
  },
});

function isConstant(x: boolean | Bool): x is boolean | ConstantBool {
  if (typeof x === 'boolean') {
    return true;
  }

  return x.isConstant();
}

function isBool(x: unknown) {
  return x instanceof Bool;
}

function toBoolean(x: boolean | Bool): boolean {
  if (typeof x === 'boolean') {
    return x;
  }
  return x.toBoolean();
}

// TODO: This is duplicated
function withMessage(error: unknown, message?: string) {
  if (message === undefined || !(error instanceof Error)) return error;
  error.message = &nbsp;${message}\n${error.message}&nbsp;;
  return error;
}
```

</file>

<file>

## path: /src/lib/caller.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/caller.unit-test.ts

```
import { AccountUpdate, TokenId } from './account_update.js';
import * as Mina from './mina.js';
import { expect } from 'expect';

let Local = Mina.LocalBlockchain();
Mina.setActiveInstance(Local);

let [{ privateKey, publicKey }] = Local.testAccounts;

let parentId = TokenId.derive(publicKey);

/**
 * tests whether the following two account updates gives the child token permissions:
 *
 * InheritFromParent -> ParentsOwnToken
 */
let tx = await Mina.transaction(privateKey, () => {
  let parent = AccountUpdate.defaultAccountUpdate(publicKey);
  parent.body.mayUseToken = AccountUpdate.MayUseToken.InheritFromParent;
  parent.balance.subInPlace(Mina.accountCreationFee());

  let child = AccountUpdate.defaultAccountUpdate(publicKey, parentId);
  child.body.mayUseToken = AccountUpdate.MayUseToken.ParentsOwnToken;

  AccountUpdate.attachToTransaction(parent);
  parent.approve(child);
});

// according to this test, the child doesn't get token permissions
await expect(tx.send()).rejects.toThrow(
  'can not use or pass on token permissions'
);
```

</file>

<file>

## path: /src/lib/circuit.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/circuit.ts

```
import { ProvablePure, Snarky } from '../snarky.js';
import { MlFieldArray, MlFieldConstArray } from './ml/fields.js';
import { withThreadPool } from '../bindings/js/wrapper.js';
import { Provable } from './provable.js';
import { snarkContext, gatesFromJson } from './provable-context.js';
import { prettifyStacktrace, prettifyStacktracePromise } from './errors.js';

// external API
export { public_, circuitMain, Circuit, Keypair, Proof, VerificationKey };
```

```
class Circuit {
  // circuit-writing interface

  static _main: CircuitData<any, any>;

  /**
   * Generates a proving key and a verification key for this circuit.
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const keypair = await MyCircuit.generateKeypair();
   * &nbsp;&nbsp;&nbsp;
   */
  static generateKeypair() {
    let main = mainFromCircuitData(this._main);
    let publicInputSize = this._main.publicInputType.sizeInFields();
    return prettifyStacktracePromise(
      withThreadPool(async () => {
        let keypair = Snarky.circuit.compile(main, publicInputSize);
        return new Keypair(keypair);
      })
    );
  }

  /**
   * Proves a statement using the private input, public input, and the {@link Keypair} of the circuit.
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const keypair = await MyCircuit.generateKeypair();
   * const proof = await MyCircuit.prove(privateInput, publicInput, keypair);
   * &nbsp;&nbsp;&nbsp;
   */
  static prove(privateInput: any[], publicInput: any[], keypair: Keypair) {
    let main = mainFromCircuitData(this._main, privateInput);
    let publicInputSize = this._main.publicInputType.sizeInFields();
    let publicInputFields = this._main.publicInputType.toFields(publicInput);
    return prettifyStacktracePromise(
      withThreadPool(async () => {
        let proof = Snarky.circuit.prove(
          main,
          publicInputSize,
          MlFieldConstArray.to(publicInputFields),
          keypair.value
        );
        return new Proof(proof);
      })
    );
  }

  /**
   * Verifies a proof using the public input, the proof, and the initial {@link Keypair} of the circuit.
   * @example
   * &nbsp;&nbsp;&nbsp;ts
```

```
 * const keypair = await MyCircuit.generateKeypair();
 * const proof = await MyCircuit.prove(privateInput, publicInput, keypair);
 * const isValid = await MyCircuit.verify(publicInput, keypair.vk, proof);
 * &nbsp;&nbsp;&nbsp;
 */
static verify(
  publicInput: any[],
  verificationKey: VerificationKey,
  proof: Proof
) {
  let publicInputFields = this._main.publicInputType.toFields(publicInput);
  return prettifyStacktracePromise(
    withThreadPool(async () =>
      Snarky.circuit.verify(
        MlFieldConstArray.to(publicInputFields),
        proof.value,
        verificationKey.value
      )
    )
  );
}

// utility namespace, moved to &nbsp;Provable&nbsp;

/**
 * @deprecated use {@link Provable.witness}
 */
static witness = Provable.witness;
/**
 * @deprecated use {@link Provable.asProver}
 */
static asProver = Provable.asProver;
/**
 * @deprecated use {@link Provable.runAndCheck}
 */
static runAndCheck = Provable.runAndCheck;
/**
 * @deprecated use {@link Provable.runUnchecked}
 */
static runUnchecked = Provable.runUnchecked;
/**
 * @deprecated use {@link Provable.constraintSystem}
 */
static constraintSystem = Provable.constraintSystem;
/**
 * @deprecated use {@link Provable.Array}
 */
static array = Provable.Array;
/**
 * @deprecated use {@link Provable.assertEqual}
 */
static assertEqual = Provable.assertEqual;
```

```
  /**
   * @deprecated use {@link Provable.equal}
   */
  static equal = Provable.equal;
  /**
   * @deprecated use {@link Provable.if}
   */
  static if = Provable.if;
  /**
   * @deprecated use {@link Provable.switch}
   */
  static switch = Provable.switch;
  /**
   * @deprecated use {@link Provable.inProver}
   */
  static inProver = Provable.inProver;
  /**
   * @deprecated use {@link Provable.inCheckedComputation}
   */
  static inCheckedComputation = Provable.inCheckedComputation;
  /**
   * @deprecated use {@link Provable.log}
   */
  static log = Provable.log;
}

class Keypair {
  value: Snarky.Keypair;

  constructor(value: Snarky.Keypair) {
    this.value = value;
  }

  verificationKey() {
    return new VerificationKey(
      Snarky.circuit.keypair.getVerificationKey(this.value)
    );
  }

  /**
   * Returns a low-level JSON representation of the {@link Circuit} from its {@link Keypair}:
   * a list of gates, each of which represents a row in a table, with certain coefficients and wires to other (row,
column) pairs
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const keypair = await MyCircuit.generateKeypair();
   * const json = MyProvable.witnessFromKeypair(keypair);
   * &nbsp;&nbsp;&nbsp;
   */
  constraintSystem() {
    try {
      return gatesFromJson(
        Snarky.circuit.keypair.getConstraintSystemJSON(this.value)
      ).gates;
    } catch (error) {
      throw prettifyStacktrace(error);
    }
  }
}

/**
 * Proofs can be verified using a {@link VerificationKey} and the public input.
 */
class Proof {
  value: Snarky.Proof;

  constructor(value: Snarky.Proof) {
    this.value = value;
  }
}

/**
 * Part of the circuit {@link Keypair}. A verification key can be used to verify a {@link Proof} when you
provide the correct public input.
 */
class VerificationKey {
  value: Snarky.VerificationKey;

  constructor(value: Snarky.VerificationKey) {
    this.value = value;
  }
}

function public_(target: any, _key: string | symbol, index: number) {
  // const fieldType = Reflect.getMetadata('design:paramtypes', target, key);

  if (target._public === undefined) {
    target._public = [];
  }
  target._public.push(index);
}

type CircuitData<P, W> = {
  main(publicInput: P, privateInput: W): void;
  publicInputType: ProvablePure<P>;
  privateInputType: ProvablePure<W>;
};

function mainFromCircuitData<P, W>(
  data: CircuitData<P, W>,
  privateInput?: W
): Snarky.Main {
  return function main(publicInputFields: MlFieldArray) {
    let id = snarkContext.enter({ inCheckedComputation: true });
    try {
      let publicInput = data.publicInputType.fromFields(
        MlFieldArray.from(publicInputFields)
      );
      let privateInput_ = Provable.witness(
        data.privateInputType,
        () => privateInput as W
      );
      data.main(publicInput, privateInput_);
    } finally {
      snarkContext.leave(id);
    }
  };
}

function circuitMain(
  target: typeof Circuit,
  propertyName: string,
  _descriptor?: PropertyDescriptor
): any {
  const paramTypes = Reflect.getMetadata(
    'design:paramtypes',
    target,
    propertyName
  );
  const numArgs = paramTypes.length;

  const publicIndexSet: Set<number> = new Set((target as any)._public);
  const witnessIndexSet: Set<number> = new Set();
  for (let i = 0; i < numArgs; ++i) {
    if (!publicIndexSet.has(i)) witnessIndexSet.add(i);
  }

  target._main = {
    main(publicInput: any[], privateInput: any[]) {
      let args = [];
      for (let i = 0; i < numArgs; ++i) {
        let nextInput = publicIndexSet.has(i) ? publicInput : privateInput;
        args.push(nextInput.shift());
      }
      return (target as any)[propertyName].apply(target, args);
    },
    publicInputType: provableFromTuple(
      Array.from(publicIndexSet).map((i) => paramTypes[i])
    ),
    privateInputType: provableFromTuple(
      Array.from(witnessIndexSet).map((i) => paramTypes[i])
    ),
  };
}

// TODO support auxiliary data
```

```
function provableFromTuple(typs: ProvablePure<any>[]): ProvablePure<any> {
  return {
    sizeInFields: () => {
      return typs.reduce((acc, typ) => acc + typ.sizeInFields(), 0);
    },

    toFields: (t: Array<any>) => {
      if (t.length !== typs.length) {
        throw new Error(&nbsp;typOfArray: Expected ${typs.length}, got ${t.length}&nbsp;);
      }
      let res = [];
      for (let i = 0; i < t.length; ++i) {
        res.push(...typs[i].toFields(t[i]));
      }
      return res;
    },

    toAuxiliary() {
      return [];
    },

    fromFields: (xs: Array<any>) => {
      let offset = 0;
      let res: Array<any> = [];
      typs.forEach((typ) => {
        const n = typ.sizeInFields();
        res.push(typ.fromFields(xs.slice(offset, offset + n)));
        offset += n;
      });
      return res;
    },

    check(xs: Array<any>) {
      typs.forEach((typ, i) => (typ as any).check(xs[i]));
    },
  };
}
```

</file>

<file>

## path: /src/lib/circuit_value.test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/circuit_value.test.ts

```
import {
  Bool,
  Provable,
  Int64,
  Struct,
  Field,
  PrivateKey,
  PublicKey,
} from 'o1js';

describe('circuit', () => {
  it('Provable.if out of snark', () => {
    let x = Provable.if(Bool(false), Int64, Int64.from(-1), Int64.from(-2));
    expect(x.toString()).toBe('-2');
  });

  it('Provable.if in snark', () => {
    Provable.runAndCheck(() => {
      let x = Provable.witness(Int64, () => Int64.from(-1));
      let y = Provable.witness(Int64, () => Int64.from(-2));
      let b = Provable.witness(Bool, () => Bool(true));

      let z = Provable.if(b, Int64, x, y);

      Provable.assertEqual(z, Int64.from(-1));
      Provable.asProver(() => {
        expect(z.toString()).toBe('-1');
      });

      z = Provable.if(b, Int64, Int64.from(99), y);

      Provable.assertEqual(z, Int64.from(99));
      Provable.asProver(() => {
        expect(z.toString()).toBe('99');
      });

      z = Provable.if(b.not(), Int64, Int64.from(99), y);

      Provable.assertEqual(z, Int64.from(-2));
      Provable.asProver(() => {
        expect(z.toString()).toBe('-2');
      });

      z = Provable.if(Bool(false), Int64, x, y);

      Provable.assertEqual(z, Int64.from(-2));
      Provable.asProver(() => {
        expect(z.toString()).toBe('-2');
      });
    });
  });

  it('Provable.switch picks the right value', () => {
    const x = Provable.switch([Bool(false), Bool(true), Bool(false)], Int64, [
      Int64.from(-1),
      Int64.from(-2),
      Int64.from(-3),
    ]);
    expect(x.toString()).toBe('-2');
  });

  it('Provable.switch returns 0 if the mask has only false elements', () => {
    const x = Provable.switch([Bool(false), Bool(false), Bool(false)], Int64, [
      Int64.from(-1),
      Int64.from(-2),
      Int64.from(-3),
    ]);
    expect(x.toString()).toBe('0');
  });

  it('Provable.switch throws when mask has >1 true elements', () => {
    expect(() =>
      Provable.switch([Bool(true), Bool(true), Bool(false)], Int64, [
        Int64.from(-1),
        Int64.from(-2),
        Int64.from(-3),
      ])
    ).toThrow(/&nbsp;mask&nbsp; must have 0 or 1 true element, found 2/);
  });

  it('Provable.assertEqual', () => {
    const FieldAndBool = Struct({ x: Field, b: Bool });

    Provable.runAndCheck(() => {
      let x = Provable.witness(Field, () => Field(1));
      let b = Provable.witness(Bool, () => Bool(true));

      // positive
      Provable.assertEqual(b, Bool(true));
      Provable.assertEqual(
        FieldAndBool,
        { x, b },
        { x: Field(1), b: Bool(true) }
      );

      //negative
      expect(() => Provable.assertEqual(b, Bool(false))).toThrow();
      expect(() =>
        Provable.assertEqual(
          FieldAndBool,
          { x, b },
          { x: Field(5), b: Bool(true) }
        )
      ).toThrow();
      expect(() => Provable.assertEqual(b, PrivateKey.random() as any)).toThrow(
        'must contain the same number of field elements'
      );
    });
  });
```

```
it('Provable.equal', () => {
  const FieldAndBool = Struct({ x: Field, b: Bool });
  let pk1 = PublicKey.fromBase58(
    'B62qoCHJ1dcGjKhdMTMuAytzRkLxRFUgq6YC5XSgmmxAt8r7FVi1DhT'
  );
  let pk2 = PublicKey.fromBase58(
    'B62qnDjh7J27q6CoG6hkQzP6J6t1USA6bCoKsBFhxNughNHQgVwEtT9'
  );

  function expectBoolean(b: Bool, expected: boolean) {
    Provable.asProver(() => {
      expect(b.toBoolean()).toEqual(expected);
    });
  }

  Provable.runAndCheck(() => {
    let x = Provable.witness(Field, () => Field(1));
    let b = Provable.witness(Bool, () => Bool(true));
    let pk = Provable.witness(PublicKey, () => pk1);

    expectBoolean(Provable.equal(pk, pk1), true);
    expectBoolean(
      Provable.equal(FieldAndBool, { x, b }, { x: Field(1), b: Bool(true) }),
      true
    );

    expectBoolean(Provable.equal(pk, pk2), false);
    expectBoolean(
      Provable.equal(FieldAndBool, { x, b }, { x: Field(1), b: Bool(false) }),
      false
    );

    expect(() => Provable.equal(b, pk2 as any)).toThrow(
      'must contain the same number of field elements'
    );
  });
});

it('can serialize Struct with array', async () => {
  class MyStruct extends Struct({
    values: Provable.Array(Field, 2),
  }) {}

  const original = new MyStruct({ values: [Field(0), Field(1)] });

  const serialized = MyStruct.toJSON(original);
  const reconstructed = MyStruct.fromJSON(serialized);

  Provable.assertEqual<MyStruct>(MyStruct, original, reconstructed);
});
```

```
  it('can serialize nested Struct', async () => {
    class OtherStruct extends Struct({
      x: Field,
    }) {
      toString() {
        return &nbsp;other-struct:${this.x}&nbsp;;
      }
    }

    class MyStruct extends Struct({
      a: Field,
      b: PrivateKey,
      y: OtherStruct,
    }) {
      toString() {
        return &nbsp;my-struct:${this.a};${this.b.toBase58()};${this.y}&nbsp;;
      }
    }

    const original = new MyStruct({
      a: Field(42),
      b: PrivateKey.random(),
      y: new OtherStruct({ x: Field(99) }),
    });

    const serialized = MyStruct.toJSON(original);
    const reconstructed = MyStruct.fromJSON(serialized);

    Provable.assertEqual(MyStruct, original, reconstructed);
    expect(reconstructed.toString()).toEqual(original.toString());
  });
});
```

</file>

<file>

## path: /src/lib/circuit_value.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/circuit_value.ts

```
import 'reflect-metadata';
import { ProvablePure } from '../snarky.js';
import { Field, Bool, Scalar, Group } from './core.js';
import {
  provable,
  provablePure,
  HashInput,
  NonMethods,
} from '../bindings/lib/provable-snarky.js';
import type {
  InferJson,
  InferProvable,
  InferredProvable,
  IsPure,
} from '../bindings/lib/provable-snarky.js';
import { Provable } from './provable.js';

// external API
export {
  CircuitValue,
  ProvableExtended,
  ProvablePureExtended,
  prop,
  arrayProp,
  matrixProp,
  provable,
  provablePure,
  Struct,
  FlexibleProvable,
  FlexibleProvablePure,
};

// internal API
export {
  AnyConstructor,
  cloneCircuitValue,
  circuitValueEquals,
  toConstant,
  isConstant,
  InferProvable,
  HashInput,
  InferJson,
  InferredProvable,
};

type ProvableExtension<T, TJson = any> = {
  toInput: (x: T) => { fields?: Field[]; packed?: [Field, number][] };
  toJSON: (x: T) => TJson;
  fromJSON: (x: TJson) => T;
};

type ProvableExtended<T, TJson = any> = Provable<T> &
  ProvableExtension<T, TJson>;
type ProvablePureExtended<T, TJson = any> = ProvablePure<T> &
  ProvableExtension<T, TJson>;

type Struct<T> = ProvableExtended<NonMethods<T>> &
  Constructor<T> & { _isStruct: true };
type StructPure<T> = ProvablePure<NonMethods<T>> &
  ProvableExtension<NonMethods<T>> &
  Constructor<T> & { _isStruct: true };
type FlexibleProvable<T> = Provable<T> | Struct<T>;
type FlexibleProvablePure<T> = ProvablePure<T> | StructPure<T>;

type Constructor<T> = new (...args: any) => T;
type AnyConstructor = Constructor<any>;

/**
 * @deprecated &nbsp;CircuitValue&nbsp; is deprecated in favor of {@link Struct}, which features a simpler
API and better typing.
 */
abstract class CircuitValue {
  constructor(...props: any[]) {
    // if this is called with no arguments, do nothing, to support simple super() calls
    if (props.length === 0) return;

    let fields = this.constructor.prototype._fields;
    if (fields === undefined) return;
    if (props.length !== fields.length) {
      throw Error(
        &nbsp;${this.constructor.name} constructor called with ${props.length} arguments, but expected
${fields.length}&nbsp;
      );
    }
    for (let i = 0; i < fields.length; ++i) {
      let [key] = fields[i];
      (this as any)[key] = props[i];
    }
  }

  static fromObject<T extends AnyConstructor>(
    this: T,
    value: NonMethods<InstanceType<T>>
  ): InstanceType<T> {
    return Object.assign(Object.create(this.prototype), value);
  }

  static sizeInFields(): number {
    const fields: [string, any][] = (this as any).prototype._fields;
    return fields.reduce((acc, [_, typ]) => acc + typ.sizeInFields(), 0);
  }

  static toFields<T extends AnyConstructor>(
    this: T,
    v: InstanceType<T>
  ): Field[] {
    const res: Field[] = [];
    const fields = this.prototype._fields;
    if (fields === undefined || fields === null) {
      return res;
    }
    for (let i = 0, n = fields.length; i < n; ++i) {
      const [key, propType] = fields[i];
      const subElts: Field[] = propType.toFields((v as any)[key]);
      subElts.forEach((x) => res.push(x));
    }
    return res;
  }

  static toAuxiliary(): [] {
    return [];
  }

  static toInput<T extends AnyConstructor>(
    this: T,
    v: InstanceType<T>
  ): HashInput {
    let input: HashInput = { fields: [], packed: [] };
    let fields = this.prototype._fields;
    if (fields === undefined) return input;
    for (let i = 0, n = fields.length; i < n; ++i) {
      let [key, type] = fields[i];
      if ('toInput' in type) {
        input = HashInput.append(input, type.toInput(v[key]));
        continue;
      }
      // as a fallback, use toFields on the type
      // TODO: this is problematic -- ignores if there's a toInput on a nested type
      // so, remove this? should every provable define toInput?
      let xs: Field[] = type.toFields(v[key]);
      input.fields!.push(...xs);
    }
    return input;
  }

  toFields(): Field[] {
    return (this.constructor as any).toFields(this);
  }

  toJSON(): any {
    return (this.constructor as any).toJSON(this);
  }

  toConstant(): this {
    return (this.constructor as any).toConstant(this);
  }

  equals(x: this) {
    return Provable.equal(this, x);
  }

  assertEquals(x: this) {
    Provable.assertEqual(this, x);
  }

  isConstant() {
    return this.toFields().every((x) => x.isConstant());
  }

  static fromFields<T extends AnyConstructor>(
    this: T,
    xs: Field[]
  ): InstanceType<T> {
    const fields: [string, any][] = (this as any).prototype._fields;
    if (xs.length < fields.length) {
      throw Error(
        &nbsp;${this.name}.fromFields: Expected ${fields.length} field elements, got ${xs?.length}&nbsp;
      );
    }
    let offset = 0;
    const props: any = {};
    for (let i = 0; i < fields.length; ++i) {
      const [key, propType] = fields[i];
      const propSize = propType.sizeInFields();
      const propVal = propType.fromFields(
        xs.slice(offset, offset + propSize),
        []
      );
      props[key] = propVal;
      offset += propSize;
    }
    return Object.assign(Object.create(this.prototype), props);
  }

  static check<T extends AnyConstructor>(this: T, v: InstanceType<T>) {
    const fields = (this as any).prototype._fields;
    if (fields === undefined || fields === null) {
      return;
    }
    for (let i = 0; i < fields.length; ++i) {
      const [key, propType] = fields[i];
      const value = (v as any)[key];
      if (propType.check === undefined)
        throw Error('bug: CircuitValue without .check()');
      propType.check(value);
    }
  }

  static toConstant<T extends AnyConstructor>(
    this: T,
    t: InstanceType<T>
  ): InstanceType<T> {
    const xs: Field[] = (this as any).toFields(t);
    return (this as any).fromFields(xs.map((x) => x.toConstant()));
  }

  static toJSON<T extends AnyConstructor>(this: T, v: InstanceType<T>) {
    const res: any = {};
```

```
    if ((this as any).prototype._fields !== undefined) {
      const fields: [string, any][] = (this as any).prototype._fields;
      fields.forEach(([key, propType]) => {
        res[key] = propType.toJSON((v as any)[key]);
      });
    }
    return res;
  }

  static fromJSON<T extends AnyConstructor>(
    this: T,
    value: any
  ): InstanceType<T> {
    let props: any = {};
    let fields: [string, any][] = (this as any).prototype._fields;
    if (typeof value !== 'object' || value === null || Array.isArray(value)) {
      throw Error(&nbsp;${this.name}.fromJSON(): invalid input ${value}&nbsp;);
    }
    if (fields !== undefined) {
      for (let i = 0; i < fields.length; ++i) {
        let [key, propType] = fields[i];
        if (value[key] === undefined) {
          throw Error(&nbsp;${this.name}.fromJSON(): invalid input ${value}&nbsp;);
        } else {
          props[key] = propType.fromJSON(value[key]);
        }
      }
    }
    return Object.assign(Object.create(this.prototype), props);
  }
}

function prop(this: any, target: any, key: string) {
  const fieldType = Reflect.getMetadata('design:type', target, key);
  if (!target.hasOwnProperty('_fields')) {
    target._fields = [];
  }
  if (fieldType === undefined) {
  } else if (fieldType.toFields && fieldType.fromFields) {
    target._fields.push([key, fieldType]);
  } else {
    console.log(
      &nbsp;warning: property ${key} missing field element conversion methods&nbsp;
    );
  }
}

function arrayProp<T>(elementType: FlexibleProvable<T>, length: number) {
  return function (target: any, key: string) {
    if (!target.hasOwnProperty('_fields')) {
      target._fields = [];
    }
    target._fields.push([key, Provable.Array(elementType, length)]);
  };
}

function matrixProp<T>(
  elementType: FlexibleProvable<T>,
  nRows: number,
  nColumns: number
) {
  return function (target: any, key: string) {
    if (!target.hasOwnProperty('_fields')) {
      target._fields = [];
    }
    target._fields.push([
      key,
      Provable.Array(Provable.Array(elementType, nColumns), nRows),
    ]);
  };
}

/**
 * &nbsp;Struct&nbsp; lets you declare composite types for use in o1js circuits.
 *
 * These composite types can be passed in as arguments to smart contract methods, used for on-chain state
variables
 * or as event / action types.
 *
 * Here's an example of creating a "Voter" struct, which holds a public key and a collection of votes on 3
different proposals:
 * &nbsp;&nbsp;&nbsp;ts
 * let Vote = { hasVoted: Bool, inFavor: Bool };
 *
 * class Voter extends Struct({
 *   publicKey: PublicKey,
 *   votes: [Vote, Vote, Vote]
 * }) {}
 *
 * // use Voter as SmartContract input:
 * class VoterContract extends SmartContract {
 *   \@method register(voter: Voter) {
 *     // ...
 *   }
 * }
 * &nbsp;&nbsp;&nbsp;
 * In this example, there are no instance methods on the class. This makes &nbsp;Voter&nbsp; type-
compatible with an anonymous object of the form
 * &nbsp;{ publicKey: PublicKey, votes: Vote[] }&nbsp;.
 * This mean you don't have to create instances by using &nbsp;new Voter(...)&nbsp;, you can operate with
plain objects:
 * &nbsp;&nbsp;&nbsp;ts
 * voterContract.register({ publicKey, votes });
 * &nbsp;&nbsp;&nbsp;
 *
 * On the other hand, you can also add your own methods:
 * &nbsp;&nbsp;&nbsp;ts
 * class Voter extends Struct({
 *   publicKey: PublicKey,
 *   votes: [Vote, Vote, Vote]
 * }) {
 *   vote(index: number, inFavor: Bool) {
 *     let vote = this.votes[i];
 *     vote.hasVoted = Bool(true);
 *     vote.inFavor = inFavor;
 *   }
 * }
 * &nbsp;&nbsp;&nbsp;
 *
 * In this case, you'll need the constructor to create instances of &nbsp;Voter&nbsp;. It always takes as input
the plain object:
 * &nbsp;&nbsp;&nbsp;ts
 * let emptyVote = { hasVoted: Bool(false), inFavor: Bool(false) };
 * let voter = new Voter({ publicKey, votes: Array(3).fill(emptyVote) });
 * voter.vote(1, Bool(true));
 * &nbsp;&nbsp;&nbsp;
 *
 * In addition to creating types composed of Field elements, you can also include auxiliary data which does
not become part of the proof.
 * This, for example, allows you to re-use the same type outside o1js methods, where you might want to
store additional metadata.
 *
 * To declare non-proof values of type &nbsp;string&nbsp;, &nbsp;number&nbsp;, etc, you can use the built-
in objects &nbsp;String&nbsp;, &nbsp;Number&nbsp;, etc.
 * Here's how we could add the voter's name (a string) as auxiliary data:
 * &nbsp;&nbsp;&nbsp;ts
 * class Voter extends Struct({
 *   publicKey: PublicKey,
 *   votes: [Vote, Vote, Vote],
 *   fullName: String
 * }) {}
 * &nbsp;&nbsp;&nbsp;
 *
 * Again, it's important to note that this doesn't enable you to prove anything about the
&nbsp;fullName&nbsp; string.
 * From the circuit point of view, it simply doesn't exist!
 *
 * @param type Object specifying the layout of the &nbsp;Struct&nbsp;
 * @param options Advanced option which allows you to force a certain order of object keys
 * @returns Class which you can extend
 */
function Struct<
  A,
  T extends InferProvable<A> = InferProvable<A>,
  J extends InferJson<A> = InferJson<A>,
  Pure extends boolean = IsPure<A>
```

```
>(
  type: A
): (new (value: T) => T) & { _isStruct: true } & (Pure extends true
    ? ProvablePure<T>
    : Provable<T>) & {
    toInput: (x: T) => {
      fields?: Field[] | undefined;
      packed?: [Field, number][] | undefined;
    };
    toJSON: (x: T) => J;
    fromJSON: (x: J) => T;
  } {
  class Struct_ {
    static type = provable<A>(type);
    static _isStruct: true;

    constructor(value: T) {
      Object.assign(this, value);
    }
    /**
     * This method is for internal use, you will probably not need it.
     * @returns the size of this struct in field elements
     */
    static sizeInFields() {
      return this.type.sizeInFields();
    }
    /**
     * This method is for internal use, you will probably not need it.
     * @param value
     * @returns the raw list of field elements that represent this struct inside the proof
     */
    static toFields(value: T): Field[] {
      return this.type.toFields(value);
    }
    /**
     * This method is for internal use, you will probably not need it.
     * @param value
     * @returns the raw non-field element data contained in the struct
     */
    static toAuxiliary(value: T): any[] {
      return this.type.toAuxiliary(value);
    }
    /**
     * This method is for internal use, you will probably not need it.
     * @param value
     * @returns a representation of this struct as field elements, which can be hashed efficiently
     */
    static toInput(value: T): HashInput {
      return this.type.toInput(value);
    }
    /**
     * Convert this struct to a JSON object, consisting only of numbers, strings, booleans, arrays and plain
objects.
     * @param value
     * @returns a JSON representation of this struct
     */
    static toJSON(value: T): J {
      return this.type.toJSON(value) as J;
    }
    /**
     * Convert from a JSON object to an instance of this struct.
     * @param json
     * @returns a JSON representation of this struct
     */
    static fromJSON(json: J): T {
      let value = this.type.fromJSON(json);
      let struct = Object.create(this.prototype);
      return Object.assign(struct, value);
    }
    /**
     * This method is for internal use, you will probably not need it.
     * Method to make assertions which should be always made whenever a struct of this type is created in a
proof.
     * @param value
     */
    static check(value: T) {
      return this.type.check(value);
    }
    /**
     * This method is for internal use, you will probably not need it.
     * Recover a struct from its raw field elements and auxiliary data.
     * @param fields the raw fields elements
     * @param aux the raw non-field element data
     */
    static fromFields(fields: Field[], aux: any[]) {
      let value = this.type.fromFields(fields, aux) as T;
      let struct = Object.create(this.prototype);
      return Object.assign(struct, value);
    }
  }
  return Struct_ as any;
}

let primitives = new Set([Field, Bool, Scalar, Group]);
function isPrimitive(obj: any) {
  for (let P of primitives) {
    if (obj instanceof P) return true;
  }
  return false;
}

function cloneCircuitValue<T>(obj: T): T {
  // primitive JS types and functions aren't cloned
  if (typeof obj !== 'object' || obj === null) return obj;
  // HACK: callbacks, account udpates
  if (
    obj.constructor?.name.includes('GenericArgument') ||
    obj.constructor?.name.includes('Callback')
  ) {
    return obj;
  }
  if (obj.constructor?.name.includes('AccountUpdate')) {
    return (obj as any).constructor.clone(obj);
  }

  // built-in JS datatypes with custom cloning strategies
  if (Array.isArray(obj)) return obj.map(cloneCircuitValue) as any as T;
  if (obj instanceof Set)
    return new Set([...obj].map(cloneCircuitValue)) as any as T;
  if (obj instanceof Map)
    return new Map(
      [...obj].map(([k, v]) => [k, cloneCircuitValue(v)])
    ) as any as T;
  if (ArrayBuffer.isView(obj)) return new (obj.constructor as any)(obj);

  // o1js primitives aren't cloned
  if (isPrimitive(obj)) {
    return obj;
  }

  // cloning strategy that works for plain objects AND classes whose constructor only assigns properties
  let propertyDescriptors: Record<string, PropertyDescriptor> = {};
  for (let [key, value] of Object.entries(obj)) {
    propertyDescriptors[key] = {
      value: cloneCircuitValue(value),
      writable: true,
      enumerable: true,
      configurable: true,
    };
  }
  return Object.create(Object.getPrototypeOf(obj), propertyDescriptors);
}

function circuitValueEquals<T>(a: T, b: T): boolean {
  // primitive JS types and functions are checked for exact equality
  if (
    typeof a !== 'object' ||
    a === null ||
    typeof b !== 'object' ||
    b === null
  )
    return a === b;

  // built-in JS datatypes with custom equality checks
  if (Array.isArray(a)) {
    return (
      Array.isArray(b) &&
      a.length === b.length &&
      a.every((a_, i) => circuitValueEquals(a_, b[i]))
    );
  }
  if (a instanceof Set) {
    return (
      b instanceof Set && a.size === b.size && [...a].every((a_) => b.has(a_))
    );
  }
  if (a instanceof Map) {
    return (
      b instanceof Map &&
      a.size === b.size &&
      [...a].every(([k, v]) => circuitValueEquals(v, b.get(k)))
    );
  }
  if (ArrayBuffer.isView(a) && !(a instanceof DataView)) {
    // typed array
    return (
      ArrayBuffer.isView(b) &&
      !(b instanceof DataView) &&
      circuitValueEquals([...(a as any)], [...(b as any)])
    );
  }

  // the two checks below cover o1js primitives and CircuitValues
  // if we have an .equals method, try to use it
  if ('equals' in a && typeof (a as any).equals === 'function') {
    let isEqual = (a as any).equals(b).toBoolean();
    if (typeof isEqual === 'boolean') return isEqual;
    if (isEqual instanceof Bool) return isEqual.toBoolean();
  }
  // if we have a .toFields method, try to use it
  if (
    'toFields' in a &&
    typeof (a as any).toFields === 'function' &&
    'toFields' in b &&
    typeof (b as any).toFields === 'function'
  ) {
    let aFields = (a as any).toFields() as Field[];
    let bFields = (b as any).toFields() as Field[];
    return aFields.every((a, i) => a.equals(bFields[i]).toBoolean());
  }

  // equality test that works for plain objects AND classes whose constructor only assigns properties
  let aEntries = Object.entries(a as any).filter(([, v]) => v !== undefined);
  let bEntries = Object.entries(b as any).filter(([, v]) => v !== undefined);
  if (aEntries.length !== bEntries.length) return false;
  return aEntries.every(
    ([key, value]) => key in b && circuitValueEquals((b as any)[key], value)
  );
}

function toConstant<T>(type: FlexibleProvable<T>, value: T): T;
function toConstant<T>(type: Provable<T>, value: T): T {
  return type.fromFields(
    type.toFields(value).map((x) => x.toConstant()),
    type.toAuxiliary(value)
  );
}

function isConstant<T>(type: FlexibleProvable<T>, value: T): boolean;
function isConstant<T>(type: Provable<T>, value: T): boolean {
  return type.toFields(value).every((x) => x.isConstant());
}
```

</file>

<file>

## path: /src/lib/circuit_value.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/circuit_value.unit-test.ts

```
import { provable, Struct } from './circuit_value.js';
import { UInt32 } from './int.js';
import { PrivateKey, PublicKey } from './signature.js';
import { expect } from 'expect';
import { method, SmartContract } from './zkapp.js';
import { LocalBlockchain, setActiveInstance, transaction } from './mina.js';
import { State, state } from './state.js';
import { AccountUpdate } from './account_update.js';
import { Provable } from './provable.js';
import { Field } from './core.js';

let type = provable({
  nested: { a: Number, b: Boolean },
  other: String,
  pk: PublicKey,
  uint: [UInt32, UInt32],
});

let value = {
  nested: { a: 1, b: true },
  other: 'arbitrary data!!!',
  pk: PublicKey.empty(),
  uint: [UInt32.one, UInt32.from(2)],
};
let original = JSON.stringify(value);

// sizeInFields
```

```
expect(type.sizeInFields()).toEqual(4);

// toFields
// note that alphabetical order of keys determines ordering here and elsewhere
let fields = type.toFields(value);
expect(fields).toEqual([Field(0), Field(0), Field(1), Field(2)]);

// toAuxiliary
let aux = type.toAuxiliary(value);
expect(aux).toEqual([[[1], [true]], ['arbitrary data!!!'], [], [[], []]]);

// toInput
let input = type.toInput(value);
expect(input).toEqual({
  fields: [Field(0)],
  packed: [
    [Field(0), 1],
    [Field(1), 32],
    [Field(2), 32],
  ],
});

// toJSON
expect(type.toJSON(value)).toEqual({
  nested: { a: 1, b: true },
  other: 'arbitrary data!!!',
  pk: PublicKey.toBase58(PublicKey.empty()),
  uint: ['1', '2'],
});

// fromFields
let restored = type.fromFields(fields, aux);
expect(JSON.stringify(restored)).toEqual(original);

// check
Provable.runAndCheck(() => {
  type.check(value);
});

// should fail &nbsp;check&nbsp; if &nbsp;check&nbsp; of subfields doesn't pass
expect(() =>
  Provable.runAndCheck(() => {
    let x = Provable.witness(type, () => ({
      ...value,
      uint: [
        UInt32.zero,
        // invalid Uint32
        new UInt32(Field(-1)),
      ],
    }));
  })
).toThrow(&nbsp;Constraint unsatisfied&nbsp;);
```

```
// class version of &nbsp;provable&nbsp;
class MyStruct extends Struct({
  nested: { a: Number, b: Boolean },
  other: String,
  pk: PublicKey,
  uint: [UInt32, UInt32],
}) {}

class MyStructPure extends Struct({
  nested: { a: Field, b: UInt32 },
  other: Field,
  pk: PublicKey,
  uint: [UInt32, UInt32],
}) {}

class MyTuple extends Struct([PublicKey, String]) {}

let targetString = 'some particular string';
let gotTargetString = false;

// create a smart contract and pass auxiliary data to a method
class MyContract extends SmartContract {
  // this is correctly rejected by the compiler -- on-chain state can't have stuff like strings in it
  // @state(MyStruct) y = State<MyStruct>();

  // this works because MyStructPure only contains field elements
  @state(MyStructPure) x = State<MyStructPure>();

  @method myMethod(value: MyStruct, tuple: MyTuple, update: AccountUpdate) {
    // check if we can pass in string values
    if (value.other === targetString) gotTargetString = true;
    value.uint[0].assertEquals(UInt32.zero);

    Provable.asProver(() => {
      let err = 'wrong value in prover';
      if (tuple[1] !== targetString) throw Error(err);

      // check if we can pass in account updates
      if (update.lazyAuthorization?.kind !== 'lazy-signature') throw Error(err);
      if (update.lazyAuthorization.privateKey?.toBase58() !== key.toBase58())
        throw Error(err);
    });
  }
}

setActiveInstance(LocalBlockchain());

await MyContract.compile();
let key = PrivateKey.random();
let address = key.toPublicKey();
let contract = new MyContract(address);
let tx = await transaction(() => {
  let accountUpdate = AccountUpdate.createSigned(key);

  contract.myMethod(
    {
      nested: { a: 1, b: false },
      other: targetString,
      pk: PublicKey.empty(),
      uint: [UInt32.from(0), UInt32.from(10)],
    },
    [address, targetString],
    accountUpdate
  );
});

gotTargetString = false;

await tx.prove();

// assert that prover got the target string
expect(gotTargetString).toEqual(true);

console.log('provable types work as expected!      ');
```

</file>

<file>

## path: /src/lib/core.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/core.ts

```
import { Field as InternalField } from './field.js';
import { Bool as InternalBool } from './bool.js';
import { Group as InternalGroup } from './group.js';
import { Scalar } from './scalar.js';

export { Field, Bool, Scalar, Group };

/**
 * A {@link Field} is an element of a prime order [finite field](https://en.wikipedia.org/wiki/Finite_field).
 * Every other provable type is built using the {@link Field} type.
 *
 * The field is the [pasta base field](https://electriccoin.co/blog/the-pasta-curves-for-halo-2-and-beyond/) of
order 2^254 + 0x224698fc094cf91b992d30ed00000001 ({@link Field.ORDER}).
 *
 * You can create a new Field from everything "field-like" (&nbsp;bigint&nbsp;, integer &nbsp;number&nbsp;,
decimal &nbsp;string&nbsp;, &nbsp;Field&nbsp;).
 * @example
 * &nbsp;&nbsp;&nbsp;
 * Field(10n); // Field construction from a bigint
 * Field(100); // Field construction from a number
 * Field("1"); // Field construction from a decimal string
 * &nbsp;&nbsp;&nbsp;
 *
 * **Beware**: Fields _cannot_ be constructed from fractional numbers or alphanumeric strings:
 * &nbsp;&nbsp;&nbsp;ts
 * Field(3.141); // ERROR: Cannot convert a float to a field element
 * Field("abc"); // ERROR: Invalid argument "abc"
 * &nbsp;&nbsp;&nbsp;
 *
 * Creating a Field from a negative number can result in unexpected behavior if you are not familiar with
[modular arithmetic](https://en.wikipedia.org/wiki/Modular_arithmetic).
 * @example
 * &nbsp;&nbsp;&nbsp;
 * const x = Field(-1); // valid Field construction from negative number
 * const y = Field(Field.ORDER - 1n); // same as &nbsp;x&nbsp;
 * &nbsp;&nbsp;&nbsp;
 *
 * **Important**: All the functions defined on a Field (arithmetic, logic, etc.) take their arguments as "field-
like".
 * A Field itself is also defined as a "field-like" element.
 *
 * @param value - the value to convert to a {@link Field}
 *
 * @return A {@link Field} with the value converted from the argument
 */
const Field = toFunctionConstructor(InternalField);
type Field = InternalField;

/**
 * A boolean value. You can create it like this:
 *
 * @example
 * &nbsp;&nbsp;&nbsp;
 * const b = Bool(true);
 * &nbsp;&nbsp;&nbsp;
 *
 * You can also combine multiple Bools with boolean operations:
 *
 * @example
 * &nbsp;&nbsp;&nbsp;ts
 * const c = Bool(false);
 *
 * const d = b.or(c).and(false).not();
 *
 * d.assertTrue();
 * &nbsp;&nbsp;&nbsp;
 *
 * Bools are often created by methods on other types such as &nbsp;Field.equals()&nbsp;:
 *
 * &nbsp;&nbsp;&nbsp;ts
 * const b: Bool = Field(5).equals(6);
 * &nbsp;&nbsp;&nbsp;
 */
const Bool = toFunctionConstructor(InternalBool);
type Bool = InternalBool;

/**
 * An element of a Group.
 */
const Group = toFunctionConstructor(InternalGroup);
type Group = InternalGroup;

function toFunctionConstructor<Class extends new (...args: any) => any>(
  Class: Class
): Class & ((...args: InferArgs<Class>) => InferReturn<Class>) {
  function Constructor(...args: any) {
    return new Class(...args);
  }
  Object.defineProperties(Constructor, Object.getOwnPropertyDescriptors(Class));
  return Constructor as any;
}

type InferArgs<T> = T extends new (...args: infer Args) => any ? Args : never;
type InferReturn<T> = T extends new (...args: any) => infer Return
  ? Return
  : never;
```

</file>

<file>

## path: /src/lib/encryption.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/encryption.ts

```
import { Field, Scalar, Group } from './core.js';
import { Poseidon } from './hash.js';
import { Provable } from './provable.js';
import { PrivateKey, PublicKey } from './signature.js';

export { encrypt, decrypt };

type CipherText = {
  publicKey: Group;
  cipherText: Field[];
};

/**
 * Public Key Encryption, using a given array of {@link Field} elements and encrypts it using a {@link
PublicKey}.
 */
```

```
function encrypt(message: Field[], otherPublicKey: PublicKey) {
  // key exchange
  let privateKey = Provable.witness(Scalar, () => Scalar.random());
  let publicKey = Group.generator.scale(privateKey);
  let sharedSecret = otherPublicKey.toGroup().scale(privateKey);

  let sponge = new Poseidon.Sponge();
  sponge.absorb(sharedSecret.x); // don't think we need y, that's enough entropy

  // encryption
  let cipherText = [];
  for (let i = 0; i < message.length; i++) {
    let keyStream = sponge.squeeze();
    let encryptedChunk = message[i].add(keyStream);
    cipherText.push(encryptedChunk);
    // absorb for the auth tag (two at a time for saving permutations)
    if (i % 2 === 1) sponge.absorb(cipherText[i - 1]);
    if (i % 2 === 1 || i === message.length - 1) sponge.absorb(cipherText[i]);
  }
  // authentication tag
  let authenticationTag = sponge.squeeze();
  cipherText.push(authenticationTag);

  return { publicKey, cipherText };
}

/**
 * Decrypts a {@link CipherText} using a {@link PrivateKey}.^
 */
function decrypt(
  { publicKey, cipherText }: CipherText,
  privateKey: PrivateKey
) {
  // key exchange
  let sharedSecret = publicKey.scale(privateKey.s);

  let sponge = new Poseidon.Sponge();
  sponge.absorb(sharedSecret.x);
  let authenticationTag = cipherText.pop();

  // decryption
  let message = [];
  for (let i = 0; i < cipherText.length; i++) {
    let keyStream = sponge.squeeze();
    let messageChunk = cipherText[i].sub(keyStream);
    message.push(messageChunk);
    if (i % 2 === 1) sponge.absorb(cipherText[i - 1]);
    if (i % 2 === 1 || i === cipherText.length - 1)
      sponge.absorb(cipherText[i]);
  }
  // authentication tag
  sponge.squeeze().assertEquals(authenticationTag!);
  return message;
}
```

</file>

<file>

## path: /src/lib/errors.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/errors.ts

```
export {
  CatchAndPrettifyStacktraceForAllMethods,
  CatchAndPrettifyStacktrace,
  prettifyStacktrace,
  prettifyStacktracePromise,
  assert,
};

/**
 * A class decorator that applies the CatchAndPrettifyStacktrace decorator function
 * to all methods of the target class.
 *
 * @param constructor - The target class constructor.
 */
function CatchAndPrettifyStacktraceForAllMethods<
  T extends { new (...args: any[]): {} }
>(constructor: T) {
  // Iterate through all properties (including methods) of the class prototype
  for (const propertyName of Object.getOwnPropertyNames(
    constructor.prototype
  )) {
    // Skip the constructor
    if (propertyName === 'constructor') continue;

    // Get the property descriptor
    const descriptor = Object.getOwnPropertyDescriptor(
      constructor.prototype,
      propertyName
    );

    // Check if the property is a method
    if (descriptor && typeof descriptor.value === 'function') {
      // Apply the CatchAndPrettifyStacktrace decorator to the method
      CatchAndPrettifyStacktrace(
        constructor.prototype,
        propertyName,
        descriptor
      );
    // Update the method descriptor
    Object.defineProperty(constructor.prototype, propertyName, descriptor);
   }
  }
  // do the same thing for static methods
  for (let [propertyName, descriptor] of Object.entries(
   Object.getOwnPropertyDescriptors(constructor)
  )) {
   if (descriptor && typeof descriptor.value === 'function') {
    CatchAndPrettifyStacktrace(constructor, propertyName, descriptor);
    Object.defineProperty(constructor, propertyName, descriptor);
   }
  }
}

/**
 * A decorator function that wraps the target method with error handling logic.
 * It catches errors thrown by the method, prettifies the stack trace, and then
 * rethrows the error with the updated stack trace.
 *
 * @param _target - The target object.
 * @param _propertyName - The name of the property being decorated.
 * @param descriptor - The property descriptor of the target method.
 */
function CatchAndPrettifyStacktrace(
  _target: any,
  _propertyName: string,
  descriptor: PropertyDescriptor
) {
  const originalMethod = descriptor.value;
  descriptor.value = function (...args: any[]) {
   try {
    const result = originalMethod.apply(this, args);
    return handleResult(result);
   } catch (error) {
    throw prettifyStacktrace(error);
   }
  };
}

/**
 * Handles the result of a function call, wrapping a Promise with error handling logic
 * that prettifies the stack trace before rethrowing the error. For non-Promise results,
 * the function returns the result unchanged. This function is intended for internal usage
 * and not exposed to users.
 *
 * @param result - The result of the function call, which can be a Promise or any other value.
 * @returns A Promise with error handling logic for prettifying the stack trace, or the original result if it's not a
Promise.
 */
function handleResult(result: any) {
  if (result instanceof Promise) {
    return result.catch((error: Error) => {
      throw prettifyStacktrace(error);
    });
  }
  return result;
}

/**
 * A list of keywords used to filter out unwanted lines from the error stack trace.
 */
const lineRemovalKeywords = [
  'snarky_js_node.bc.cjs',
  '/builtin/',
  'CatchAndPrettifyStacktrace', // Decorator name to remove from stacktrace (covers both class and method
decorator)
] as const;

/**
 * Prettifies the stack trace of an error by removing unwanted lines and trimming paths.
 *
 * @param error - The error object with a stack trace to prettify.
 * @returns The same error with the prettified stack trace
 */
function prettifyStacktrace(error: unknown) {
  error = unwrapMlException(error);
  if (!(error instanceof Error) || !error.stack) return error;

  const stacktrace = error.stack;
  const stacktraceLines = stacktrace.split('\n');
  const newStacktrace: string[] = [];

  for (let i = 0; i < stacktraceLines.length; i++) {
    const shouldRemoveLine = lineRemovalKeywords.some((lineToRemove) =>
      stacktraceLines[i].includes(lineToRemove)
    );
    if (shouldRemoveLine) {
      continue;
    }
    const trimmedLine = trimPaths(stacktraceLines[i]);
    newStacktrace.push(trimmedLine);
  }
  error.stack = newStacktrace.join('\n');
  return error;
}

async function prettifyStacktracePromise<T>(result: Promise<T>): Promise<T> {
  try {
    return await result;
  } catch (error) {
    throw prettifyStacktrace(error);
  }
}
```

```
function unwrapMlException<E extends unknown>(error: E) {
  if (error instanceof Error) return error;
  // ocaml exception re-thrown from JS
  if (Array.isArray(error) && error[2] instanceof Error) return error[2];
  return error;
}

/**
 * Trims paths in the stack trace line based on whether it includes 'o1js' or 'opam'.
 *
 * @param stacktracePath - The stack trace line containing the path to trim.
 * @returns The trimmed stack trace line.
 */
function trimPaths(stacktracePath: string) {
  const includesO1js = stacktracePath.includes('o1js');
  if (includesO1js) {
    return trimO1jsPath(stacktracePath);
  }

  const includesOpam = stacktracePath.includes('opam');
  if (includesOpam) {
    return trimOpamPath(stacktracePath);
  }

  const includesWorkspace = stacktracePath.includes('workspace_root');
  if (includesWorkspace) {
    return trimWorkspacePath(stacktracePath);
  }

  return stacktracePath;
}

/**
 * Trims the 'o1js' portion of the stack trace line's path.
 *
 * @param stacktraceLine - The stack trace line containing the 'o1js' path to trim.
 * @returns The stack trace line with the trimmed 'o1js' path.
 */
function trimO1jsPath(stacktraceLine: string) {
  const fullPath = getDirectoryPath(stacktraceLine);
  if (!fullPath) {
    return stacktraceLine;
  }
  const o1jsIndex = fullPath.indexOf('o1js');
  if (o1jsIndex === -1) {
    return stacktraceLine;
  }

  // Grab the text before the parentheses as the prefix
  const prefix = stacktraceLine.slice(0, stacktraceLine.indexOf('(') + 1);
  // Grab the text including and after the o1js path
  const updatedPath = fullPath.slice(o1jsIndex);
  return &nbsp;${prefix}${updatedPath})&nbsp;;
}

/**
 * Trims the 'opam' portion of the stack trace line's path.
 *
 * @param stacktraceLine - The stack trace line containing the 'opam' path to trim.
 * @returns The stack trace line with the trimmed 'opam' path.
 */
function trimOpamPath(stacktraceLine: string) {
  const fullPath = getDirectoryPath(stacktraceLine);
  if (!fullPath) {
    return stacktraceLine;
  }
  const opamIndex = fullPath.indexOf('opam');
  if (opamIndex === -1) {
    return stacktraceLine;
  }

  const updatedPathArray = fullPath.slice(opamIndex).split('/');
  const libIndex = updatedPathArray.lastIndexOf('lib');
  if (libIndex === -1) {
    return stacktraceLine;
  }

  // Grab the text before the parentheses as the prefix
  const prefix = stacktraceLine.slice(0, stacktraceLine.indexOf('(') + 1);
  // Grab the text including and after the opam path, removing the lib directory
  const trimmedPath = updatedPathArray.slice(libIndex + 1);
  // Add the ocaml directory to the beginning of the path
  trimmedPath.unshift('ocaml');
  return &nbsp;${prefix}${trimmedPath.join('/')})&nbsp;;
}

/**
 * Trims the 'workspace_root' portion of the stack trace line's path.
 *
 * @param stacktraceLine - The stack trace line containing the 'workspace_root' path to trim.
 * @returns The stack trace line with the trimmed 'workspace_root' path.
 */
function trimWorkspacePath(stacktraceLine: string) {
  const fullPath = getDirectoryPath(stacktraceLine);
  if (!fullPath) {
    return stacktraceLine;
  }
  const workspaceIndex = fullPath.indexOf('workspace_root');
  if (workspaceIndex === -1) {
    return stacktraceLine;
  }

  const updatedPathArray = fullPath.slice(workspaceIndex).split('/');
  const prefix = stacktraceLine.slice(0, stacktraceLine.indexOf('(') + 1);
  const trimmedPath = updatedPathArray.slice(workspaceIndex);
  return &nbsp;${prefix}${trimmedPath.join('/')})&nbsp;;
}

/**
 * Extracts the directory path from a stack trace line.
 *
 * @param stacktraceLine - The stack trace line to extract the path from.
 * @returns The extracted directory path or undefined if not found.
 */
function getDirectoryPath(stacktraceLine: string) {
  // Regex to match the path inside the parentheses (e.g. (/home/../o1js/../*.ts))
  const fullPathRegex = /\(([^)]+)\)/;
  const matchedPaths = stacktraceLine.match(fullPathRegex);
  if (matchedPaths) {
    return matchedPaths[1];
  }
}

/**
 * An error that was assumed cannot happen, and communicates to users that it's not their fault but an
internal bug.
 */
function Bug(message: string) {
  return Error(
    &nbsp;${message}\nThis shouldn't have happened and indicates an internal bug.&nbsp;
  );
}
/**
 * Make an assertion. When failing, this will communicate to users it's not their fault but indicates an internal
bug.
 */
function assert(
  condition: boolean,
  message = 'Failed assertion.'
): asserts condition {
  if (!condition) throw Bug(message);
}
```

</file>

<file>

## path: /src/lib/events.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/events.ts

```
import { prefixes } from '../bindings/crypto/constants.js';
import { prefixToField } from '../bindings/lib/binable.js';
import {
  GenericField,
  GenericProvableExtended,
} from '../bindings/lib/generic.js';

export { createEvents, dataAsHash };

type Poseidon<Field> = {
  update(state: Field[], input: Field[]): Field[];
};

function createEvents<Field>({
  Field,
  Poseidon,
}: {
  Field: GenericField<Field>;
  Poseidon: Poseidon<Field>;
}) {
  type Event = Field[];

  type Events = {
    hash: Field;
    data: Event[];
  };

  // hashing helpers
  function initialState() {
    return [Field(0), Field(0), Field(0)] as [Field, Field, Field];
  }
  function salt(prefix: string) {
    return Poseidon.update(initialState(), [prefixToField(Field, prefix)]);
  }
  function hashWithPrefix(prefix: string, input: Field[]) {
    let init = salt(prefix);
    return Poseidon.update(init, input)[0];
  }
  function emptyHashWithPrefix(prefix: string) {
    return salt(prefix)[0];
  }

  const Events = {
    empty(): Events {
      let hash = emptyHashWithPrefix('MinaZkappEventsEmpty');
      return { hash, data: [] };
    },
    pushEvent(events: Events, event: Event): Events {
      let eventHash = hashWithPrefix(prefixes.event, event);
      let hash = hashWithPrefix(prefixes.events, [events.hash, eventHash]);
      return { hash, data: [event, ...events.data] };
    },
    fromList(events: Event[]): Events {
      return [...events].reverse().reduce(Events.pushEvent, Events.empty());
    },
    hash(events: Event[]) {
      return Events.fromList(events).hash;
    },
  };
  const EventsProvable = {
    ...Events,
    ...dataAsHash({
      emptyValue: Events.empty,
      toJSON(data: Field[][]) {
        return data.map((row) => row.map((e) => Field.toJSON(e)));
      },
      fromJSON(json: string[][]) {
        let data = json.map((row) => row.map((e) => Field.fromJSON(e)));
        let hash = Events.hash(data);
        return { data, hash };
      },
    }),
  };

  const Actions = {
    // same as events but w/ different hash prefixes
    empty(): Events {
      let hash = emptyHashWithPrefix('MinaZkappActionsEmpty');
      return { hash, data: [] };
    },
    pushEvent(actions: Events, event: Event): Events {
      let eventHash = hashWithPrefix(prefixes.event, event);
      let hash = hashWithPrefix(prefixes.sequenceEvents, [
        actions.hash,
        eventHash,
      ]);
      return { hash, data: [event, ...actions.data] };
    },
    fromList(events: Event[]): Events {
      return [...events].reverse().reduce(Actions.pushEvent, Actions.empty());
    },
    hash(events: Event[]) {
      return this.fromList(events).hash;
    },
    // different than events
    emptyActionState() {
      return emptyHashWithPrefix('MinaZkappActionStateEmptyElt');
    },
    updateSequenceState(state: Field, sequenceEventsHash: Field) {
      return hashWithPrefix(prefixes.sequenceEvents, [
        state,
        sequenceEventsHash,
      ]);
    },
  };

  const SequenceEventsProvable = {
    ...Actions,
    ...dataAsHash({
      emptyValue: Actions.empty,
      toJSON(data: Field[][]) {
        return data.map((row) => row.map((e) => Field.toJSON(e)));
      },
      fromJSON(json: string[][]) {
        let data = json.map((row) => row.map((e) => Field.fromJSON(e)));
        let hash = Actions.hash(data);
        return { data, hash };
      },
    }),
  };

  return { Events: EventsProvable, Actions: SequenceEventsProvable };
}

function dataAsHash<T, J, Field>({
  emptyValue,
  toJSON,
  fromJSON,
}: {
  emptyValue: () => { data: T; hash: Field };
  toJSON: (value: T) => J;
  fromJSON: (json: J) => { data: T; hash: Field };
}): GenericProvableExtended<{ data: T; hash: Field }, J, Field> & {
  emptyValue(): { data: T; hash: Field };
} {
  return {
    emptyValue,
    sizeInFields() {
      return 1;
    },
    toFields({ hash }) {
      return [hash];
    },
    toAuxiliary(value) {
      return [value?.data ?? emptyValue().data];
    },
    fromFields([hash], [data]) {
      return { data, hash };
    },
    toJSON({ data }) {
      return toJSON(data);
    },
    fromJSON(json) {
      return fromJSON(json);
    },
    check() {},
    toInput({ hash }) {
      return { fields: [hash] };
    },
  };
}
```

</file>

<file>

## path: /src/lib/fetch.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/fetch.ts

```
import 'isomorphic-fetch';
import { Field } from './core.js';
import { UInt32, UInt64 } from './int.js';
import { Actions, TokenId } from './account_update.js';
import { PublicKey, PrivateKey } from './signature.js';
import { NetworkValue } from './precondition.js';
import { Types } from '../bindings/mina-transaction/types.js';
import { ActionStates } from './mina.js';
import { LedgerHash, EpochSeed, StateHash } from './base58-encodings.js';
import {
  Account,
  accountQuery,
  FetchedAccount,
  fillPartialAccount,
  parseFetchedAccount,
  PartialAccount,
} from './mina/account.js';

export {
  fetchAccount,
  fetchLastBlock,
  checkZkappTransaction,
  parseFetchedAccount,
  markAccountToBeFetched,
  markNetworkToBeFetched,
  markActionsToBeFetched,
  fetchMissingData,
  fetchTransactionStatus,
  TransactionStatus,
  EventActionFilterOptions,
  getCachedAccount,
  getCachedNetwork,
  getCachedActions,
  addCachedAccount,
  networkConfig,
  setGraphqlEndpoint,
  setGraphqlEndpoints,
  setMinaGraphqlFallbackEndpoints,
  setArchiveGraphqlEndpoint,
  setArchiveGraphqlFallbackEndpoints,
  setLightnetAccountManagerEndpoint,
  sendZkappQuery,
  sendZkapp,
  removeJsonQuotes,
  fetchEvents,
  fetchActions,
  Lightnet
};

type NetworkConfig = {
  minaEndpoint: string;
  minaFallbackEndpoints: string[];
  archiveEndpoint: string;
  archiveFallbackEndpoints: string[];
  lightnetAccountManagerEndpoint: string;
};

let networkConfig = {
  minaEndpoint: '',
  minaFallbackEndpoints: [] as string[],
  archiveEndpoint: '',
  archiveFallbackEndpoints: [] as string[],
  lightnetAccountManagerEndpoint: '',
} satisfies NetworkConfig;

function checkForValidUrl(url: string) {
  try {
    new URL(url);
    return true;
  } catch (e) {
    return false;
  }
}

function setGraphqlEndpoints([
  graphqlEndpoint,
  ...fallbackEndpoints
]: string[]) {
  setGraphqlEndpoint(graphqlEndpoint);
  setMinaGraphqlFallbackEndpoints(fallbackEndpoints);
}
function setGraphqlEndpoint(graphqlEndpoint: string) {
  if (!checkForValidUrl(graphqlEndpoint)) {
    throw new Error(
      &nbsp;Invalid GraphQL endpoint: ${graphqlEndpoint}. Please specify a valid URL.&nbsp;
    );
  }
  networkConfig.minaEndpoint = graphqlEndpoint;
}
function setMinaGraphqlFallbackEndpoints(graphqlEndpoints: string[]) {
  if (graphqlEndpoints.some((endpoint) => !checkForValidUrl(endpoint))) {
    throw new Error(
      &nbsp;Invalid GraphQL endpoint: ${graphqlEndpoints}. Please specify a valid URL.&nbsp;
    );
  }
  networkConfig.minaFallbackEndpoints = graphqlEndpoints;
}

/**
 * Sets up a GraphQL endpoint to be used for fetching information from an Archive Node.
 *
 * @param A GraphQL endpoint.
 */
function setArchiveGraphqlEndpoint(graphqlEndpoint: string) {
  if (!checkForValidUrl(graphqlEndpoint)) {
    throw new Error(
      &nbsp;Invalid GraphQL endpoint: ${graphqlEndpoint}. Please specify a valid URL.&nbsp;
    );
  }
  networkConfig.archiveEndpoint = graphqlEndpoint;
}
function setArchiveGraphqlFallbackEndpoints(graphqlEndpoints: string[]) {
  if (graphqlEndpoints.some((endpoint) => !checkForValidUrl(endpoint))) {
    throw new Error(
      &nbsp;Invalid GraphQL endpoint: ${graphqlEndpoints}. Please specify a valid URL.&nbsp;
    );
  }
  networkConfig.archiveFallbackEndpoints = graphqlEndpoints;
}

/**
 * Sets up the lightnet account manager endpoint to be used for accounts acquisition and releasing.
 *
 * @param endpoint Account manager endpoint.
 */
function setLightnetAccountManagerEndpoint(endpoint: string) {
  if (!checkForValidUrl(endpoint)) {
    throw new Error(
      &nbsp;Invalid account manager endpoint: ${endpoint}. Please specify a valid URL.&nbsp;
    );
  }
  networkConfig.lightnetAccountManagerEndpoint = endpoint;
}

/**
 * Gets account information on the specified publicKey by performing a GraphQL query
 * to the specified endpoint. This will call the 'GetAccountInfo' query which fetches
 * zkapp related account information.
 *
 * If an error is returned by the specified endpoint, an error is thrown. Otherwise,
 * the data is returned.
 *
 * @param publicKey The specified publicKey to get account information on
 * @param tokenId The specified tokenId to get account information on
 * @param graphqlEndpoint The graphql endpoint to fetch from
 * @param config An object that exposes an additional timeout option
 * @returns zkapp information on the specified account or an error is thrown
 */
async function fetchAccount(
  accountInfo: { publicKey: string | PublicKey; tokenId?: string | Field },
  graphqlEndpoint = networkConfig.minaEndpoint,
  { timeout = defaultTimeout } = {}
): Promise<
  | { account: Types.Account; error: undefined }
  | { account: undefined; error: FetchError }
> {
  let publicKeyBase58 =
    accountInfo.publicKey instanceof PublicKey
      ? accountInfo.publicKey.toBase58()
      : accountInfo.publicKey;
  let tokenIdBase58 =
    typeof accountInfo.tokenId === 'string' || !accountInfo.tokenId
      ? accountInfo.tokenId
      : TokenId.toBase58(accountInfo.tokenId);

  return await fetchAccountInternal(
    { publicKey: publicKeyBase58, tokenId: tokenIdBase58 },
    graphqlEndpoint,
    {
      timeout,
    }
  );
}

// internal version of fetchAccount which does the same, but returns the original JSON version
// of the account, to save some back-and-forth conversions when caching accounts
async function fetchAccountInternal(
  accountInfo: { publicKey: string; tokenId?: string },
  graphqlEndpoint = networkConfig.minaEndpoint,
  config?: FetchConfig
) {
  const { publicKey, tokenId } = accountInfo;
  let [response, error] = await makeGraphqlRequest(
    accountQuery(publicKey, tokenId ?? TokenId.toBase58(TokenId.default)),
    graphqlEndpoint,
    networkConfig.minaFallbackEndpoints,
    config
  );
  if (error !== undefined) return { account: undefined, error };
  let fetchedAccount = (response as FetchResponse).data
    .account as FetchedAccount | null;
  if (fetchedAccount === null) {
    return {
      account: undefined,
      error: {
        statusCode: 404,
        statusText: &nbsp;fetchAccount: Account with public key ${publicKey} does not exist.&nbsp;,
      },
    };
  }
  let account = parseFetchedAccount(fetchedAccount);
  // account successfully fetched - add to cache before returning
  addCachedAccountInternal(account, graphqlEndpoint);
  return {
    account,
    error: undefined,
  };
}

type FetchConfig = { timeout?: number };
type FetchResponse = { data: any; errors?: any };
type FetchError = {
  statusCode: number;
  statusText: string;
};
type ActionStatesStringified = {
  [K in keyof ActionStates]: string;
};
// Specify 5min as the default timeout
const defaultTimeout = 5 * 60 * 1000;

let accountCache = {} as Record<
  string,
  {
    account: Account;
    graphqlEndpoint: string;
    timestamp: number;
  }
>;
let networkCache = {} as Record<
  string,
  {
    network: NetworkValue;
    graphqlEndpoint: string;
    timestamp: number;
  }
>;
let actionsCache = {} as Record<
  string,
  {
    actions: { hash: string; actions: string[][] }[];
    graphqlEndpoint: string;
    timestamp: number;
  }
>;
let accountsToFetch = {} as Record<
  string,
  { publicKey: string; tokenId: string; graphqlEndpoint: string }
```

```
>;
let networksToFetch = {} as Record<string, { graphqlEndpoint: string }>;
let actionsToFetch = {} as Record<
  string,
  {
    publicKey: string;
    tokenId: string;
    actionStates: ActionStatesStringified;
    graphqlEndpoint: string;
  }
>;

function markAccountToBeFetched(
  publicKey: PublicKey,
  tokenId: Field,
  graphqlEndpoint: string
) {
  let publicKeyBase58 = publicKey.toBase58();
  let tokenBase58 = TokenId.toBase58(tokenId);
  accountsToFetch[&nbsp;${publicKeyBase58};${tokenBase58};${graphqlEndpoint}&nbsp;] = {
    publicKey: publicKeyBase58,
    tokenId: tokenBase58,
    graphqlEndpoint,
  };
}
function markNetworkToBeFetched(graphqlEndpoint: string) {
  networksToFetch[graphqlEndpoint] = { graphqlEndpoint };
}
function markActionsToBeFetched(
  publicKey: PublicKey,
  tokenId: Field,
  graphqlEndpoint: string,
  actionStates: ActionStates = {}
) {
  let publicKeyBase58 = publicKey.toBase58();
  let tokenBase58 = TokenId.toBase58(tokenId);
  let { fromActionState, endActionState } = actionStates;
  let fromActionStateBase58 = fromActionState
    ? fromActionState.toString()
    : undefined;
  let endActionStateBase58 = endActionState
    ? endActionState.toString()
    : undefined;

  actionsToFetch[&nbsp;${publicKeyBase58};${tokenBase58};${graphqlEndpoint}&nbsp;] = {
    publicKey: publicKeyBase58,
    tokenId: tokenBase58,
    actionStates: {
      fromActionState: fromActionStateBase58,
      endActionState: endActionStateBase58,
    },
    graphqlEndpoint,
  };
}

async function fetchMissingData(
  graphqlEndpoint: string,
  archiveEndpoint?: string
) {
  let promises = Object.entries(accountsToFetch).map(
    async ([key, { publicKey, tokenId }]) => {
      let response = await fetchAccountInternal(
        { publicKey, tokenId },
        graphqlEndpoint
      );
      if (response.error === undefined) delete accountsToFetch[key];
    }
  );
  let actionPromises = Object.entries(actionsToFetch).map(
    async ([key, { publicKey, actionStates, tokenId }]) => {
      let response = await fetchActions(
        { publicKey, actionStates, tokenId },
        archiveEndpoint
      );
      if (!('error' in response) || response.error === undefined)
        delete actionsToFetch[key];
    }
  );
  promises.push(...actionPromises);
  let network = Object.entries(networksToFetch).find(([, network]) => {
    return network.graphqlEndpoint === graphqlEndpoint;
  });
  if (network !== undefined) {
    promises.push(
      (async () => {
        try {
          await fetchLastBlock(graphqlEndpoint);
          delete networksToFetch[network[0]];
        } catch {}
      })()
    );
  }
  await Promise.all(promises);
}

function getCachedAccount(
  publicKey: PublicKey,
  tokenId: Field,
  graphqlEndpoint = networkConfig.minaEndpoint
): Account | undefined {
  return accountCache[accountCacheKey(publicKey, tokenId, graphqlEndpoint)]
    ?.account;
}
```

```
function getCachedNetwork(graphqlEndpoint = networkConfig.minaEndpoint) {
  return networkCache[graphqlEndpoint]?.network;
}

function getCachedActions(
  publicKey: PublicKey,
  tokenId: Field,
  graphqlEndpoint = networkConfig.archiveEndpoint
) {
  return actionsCache[accountCacheKey(publicKey, tokenId, graphqlEndpoint)]
    ?.actions;
}

/**
 * Adds an account to the local cache, indexed by a GraphQL endpoint.
 */
function addCachedAccount(
  partialAccount: PartialAccount,
  graphqlEndpoint = networkConfig.minaEndpoint
) {
  let account = fillPartialAccount(partialAccount);
  addCachedAccountInternal(account, graphqlEndpoint);
}

function addCachedAccountInternal(account: Account, graphqlEndpoint: string) {
  accountCache[
    accountCacheKey(account.publicKey, account.tokenId, graphqlEndpoint)
  ] = {
    account,
    graphqlEndpoint,
    timestamp: Date.now(),
  };
}

function addCachedActions(
  { publicKey, tokenId }: { publicKey: string; tokenId: string },
  actions: { hash: string; actions: string[][] }[],
  graphqlEndpoint: string
) {
  actionsCache[&nbsp;${publicKey};${tokenId};${graphqlEndpoint}&nbsp;] = {
    actions,
    graphqlEndpoint,
    timestamp: Date.now(),
  };
}

function accountCacheKey(
  publicKey: PublicKey,
  tokenId: Field,
  graphqlEndpoint: string
) {
  return &nbsp;${publicKey.toBase58()};${TokenId.toBase58(
    tokenId
  )};${graphqlEndpoint}&nbsp;;
}

/**
 * Fetches the last block on the Mina network.
 */
async function fetchLastBlock(graphqlEndpoint = networkConfig.minaEndpoint) {
  let [resp, error] = await makeGraphqlRequest(
    lastBlockQuery,
    graphqlEndpoint,
    networkConfig.minaFallbackEndpoints
  );
  if (error) throw Error(error.statusText);
  let lastBlock = resp?.data?.bestChain?.[0];
  if (lastBlock === undefined) {
    throw Error('Failed to fetch latest network state.');
  }
  let network = parseFetchedBlock(lastBlock);
  networkCache[graphqlEndpoint] = {
    network,
    graphqlEndpoint,
    timestamp: Date.now(),
  };
  return network;
}

const lastBlockQuery = &nbsp;{
  bestChain(maxLength: 1) {
    protocolState {
      blockchainState {
        snarkedLedgerHash
        stagedLedgerHash
        date
        utcDate
        stagedLedgerProofEmitted
      }
      previousStateHash
      consensusState {
        blockHeight
        slotSinceGenesis
        slot
        nextEpochData {
          ledger {hash totalCurrency}
          seed
          startCheckpoint
          lockCheckpoint
          epochLength
        }
        stakingEpochData {
          ledger {hash totalCurrency}
          seed
          startCheckpoint
          lockCheckpoint
          epochLength
        }
        epochCount
        minWindowDensity
        totalCurrency
        epoch
      }
    }
  }
}&nbsp;;

type LastBlockQueryFailureCheckResponse = {
  bestChain: {
    transactions: {
      zkappCommands: {
        hash: string;
        failureReason: {
          failures: string[];
          index: number;
        }[];
      }[];
    };
  }[];
};

const lastBlockQueryFailureCheck = &nbsp;{
  bestChain(maxLength: 1) {
    transactions {
      zkappCommands {
        hash
        failureReason {
          failures
          index
        }
      }
    }
  }
}&nbsp;;

async function fetchLatestBlockZkappStatus(
  graphqlEndpoint = networkConfig.minaEndpoint
) {
  let [resp, error] = await makeGraphqlRequest(
    lastBlockQueryFailureCheck,
    graphqlEndpoint,
    networkConfig.minaFallbackEndpoints
  );
  if (error) throw Error(&nbsp;Error making GraphQL request: ${error.statusText}&nbsp;);
  let bestChain = resp?.data as LastBlockQueryFailureCheckResponse;
  if (bestChain === undefined) {
    throw Error(
      'Failed to fetch the latest zkApp transaction status. The response data is undefined.'
    );
  }
  return bestChain;
}

async function checkZkappTransaction(txnId: string) {
  let bestChainBlocks = await fetchLatestBlockZkappStatus();

  for (let block of bestChainBlocks.bestChain) {
    for (let zkappCommand of block.transactions.zkappCommands) {
      if (zkappCommand.hash === txnId) {
        if (zkappCommand.failureReason !== null) {
          let failureReason = zkappCommand.failureReason
            .reverse()
            .map((failure) => {
              return &nbsp; AccountUpdate #${
                failure.index
              } failed. Reason: "${failure.failures.join(', ')}"&nbsp;;
            });
          return {
            success: false,
            failureReason,
          };
        } else {
          return {
            success: true,
            failureReason: null,
          };
        }
      }
    }
  }
  return {
    success: false,
    failureReason: null,
  };
}

type FetchedBlock = {
  protocolState: {
    blockchainState: {
      snarkedLedgerHash: string; // hash-like encoding
      stagedLedgerHash: string; // hash-like encoding
      date: string; // String(Date.now())
      utcDate: string; // String(Date.now())
      stagedLedgerProofEmitted: boolean; // bool
    };
    previousStateHash: string; // hash-like encoding
    consensusState: {
      blockHeight: string; // String(number)
      slotSinceGenesis: string; // String(number)
      slot: string; // String(number)
      nextEpochData: {
        ledger: {
          hash: string; // hash-like encoding
          totalCurrency: string; // String(number)
        };
        seed: string; // hash-like encoding
        startCheckpoint: string; // hash-like encoding
        lockCheckpoint: string; // hash-like encoding
        epochLength: string; // String(number)
      };
      stakingEpochData: {
        ledger: {
          hash: string; // hash-like encoding
          totalCurrency: string; // String(number)
        };
        seed: string; // hash-like encoding
        startCheckpoint: string; // hash-like encoding
        lockCheckpoint: string; // hash-like encoding
        epochLength: string; // String(number)
      };
      epochCount: string; // String(number)
      minWindowDensity: string; // String(number)
      totalCurrency: string; // String(number)
      epoch: string; // String(number)
    };
  };
};

function parseFetchedBlock({
  protocolState: {
    blockchainState: { snarkedLedgerHash, utcDate },
    consensusState: {
      blockHeight,
      minWindowDensity,
      totalCurrency,
      slot,
      slotSinceGenesis,
      nextEpochData,
      stakingEpochData,
    },
  },
}: FetchedBlock): NetworkValue {
  return {
    snarkedLedgerHash: LedgerHash.fromBase58(snarkedLedgerHash),
    // TODO: use date or utcDate?
    blockchainLength: UInt32.from(blockHeight),
    minWindowDensity: UInt32.from(minWindowDensity),
    totalCurrency: UInt64.from(totalCurrency),
    globalSlotSinceGenesis: UInt32.from(slotSinceGenesis),
    nextEpochData: parseEpochData(nextEpochData),
    stakingEpochData: parseEpochData(stakingEpochData),
  };
}

function parseEpochData({
  ledger: { hash, totalCurrency },
  seed,
  startCheckpoint,
  lockCheckpoint,
  epochLength,
}: FetchedBlock['protocolState']['consensusState']['nextEpochData']): NetworkValue['nextEpochData'] {
  return {
    ledger: {
      hash: LedgerHash.fromBase58(hash),
      totalCurrency: UInt64.from(totalCurrency),
    },
    seed: EpochSeed.fromBase58(seed),
    startCheckpoint: StateHash.fromBase58(startCheckpoint),
    lockCheckpoint: StateHash.fromBase58(lockCheckpoint),
    epochLength: UInt32.from(epochLength),
  };
}

const transactionStatusQuery = (txId: string) => &nbsp;query {
  transactionStatus(zkappTransaction:"${txId}")
}&nbsp;;

/**
 * Fetches the status of a transaction.
 */
async function fetchTransactionStatus(
  txId: string,
  graphqlEndpoint = networkConfig.minaEndpoint
): Promise<TransactionStatus> {
  let [resp, error] = await makeGraphqlRequest(
    transactionStatusQuery(txId),
    graphqlEndpoint,
    networkConfig.minaFallbackEndpoints
  );
  if (error) throw Error(error.statusText);
  let txStatus = resp?.data?.transactionStatus;
  if (txStatus === undefined || txStatus === null) {
    throw Error(&nbsp;Failed to fetch transaction status. TransactionId: ${txId}&nbsp;);
  }
  return txStatus as TransactionStatus;
}

/**
 * INCLUDED: A transaction that is on the longest chain
 *
 * PENDING: A transaction either in the transition frontier or in transaction pool but is not on the longest
chain
```

```
 *
 * UNKNOWN: The transaction has either been snarked, reached finality through consensus or has been
dropped
 *
 */
type TransactionStatus = 'INCLUDED' | 'PENDING' | 'UNKNOWN';

/**
 * Sends a zkApp command (transaction) to the specified GraphQL endpoint.
 */
function sendZkapp(
  json: string,
  graphqlEndpoint = networkConfig.minaEndpoint,
  { timeout = defaultTimeout } = {}
) {
  return makeGraphqlRequest(
    sendZkappQuery(json),
    graphqlEndpoint,
    networkConfig.minaFallbackEndpoints,
    {
      timeout,
    }
  );
}

// TODO: Decide an appropriate response structure.
function sendZkappQuery(json: string) {
  return &nbsp;mutation {
  sendZkapp(input: {
    zkappCommand: ${removeJsonQuotes(json)}
  }) {
    zkapp {
      hash
      id
      failureReason {
        failures
        index
      }
      zkappCommand {
        memo
        feePayer {
          body {
            publicKey
          }
        }
        accountUpdates {
          body {
            publicKey
            useFullCommitment
            incrementNonce
          }
        }
      }
    }
  }
}
&nbsp;;
}
type FetchedEvents = {
  blockInfo: {
    distanceFromMaxBlockHeight: number;
    globalSlotSinceGenesis: number;
    height: number;
    stateHash: string;
    parentHash: string;
    chainStatus: string;
  };
  eventData: {
    transactionInfo: {
      hash: string;
      memo: string;
      status: string;
    };
    data: string[];
  }[];
};
type FetchedActions = {
  blockInfo: {
    distanceFromMaxBlockHeight: number;
  };
  actionState: {
    actionStateOne: string;
    actionStateTwo: string;
  };
  actionData: {
    accountUpdateId: string;
    data: string[];
  }[];
};

type EventActionFilterOptions = {
  to?: UInt32;
  from?: UInt32;
};

const getEventsQuery = (
  publicKey: string,
  tokenId: string,
  filterOptions?: EventActionFilterOptions
) => {
  const { to, from } = filterOptions ?? {};
  let input = &nbsp;address: "${publicKey}", tokenId: "${tokenId}"&nbsp;;
  if (to !== undefined) {
    input += &nbsp;, to: ${to}&nbsp;;
  }
  if (from !== undefined) {
    input += &nbsp;, from: ${from}&nbsp;;
  }
  return &nbsp;{
  events(input: { ${input} }) {
    blockInfo {
      distanceFromMaxBlockHeight
      height
      globalSlotSinceGenesis
      stateHash
      parentHash
      chainStatus
    }
    eventData {
      transactionInfo {
        hash
        memo
        status
      }
      data
    }
  }
}&nbsp;;
};
const getActionsQuery = (
  publicKey: string,
  actionStates: ActionStatesStringified,
  tokenId: string,
  _filterOptions?: EventActionFilterOptions
) => {
  const { fromActionState, endActionState } = actionStates ?? {};
  let input = &nbsp;address: "${publicKey}", tokenId: "${tokenId}"&nbsp;;
  if (fromActionState !== undefined) {
    input += &nbsp;, fromActionState: "${fromActionState}"&nbsp;;
  }
  if (endActionState !== undefined) {
    input += &nbsp;, endActionState: "${endActionState}"&nbsp;;
  }
  return &nbsp;{
  actions(input: { ${input} }) {
    blockInfo {
      distanceFromMaxBlockHeight
    }
    actionState {
      actionStateOne
      actionStateTwo
    }
    actionData {
      accountUpdateId
      data
    }
  }
}&nbsp;;
};

/**
 * Asynchronously fetches event data for an account from the Mina Archive Node GraphQL API.
 * @async
 * @param accountInfo - The account information object.
 * @param accountInfo.publicKey - The account public key.
 * @param [accountInfo.tokenId] - The optional token ID for the account.
 * @param [graphqlEndpoint=networkConfig.archiveEndpoint] - The GraphQL endpoint to query. Defaults to
the Archive Node GraphQL API.
 * @param [filterOptions={}] - The optional filter options object.
 * @returns A promise that resolves to an array of objects containing event data, block information and
transaction information for the account.
 * @throws If the GraphQL request fails or the response is invalid.
 * @example
 * const accountInfo = { publicKey:
'B62qiwmXrWn7Cok5VhhB3KvCwyZ7NHHstFGbiU5n7m8s2RqqNW1p1wF' };
 * const events = await fetchEvents(accountInfo);
 * console.log(events);
 */
async function fetchEvents(
  accountInfo: { publicKey: string; tokenId?: string },
  graphqlEndpoint = networkConfig.archiveEndpoint,
  filterOptions: EventActionFilterOptions = {}
) {
  if (!graphqlEndpoint)
    throw new Error(
      'fetchEvents: Specified GraphQL endpoint is undefined. Please specify a valid endpoint.'
    );
  const { publicKey, tokenId } = accountInfo;
  let [response, error] = await makeGraphqlRequest(
    getEventsQuery(
      publicKey,
      tokenId ?? TokenId.toBase58(TokenId.default),
      filterOptions
    ),
    graphqlEndpoint,
    networkConfig.archiveFallbackEndpoints
  );
  if (error) throw Error(error.statusText);
  let fetchedEvents = response?.data.events as FetchedEvents[];
  if (fetchedEvents === undefined) {
    throw Error(
      &nbsp;Failed to fetch events data. Account: ${publicKey} Token: ${tokenId}&nbsp;
    );
  }

  // TODO: This is a temporary fix. We should be able to fetch the event/action data from any block at the
best tip.
  // Once https://github.com/o1-labs/Archive-Node-API/issues/7 is resolved, we can remove this.
  // If we have multiple blocks returned at the best tip (e.g. distanceFromMaxBlockHeight === 0),
  // then filter out the blocks at the best tip. This is because we cannot guarantee that every block
  // at the best tip will have the correct event data or guarantee that the specific block data will not
  // fork in anyway. If this happens, we delay fetching event data until another block has been added to the
network.
  let numberOfBestTipBlocks = 0;
  for (let i = 0; i < fetchedEvents.length; i++) {
    if (fetchedEvents[i].blockInfo.distanceFromMaxBlockHeight === 0) {
      numberOfBestTipBlocks++;
    }
    if (numberOfBestTipBlocks > 1) {
      fetchedEvents = fetchedEvents.filter((event) => {
        return event.blockInfo.distanceFromMaxBlockHeight !== 0;
      });
      break;
    }
  }

  return fetchedEvents.map((event) => {
    let events = event.eventData.map(({ data, transactionInfo }) => {
      return {
        data,
        transactionInfo,
      };
    });

    return {
      events,
      blockHeight: UInt32.from(event.blockInfo.height),
      blockHash: event.blockInfo.stateHash,
      parentBlockHash: event.blockInfo.parentHash,
      globalSlot: UInt32.from(event.blockInfo.globalSlotSinceGenesis),
      chainStatus: event.blockInfo.chainStatus,
    };
  });
}

async function fetchActions(
  accountInfo: {
    publicKey: string;
    actionStates: ActionStatesStringified;
    tokenId?: string;
  },
  graphqlEndpoint = networkConfig.archiveEndpoint
) {
  if (!graphqlEndpoint)
    throw new Error(
      'fetchActions: Specified GraphQL endpoint is undefined. Please specify a valid endpoint.'
    );
  const {
    publicKey,
    actionStates,
    tokenId = TokenId.toBase58(TokenId.default),
  } = accountInfo;
  let [response, error] = await makeGraphqlRequest(
    getActionsQuery(publicKey, actionStates, tokenId),
    graphqlEndpoint,
    networkConfig.archiveFallbackEndpoints
  );
  if (error) throw Error(error.statusText);
  let fetchedActions = response?.data.actions as FetchedActions[];
  if (fetchedActions === undefined) {
    return {
      error: {
        statusCode: 404,
        statusText: &nbsp;fetchActions: Account with public key ${publicKey} with tokenId ${tokenId} does not
exist.&nbsp;,
      },
    };
  }

  // TODO: This is a temporary fix. We should be able to fetch the event/action data from any block at the
best tip.
  // Once https://github.com/o1-labs/Archive-Node-API/issues/7 is resolved, we can remove this.
  // If we have multiple blocks returned at the best tip (e.g. distanceFromMaxBlockHeight === 0),
  // then filter out the blocks at the best tip. This is because we cannot guarantee that every block
  // at the best tip will have the correct action data or guarantee that the specific block data will not
  // fork in anyway. If this happens, we delay fetching action data until another block has been added to the
network.
  let numberOfBestTipBlocks = 0;
  for (let i = 0; i < fetchedActions.length; i++) {
    if (fetchedActions[i].blockInfo.distanceFromMaxBlockHeight === 0) {
      numberOfBestTipBlocks++;
    }
    if (numberOfBestTipBlocks > 1) {
      fetchedActions = fetchedActions.filter((action) => {
        return action.blockInfo.distanceFromMaxBlockHeight !== 0;
      });
      break;
    }
  }
  // Archive Node API returns actions in the latest order, so we reverse the array to get the actions in
chronological order.
  fetchedActions.reverse();
  let actionsList: { actions: string[][]; hash: string }[] = [];

  // correct for archive node sending one block too many
  if (
    fetchedActions.length !== 0 &&
    fetchedActions[0].actionState.actionStateOne ===
      actionStates.fromActionState
  ) {
    fetchedActions = fetchedActions.slice(1);
  }
  fetchedActions.forEach((actionBlock) => {
    let { actionData } = actionBlock;
    let latestActionState = Field(actionBlock.actionState.actionStateTwo);
    let actionState = actionBlock.actionState.actionStateOne;

    if (actionData.length === 0)
      throw Error(
        &nbsp;No action data was found for the account ${publicKey} with the latest action state
${actionState}&nbsp;
      );

    // split actions by account update
    let actionsByAccountUpdate: string[][][] = [];
    let currentAccountUpdateId = 'none';
    let currentActions: string[][];
    actionData.forEach(({ accountUpdateId, data }) => {
      if (accountUpdateId === currentAccountUpdateId) {
        currentActions.push(data);
      } else {
        currentAccountUpdateId = accountUpdateId;
        currentActions = [data];
        actionsByAccountUpdate.push(currentActions);
      }
    });

    // re-hash actions
    for (let actions of actionsByAccountUpdate) {
      latestActionState = updateActionState(actions, latestActionState);
      actionsList.push({ actions, hash: latestActionState.toString() });
    }

    const finalActionState = latestActionState.toString();
    const expectedActionState = actionState;

    if (finalActionState !== expectedActionState) {
      throw new Error(
        &nbsp;Failed to derive correct actions hash for ${publicKey}.
        Derived hash: ${finalActionState}, expected hash: ${expectedActionState}).
        All action hashes derived: ${JSON.stringify(actionsList, null, 2)}
        Please try a different Archive Node API endpoint.
        &nbsp;
      );
    }
  });
  addCachedActions({ publicKey, tokenId }, actionsList, graphqlEndpoint);
  return actionsList;
}

namespace Lightnet {
  /**
   * Gets random key pair (public and private keys) from account manager
```

```
 * that operates with accounts configured in target network Genesis Ledger.
 *
 * If an error is returned by the specified endpoint, an error is thrown. Otherwise,
 * the data is returned.
 *
 * @param options.isRegularAccount Whether to acquire regular or zkApp account (one with already
configured verification key)
 * @param options.lightnetAccountManagerEndpoint Account manager endpoint to fetch from
 * @returns Key pair
 */
export async function acquireKeyPair(
  options: {
    isRegularAccount?: boolean;
    lightnetAccountManagerEndpoint?: string;
  } = {}
): Promise<{
  publicKey: PublicKey;
  privateKey: PrivateKey;
}> {
  const {
    isRegularAccount = true,
    lightnetAccountManagerEndpoint = networkConfig.lightnetAccountManagerEndpoint,
  } = options;
  const response = await fetch(
    &nbsp;${lightnetAccountManagerEndpoint}/acquire-account?
isRegularAccount=${isRegularAccount}&nbsp;,
    {
      method: 'GET',
      headers: {
        'Content-Type': 'application/json',
      },
    }
  );

  if (response.ok) {
    const data = await response.json();
    if (data) {
      return {
        publicKey: PublicKey.fromBase58(data.pk),
        privateKey: PrivateKey.fromBase58(data.sk),
      };
    }
  }

  throw new Error('Failed to acquire the key pair');
}

/**
 * Releases previously acquired key pair by public key.
 *
 * @param options.publicKey Public key of previously acquired key pair to release
 * @param options.lightnetAccountManagerEndpoint Account manager endpoint to fetch from
```

```
   * @returns Response message from the account manager as string or null if the request failed
   */
  export async function releaseKeyPair(options: {
    publicKey: string;
    lightnetAccountManagerEndpoint?: string;
  }): Promise<string | null> {
    const {
      publicKey,
      lightnetAccountManagerEndpoint = networkConfig.lightnetAccountManagerEndpoint,
    } = options;
    const response = await fetch(
      &nbsp;${lightnetAccountManagerEndpoint}/release-account&nbsp;,
      {
        method: 'PUT',
        headers: {
          'Content-Type': 'application/json',
        },
        body: JSON.stringify({
          pk: publicKey,
        }),
      }
    );

    if (response.ok) {
      const data = await response.json();
      if (data) {
        return data.message as string;
      }
    }

    return null;
  }
}

function updateActionState(actions: string[][], actionState: Field) {
  let actionHash = Actions.fromJSON(actions).hash;
  return Actions.updateSequenceState(actionState, actionHash);
}

// removes the quotes on JSON keys
function removeJsonQuotes(json: string) {
  let cleaned = JSON.stringify(JSON.parse(json), null, 2);
  return cleaned.replace(/\"(\S+)\"\s*:/gm, '$1:');
}

// TODO it seems we're not actually catching most errors here
async function makeGraphqlRequest(
  query: string,
  graphqlEndpoint = networkConfig.minaEndpoint,
  fallbackEndpoints: string[],
  { timeout = defaultTimeout } = {} as FetchConfig
) {
  if (graphqlEndpoint === 'none')
    throw Error(
      "Should have made a graphql request, but don't know to which endpoint. Try calling
&nbsp;setGraphqlEndpoint&nbsp; first."
    );
  let timeouts: NodeJS.Timeout[] = [];
  const clearTimeouts = () => {
    timeouts.forEach((t) => clearTimeout(t));
    timeouts = [];
  };

  const makeRequest = async (url: string) => {
    const controller = new AbortController();
    const timer = setTimeout(() => controller.abort(), timeout);
    timeouts.push(timer);
    let body = JSON.stringify({ operationName: null, query, variables: {} });
    try {
      let response = await fetch(url, {
        method: 'POST',
        headers: { 'Content-Type': 'application/json' },
        body,
        signal: controller.signal,
      });
      return checkResponseStatus(response);
    } finally {
      clearTimeouts();
    }
  };
  // try to fetch from endpoints in pairs
  let timeoutErrors: { url1: string; url2: string; error: any }[] = [];
  let urls = [graphqlEndpoint, ...fallbackEndpoints];
  for (let i = 0; i < urls.length; i += 2) {
    let url1 = urls[i];
    let url2 = urls[i + 1];
    if (url2 === undefined) {
      try {
        return await makeRequest(url1);
      } catch (error) {
        return [undefined, inferError(error)] as [undefined, FetchError];
      }
    }
    try {
      return await Promise.race([makeRequest(url1), makeRequest(url2)]);
    } catch (unknownError) {
      let error = inferError(unknownError);
      if (error.statusCode === 408) {
        // If the request timed out, try the next 2 endpoints
        timeoutErrors.push({ url1, url2, error });
      } else {
        // If the request failed for some other reason (e.g. o1js error), return the error
        return [undefined, error] as [undefined, FetchError];
      }
```

```
    }
  }
  const statusText = timeoutErrors
    .map(
      ({ url1, url2, error }) =>
        &nbsp;Request to ${url1} and ${url2} timed out. Error: ${error}&nbsp;
    )
    .join('\n');
  return [undefined, { statusCode: 408, statusText }] as [
    undefined,
    FetchError
  ];
}

async function checkResponseStatus(
  response: Response
): Promise<[FetchResponse, undefined] | [undefined, FetchError]> {
  if (response.ok) {
    let jsonResponse = await response.json();
    if (jsonResponse.errors && jsonResponse.errors.length > 0) {
      return [
        undefined,
        {
          statusCode: response.status,
          statusText: jsonResponse.errors
            .map((error: any) => error.message)
            .join('\n'),
        } as FetchError,
      ];
    } else if (jsonResponse.data === undefined) {
      return [
        undefined,
        {
          statusCode: response.status,
          statusText: &nbsp;GraphQL response data is undefined&nbsp;,
        } as FetchError,
      ];
    }
    return [jsonResponse as FetchResponse, undefined];
  } else {
    return [
      undefined,
      {
        statusCode: response.status,
        statusText: response.statusText,
      } as FetchError,
    ];
  }
}

function inferError(error: unknown): FetchError {
  let errorMessage = JSON.stringify(error);
  if (error instanceof AbortSignal) {
    return { statusCode: 408, statusText: &nbsp;Request Timeout: ${errorMessage}&nbsp; };
  } else {
    return {
      statusCode: 500,
      statusText: &nbsp;Unknown Error: ${errorMessage}&nbsp;,
    };
  }
}
```

</file>

<file>

## path: /src/lib/fetch.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/fetch.unit-test.ts

```
import { shutdown } from '../index.js';
import * as Fetch from './fetch.js';
import { expect } from 'expect';

console.log('testing regex helpers');

let input, actual, expected;

input = &nbsp;{
  "array": [
    {"data": "string with \\"escaped quotes\\": 1"}, { "otherdata": "otherstrin\\"g"}]
}&nbsp;;

expected = &nbsp;{
  array: [
    {
      data: "string with \\"escaped quotes\\": 1"
    },
    {
      otherdata: "otherstrin\\"g"
    }
  ]
}&nbsp;;

actual = Fetch.removeJsonQuotes(input);
expect(actual).toEqual(expected);

input = &nbsp;{
  "array": [
    {
      "data"      : "x"
    },
    {
```

```
    "data":             1},{
    "data": {
            "otherData":   "x"
    }
  }
 ]
}&nbsp;;

expected = &nbsp;{
 array: [
  {
    data: "x"
  },
  {
    data: 1
  },
  {
    data: {
      otherData: "x"
    }
  }
 ]
}&nbsp;;

actual = Fetch.removeJsonQuotes(input);
expect(actual).toEqual(expected);

input = &nbsp;{
 "FirstName" :"abc",
 "Email" : "a@a.com",
 "Id" : "1",
 "Phone" : "1234567890",
 "Date" : "2 May 2016 23:59:59"
}&nbsp;;

expected = &nbsp;{
 FirstName: "abc",
 Email: "a@a.com",
 Id: "1",
 Phone: "1234567890",
 Date: "2 May 2016 23:59:59"
}&nbsp;;

actual = Fetch.removeJsonQuotes(input);
expect(actual).toEqual(expected);

input = &nbsp;{
"FirstName":"abc",
"Email" : "a@a.com",
"Id" :      "1",
"Phone" :"1234567890",
"Date":
  "2 May 2016 23:59:59"
}&nbsp;;

expected = &nbsp;{
  FirstName: "abc",
  Email: "a@a.com",
  Id: "1",
  Phone: "1234567890",
  Date: "2 May 2016 23:59:59"
}&nbsp;;
actual = Fetch.removeJsonQuotes(input);

expect(actual).toEqual(expected);

input = &nbsp;{
  "First-Name":"abc",
  "Email" : "a@a.com",
  "Id" :      "1",
  "Phone" :"1234567890",
  "Date":
    "2 May 2016 23:59:59"
}&nbsp;;

expected = &nbsp;{
  First-Name: "abc",
  Email: "a@a.com",
  Id: "1",
  Phone: "1234567890",
  Date: "2 May 2016 23:59:59"
}&nbsp;;

actual = Fetch.removeJsonQuotes(input);

expect(actual).toEqual(expected);

console.log('regex tests complete      ');
shutdown();
```

</file>

<file>

## path: /src/lib/field.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/field.ts

```
import { Snarky, Provable } from '../snarky.js';
import { Field as Fp } from '../provable/field-bigint.js';
import { defineBinable } from '../bindings/lib/binable.js';
import type { NonNegativeInteger } from '../bindings/crypto/non-negative.js';
import { asProver, inCheckedComputation } from './provable-context.js';
import { Bool } from './bool.js';
import { assert } from './errors.js';

// external API
export { Field };

// internal API
export {
  ConstantField,
  FieldType,
  FieldVar,
  FieldConst,
  isField,
  withMessage,
  readVarMessage,
  toConstantField,
  toFp,
};

type FieldConst = [0, bigint];

function constToBigint(x: FieldConst): Fp {
  return x[1];
}
function constFromBigint(x: Fp): FieldConst {
  return [0, x];
}

const FieldConst = {
  fromBigint: constFromBigint,
  toBigint: constToBigint,
  equal(x: FieldConst, y: FieldConst) {
    return x[1] === y[1];
  },
  [0]: constFromBigint(0n),
  [1]: constFromBigint(1n),
  [-1]: constFromBigint(Fp(-1n)),
};

enum FieldType {
  Constant,
  Var,
  Add,
  Scale,
}

/**
 * &nbsp;FieldVar&nbsp; is the core data type in snarky. It is eqivalent to &nbsp;Cvar.t&nbsp; in OCaml.
 * It represents a field element that is part of provable code - either a constant or a variable.
 *
 * **Variables** end up filling the witness columns of a constraint system.
 * Think of a variable as a value that has to be provided by the prover, and that has to satisfy all the
 * constraints it is involved in.
 *
 * **Constants** end up being hard-coded into the constraint system as gate coefficients.
 * Think of a constant as a value that is known publicly, at compile time, and that defines the constraint
system.
 *
 * Both constants and variables can be combined into an AST using the Add and Scale combinators.
 */
type FieldVar =
  | [FieldType.Constant, FieldConst]
  | [FieldType.Var, number]
  | [FieldType.Add, FieldVar, FieldVar]
  | [FieldType.Scale, FieldConst, FieldVar];

type ConstantFieldVar = [FieldType.Constant, FieldConst];

const FieldVar = {
  constant(x: bigint | FieldConst): ConstantFieldVar {
    let x0 = typeof x === 'bigint' ? FieldConst.fromBigint(x) : x;
    return [FieldType.Constant, x0];
  },
  isConstant(x: FieldVar): x is ConstantFieldVar {
    return x[0] === FieldType.Constant;
  },
  // TODO: handle (special) constants
  add(x: FieldVar, y: FieldVar): FieldVar {
    return [FieldType.Add, x, y];
  },
  // TODO: handle (special) constants
  scale(c: FieldConst, x: FieldVar): FieldVar {
    return [FieldType.Scale, c, x];
  },
  [0]: [FieldType.Constant, FieldConst[0]] satisfies ConstantFieldVar,
  [1]: [FieldType.Constant, FieldConst[1]] satisfies ConstantFieldVar,
  [-1]: [FieldType.Constant, FieldConst[-1]] satisfies ConstantFieldVar,
};

type ConstantField = Field & { value: ConstantFieldVar };

/**
 * A {@link Field} is an element of a prime order [finite field](https://en.wikipedia.org/wiki/Finite_field).
 * Every other provable type is built using the {@link Field} type.
 *
 * The field is the [pasta base field](https://electriccoin.co/blog/the-pasta-curves-for-halo-2-and-beyond/) of
order 2^254 + 0x224698fc094cf91b992d30ed00000001 ({@link Field.ORDER}).
 *
 * You can create a new Field from everything "field-like" (&nbsp;bigint&nbsp;, integer &nbsp;number&nbsp;,
decimal &nbsp;string&nbsp;, &nbsp;Field&nbsp;).
 * @example
 * &nbsp;&nbsp;&nbsp;
 * Field(10n); // Field contruction from a big integer
 * Field(100); // Field construction from a number
 * Field("1"); // Field construction from a decimal string
 * &nbsp;&nbsp;&nbsp;
 *
 * **Beware**: Fields _cannot_ be constructed from fractional numbers or alphanumeric strings:
 * &nbsp;&nbsp;&nbsp;ts
 * Field(3.141); // ERROR: Cannot convert a float to a field element
 * Field("abc"); // ERROR: Invalid argument "abc"
 * &nbsp;&nbsp;&nbsp;
 *
 * Creating a Field from a negative number can result in unexpected behavior if you are not familiar with
[modular arithmetic](https://en.wikipedia.org/wiki/Modular_arithmetic).
 * @example
 * &nbsp;&nbsp;&nbsp;
 * const x = Field(-1); // Valid Field construction from negative number
 * const y = Field(Field.ORDER - 1n); // equivalent to &nbsp;x&nbsp;
 * &nbsp;&nbsp;&nbsp;
 *
 * **Important**: All the functions defined on a Field (arithmetic, logic, etc.) take their arguments as "field-
like". A Field itself is also defined as a "field-like" element.
 *
 * @param value - the value to convert to a {@link Field}
 *
 * @return A {@link Field} with the value converted from the argument
 */
class Field {
 value: FieldVar;

 /**
  * The order of the pasta curve that {@link Field} type build on as a &nbsp;bigint&nbsp;.
  * Order of the {@link Field} is
28948022309329048855892746252171976963363056481941560715954676764349967630337.
  */
 static ORDER = Fp.modulus;

 /**
  * Coerce anything "field-like" (bigint, number, string, and {@link Field}) to a Field.
  */
 constructor(x: bigint | number | string | Field | FieldVar | FieldConst) {
   if (Field.#isField(x)) {
     this.value = x.value;
     return;
   }
   if (Array.isArray(x)) {
     if (typeof x[1] === 'bigint') {
       // FieldConst
       this.value = FieldVar.constant(x as FieldConst);
       return;
     } else {
       // FieldVar
       this.value = x as FieldVar;
       return;
     }
   }
   // TODO this should handle common values efficiently by reading from a lookup table
   this.value = FieldVar.constant(Fp(x));
 }

 // helpers
 static #isField(
   x: bigint | number | string | Field | FieldVar | FieldConst
 ): x is Field {
   return x instanceof Field;
 }
 static #toConst(x: bigint | number | string | ConstantField): FieldConst {
   if (Field.#isField(x)) return x.value[1];
   return FieldConst.fromBigint(Fp(x));
 }
 static #toVar(x: bigint | number | string | Field): FieldVar {
   if (Field.#isField(x)) return x.value;
   return FieldVar.constant(Fp(x));
 }
 static from(x: bigint | number | string | Field): Field {
   if (Field.#isField(x)) return x;
   return new Field(x);
 }

 /**
  * Check whether this {@link Field} element is a hard-coded constant in the constraint system.
  * If a {@link Field} is constructed outside a zkApp method, it is a constant.
  *
  * @example
  * &nbsp;&nbsp;&nbsp;ts
  * console.log(Field(42).isConstant()); // true
  * &nbsp;&nbsp;&nbsp;
  *
  * @example
  * &nbsp;&nbsp;&nbsp;ts
  * \@method myMethod(x: Field) {
  *    console.log(x.isConstant()); // false
  * }
  * &nbsp;&nbsp;&nbsp;
  *
  * @return A &nbsp;boolean&nbsp; showing if this {@link Field} is a constant or not.
  */
 isConstant(): this is { value: ConstantFieldVar } {
   return this.value[0] === FieldType.Constant;
 }

 #toConstant(name: string): ConstantField {
   return toConstantField(this, name, 'x', 'field element');
 }

 /**
  * Create a {@link Field} element equivalent to this {@link Field} element's value,
   * but is a constant.
   * See {@link Field.isConstant} for more information about what is a constant {@link Field}.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const someField = Field(42);
   * someField.toConstant().assertEquals(someField); // Always true
   * &nbsp;&nbsp;&nbsp;
   *
   * @return A constant {@link Field} element equivalent to this {@link Field} element.
   */
  toConstant(): ConstantField {
    return this.#toConstant('toConstant');
  }

  /**
   * Serialize the {@link Field} to a bigint, e.g. for printing. Trying to print a {@link Field} without this function
will directly stringify the Field object, resulting in unreadable output.
   *
   * **Warning**: This operation does _not_ affect the circuit and can't be used to prove anything about the
bigint representation of the {@link Field}. Use the operation only during debugging.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const someField = Field(42);
   * console.log(someField.toBigInt());
   * &nbsp;&nbsp;&nbsp;
   *
   * @return A bigint equivalent to the bigint representation of the Field.
   */
  toBigInt() {
    let x = this.#toConstant('toBigInt');
    return FieldConst.toBigint(x.value[1]);
  }

  /**
   * Serialize the {@link Field} to a string, e.g. for printing. Trying to print a {@link Field} without this function
will directly stringify the Field object, resulting in unreadable output.
   *
   * **Warning**: This operation does _not_ affect the circuit and can't be used to prove anything about the
string representation of the {@link Field}. Use the operation only during debugging.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const someField = Field(42);
   * console.log(someField.toString());
   * &nbsp;&nbsp;&nbsp;
   *
   * @return A string equivalent to the string representation of the Field.
   */
  toString() {
    return this.#toConstant('toString').toBigInt().toString();
  }

  /**
   * Assert that this {@link Field} is equal another "field-like" value.
   * Calling this function is equivalent to &nbsp;Field(...).equals(...).assertEquals(Bool(true))&nbsp;.
   * See {@link Field.equals} for more details.
   *
   * **Important**: If an assertion fails, the code throws an error.
   *
   * @param value - the "field-like" value to compare & assert with this {@link Field}.
   * @param message? - a string error message to print if the assertion fails, optional.
   */
  assertEquals(y: Field | bigint | number | string, message?: string) {
    try {
      if (this.isConstant() && isConstant(y)) {
        if (this.toBigInt() !== toFp(y)) {
          throw Error(&nbsp;Field.assertEquals(): ${this} != ${y}&nbsp;);
        }
        return;
      }
      Snarky.field.assertEqual(this.value, Field.#toVar(y));
    } catch (err) {
      throw withMessage(err, message);
    }
  }

  /**
   * Add a "field-like" value to this {@link Field} element.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const x = Field(3);
   * const sum = x.add(5);
   *
   * sum.assertEquals(Field(8));
   * &nbsp;&nbsp;&nbsp;
   *
   * **Warning**: This is a modular addition in the pasta field.
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const x = Field(1);
   * const sum = x.add(Field(-7));
   *
   * // If you try to print sum - &nbsp;console.log(sum.toBigInt())&nbsp; - you will realize that it prints a very
big integer because this is modular arithmetic, and 1 + (-7) circles around the field to become p - 6.
   * // You can use the reverse operation of addition (substraction) to prove the sum is calculated correctly.
   *
   * sum.sub(x).assertEquals(Field(-7));
   * sum.sub(Field(-7)).assertEquals(x);
   * &nbsp;&nbsp;&nbsp;
   *
   * @param value - a "field-like" value to add to the {@link Field}.
```

```
   *
   * @return A {@link Field} element equivalent to the modular addition of the two value.
   */
  add(y: Field | bigint | number | string): Field {
    if (this.isConstant() && isConstant(y)) {
      return new Field(Fp.add(this.toBigInt(), toFp(y)));
    }
    // return new AST node Add(x, y)
    let z = Snarky.field.add(this.value, Field.#toVar(y));
    return new Field(z);
  }

  /**
   * Negate a {@link Field}. This is equivalent to multiplying the {@link Field} by -1.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const negOne = Field(1).neg();
   * negOne.assertEquals(-1);
   * &nbsp;&nbsp;&nbsp;
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const someField = Field(42);
   * someField.neg().assertEquals(someField.mul(Field(-1))); // This statement is always true regardless of
the value of &nbsp;someField&nbsp;
   * &nbsp;&nbsp;&nbsp;
   *
   * **Warning**: This is a modular negation. For details, see the {@link sub} method.
   *
   * @return A {@link Field} element that is equivalent to the element multiplied by -1.
   */
  neg() {
    if (this.isConstant()) {
      return new Field(Fp.negate(this.toBigInt()));
    }
    // return new AST node Scale(-1, x)
    let z = Snarky.field.scale(FieldConst[-1], this.value);
    return new Field(z);
  }

  /**
   * Substract another "field-like" value from this {@link Field} element.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const x = Field(3);
   * const difference = x.sub(5);
   *
   * difference.assertEquals(Field(-2));
   * &nbsp;&nbsp;&nbsp;
   *
   * **Warning**: This is a modular substraction in the pasta field.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const x = Field(1);
   * const difference = x.sub(Field(2));
   *
   * // If you try to print difference - &nbsp;console.log(difference.toBigInt())&nbsp; - you will realize that it
prints a very big integer because this is modular arithmetic, and 1 - 2 circles around the field to become p -
1.
   * // You can use the reverse operation of substraction (addition) to prove the difference is calculated
correctly.
   * difference.add(Field(2)).assertEquals(x);
   * &nbsp;&nbsp;&nbsp;
   *
   * @param value - a "field-like" value to substract from the {@link Field}.
   *
   * @return A {@link Field} element equivalent to the modular difference of the two value.
   */
  sub(y: Field | bigint | number | string) {
    return this.add(Field.from(y).neg());
  }

  /**
   * Checks if this {@link Field} is even. Returns &nbsp;true&nbsp; for even elements and &nbsp;false&nbsp;
for odd elements.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * let a = Field(5);
   * a.isEven(); // false
   * a.isEven().assertTrue(); // throws, as expected!
   *
   * let b = Field(4);
   * b.isEven(); // true
   * b.isEven().assertTrue(); // does not throw, as expected!
   * &nbsp;&nbsp;&nbsp;
   */
  isEven() {
    if (this.isConstant()) return new Bool(this.toBigInt() % 2n === 0n);

    let [, isOddVar, xDiv2Var] = Snarky.exists(2, () => {
      let bits = Fp.toBits(this.toBigInt());
      let isOdd = bits.shift()! ? 1n : 0n;

      return [
        0,
        FieldConst.fromBigint(isOdd),
        FieldConst.fromBigint(Fp.fromBits(bits)),
      ];
    });
    let isOdd = new Field(isOddVar);
    let xDiv2 = new Field(xDiv2Var);

    // range check for 253 bits
    // WARNING: this makes use of a special property of the Pasta curves,
    // namely that a random field element is < 2^254 with overwhelming probability
    // TODO use 88-bit RCs to make this more efficient
    xDiv2.toBits(253);

    // check composition
    xDiv2.mul(2).add(isOdd).assertEquals(this);

    return new Bool(isOddVar).not();
  }

  /**
   * Multiply another "field-like" value with this {@link Field} element.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const x = Field(3);
   * const product = x.mul(Field(5));
   *
   * product.assertEquals(Field(15));
   * &nbsp;&nbsp;&nbsp;
   *
   * @param value - a "field-like" value to multiply with the {@link Field}.
   *
   * @return A {@link Field} element equivalent to the modular difference of the two value.
   */
  mul(y: Field | bigint | number | string): Field {
    if (this.isConstant() && isConstant(y)) {
      return new Field(Fp.mul(this.toBigInt(), toFp(y)));
    }
    // if one of the factors is constant, return Scale AST node
    if (isConstant(y)) {
      let z = Snarky.field.scale(Field.#toConst(y), this.value);
      return new Field(z);
    }
    if (this.isConstant()) {
      let z = Snarky.field.scale(this.value[1], y.value);
      return new Field(z);
    }
    // create a new witness for z = x*y
    let z = Snarky.existsVar(() =>
      FieldConst.fromBigint(Fp.mul(this.toBigInt(), toFp(y)))
    );
    // add a multiplication constraint
    Snarky.field.assertMul(this.value, y.value, z);
    return new Field(z);
  }
```

```
  /**
   * [Modular inverse](https://en.wikipedia.org/wiki/Modular_multiplicative_inverse) of this {@link Field}
element.
   * Equivalent to 1 divided by this {@link Field}, in the sense of modular arithmetic.
   *
   * Proves that this Field is non-zero, or throws a "Division by zero" error.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const someField = Field(42);
   * const inverse = someField.inv();
   * inverse.assertEquals(Field(1).div(example)); // This statement is always true regardless of the value of
&nbsp;someField&nbsp;
   * &nbsp;&nbsp;&nbsp;
   *
   * **Warning**: This is a modular inverse. See {@link div} method for more details.
   *
   * @return A {@link Field} element that is equivalent to one divided by this element.
   */
  inv() {
    if (this.isConstant()) {
      let z = Fp.inverse(this.toBigInt());
      if (z === undefined) throw Error('Field.inv(): Division by zero');
      return new Field(z);
    }
    // create a witness for z = x^(-1)
    let z = Snarky.existsVar(() => {
      let z = Fp.inverse(this.toBigInt()) ?? 0n;
      return FieldConst.fromBigint(z);
    });
    // constrain x * z === 1
    Snarky.field.assertMul(this.value, z, FieldVar[1]);
    return new Field(z);
  }

  /**
   * Divide another "field-like" value through this {@link Field}.
   *
   * Proves that the denominator is non-zero, or throws a "Division by zero" error.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const x = Field(6);
   * const quotient = x.div(Field(3));
   *
   * quotient.assertEquals(Field(2));
   * &nbsp;&nbsp;&nbsp;
   *
   * **Warning**: This is a modular division in the pasta field. You can think this as the reverse operation of
modular multiplication.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const x = Field(2);
   * const y = Field(5);
   *
   * const quotient = x.div(y);
   *
   * // If you try to print quotient - &nbsp;console.log(quotient.toBigInt())&nbsp; - you will realize that it prints a
very big integer because this is a modular inverse.
   * // You can use the reverse operation of division (multiplication) to prove the quotient is calculated
correctly.
   *
   * quotient.mul(y).assertEquals(x);
   * &nbsp;&nbsp;&nbsp;
   *
   * @param value - a "field-like" value to divide with the {@link Field}.
   *
   * @return A {@link Field} element equivalent to the modular division of the two value.
   */
  div(y: Field | bigint | number | string) {
    // this intentionally uses 2 constraints instead of 1 to avoid an unconstrained output when dividing 0/0
    // (in this version, division by 0 is strictly not allowed)
    return this.mul(Field.from(y).inv());
  }

  /**
   * Square this {@link Field} element.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const someField = Field(7);
   * const square = someField.square();
   *
   * square.assertEquals(someField.mul(someField)); // This statement is always true regardless of the value
of &nbsp;someField&nbsp;
   * &nbsp;&nbsp;&nbsp;
   *
   * ** Warning: This is a modular multiplication. See &nbsp;mul()&nbsp; method for more details.
   *
   * @return A {@link Field} element equivalent to the multiplication of the {@link Field} element with itself.
   */
  square() {
    if (this.isConstant()) {
      return new Field(Fp.square(this.toBigInt()));
    }
    // create a new witness for z = x^2
    let z = Snarky.existsVar(() =>
      FieldConst.fromBigint(Fp.square(this.toBigInt()))
    );
    // add a squaring constraint
    Snarky.field.assertSquare(this.value, z);
    return new Field(z);
  }
```

```
/**
 * Take the square root of this {@link Field} element.
 *
 * Proves that the Field element has a square root in the finite field, or throws if it doesn't.
 *
 * @example
 * &nbsp;&nbsp;&nbsp;ts
 * let z = x.sqrt();
 * z.mul(z).assertEquals(x); // true for every &nbsp;x&nbsp;
 * &nbsp;&nbsp;&nbsp;
 *
 * **Warning**: This is a modular square root, which is any number z that satisfies z*z = x (mod p).
 * Note that, if a square root z exists, there also exists a second one, -z (which is different if z != 0).
 * Therefore, this method leaves an adversarial prover the choice between two different values to return.
 *
 * @return A {@link Field} element equivalent to the square root of the {@link Field} element.
 */
sqrt() {
  if (this.isConstant()) {
    let z = Fp.sqrt(this.toBigInt());
    if (z === undefined)
      throw Error(
        &nbsp;Field.sqrt(): input ${this} has no square root in the field.&nbsp;
      );
    return new Field(z);
  }
  // create a witness for sqrt(x)
  let z = Snarky.existsVar(() => {
    let z = Fp.sqrt(this.toBigInt()) ?? 0n;
    return FieldConst.fromBigint(z);
  });
  // constrain z * z === x
  Snarky.field.assertSquare(z, this.value);
  return new Field(z);
}

/**
 * @deprecated use &nbsp;x.equals(0)&nbsp; which is equivalent
 */
isZero() {
  if (this.isConstant()) {
    return new Bool(this.toBigInt() === 0n);
  }
  // create witnesses z = 1/x, or z=0 if x=0,
  // and b = 1 - zx
  let [, b, z] = Snarky.exists(2, () => {
    let x = this.toBigInt();
    let z = Fp.inverse(x) ?? 0n;
    let b = Fp.sub(1n, Fp.mul(z, x));
    return [0, FieldConst.fromBigint(b), FieldConst.fromBigint(z)];
  });
  // add constraints
  // b * x === 0
  Snarky.field.assertMul(b, this.value, FieldVar[0]);
  // z * x === 1 - b
  Snarky.field.assertMul(
    z,
    this.value,
    Snarky.field.add(FieldVar[1], Snarky.field.scale(FieldConst[-1], b))
  );
  // ^^^ these prove that b = Bool(x === 0):
  // if x = 0, the 2nd equation implies b = 1
  // if x != 0, the 1st implies b = 0
  return new Bool(b);
}

/**
 * Check if this {@link Field} is equal another "field-like" value.
 * Returns a {@link Bool}, which is a provable type and can be used to prove the validity of this statement.
 *
 * @example
 * &nbsp;&nbsp;&nbsp;ts
 * Field(5).equals(5).assertEquals(Bool(true));
 * &nbsp;&nbsp;&nbsp;
 *
 * @param value - the "field-like" value to compare with this {@link Field}.
 *
 * @return A {@link Bool} representing if this {@link Field} is equal another "field-like" value.
 */
equals(y: Field | bigint | number | string): Bool {
  // x == y is equivalent to x - y == 0
  // if one of the two is constant, we just need the two constraints in &nbsp;isZero&nbsp;
  if (this.isConstant() || isConstant(y)) {
    return this.sub(y).isZero();
  }
  // if both are variables, we create one new variable for x-y so that &nbsp;isZero&nbsp; doesn't create two
  let xMinusY = Snarky.existsVar(() =>
    FieldConst.fromBigint(Fp.sub(this.toBigInt(), toFp(y)))
  );
  Snarky.field.assertEqual(this.sub(y).value, xMinusY);
  return new Field(xMinusY).isZero();
}

// internal base method for all comparisons
#compare(y: FieldVar) {
  // TODO: support all bit lengths
  let maxLength = Fp.sizeInBits - 2;
  asProver(() => {
    let actualLength = Math.max(
      this.toBigInt().toString(2).length,
      new Field(y).toBigInt().toString(2).length
    );
    if (actualLength > maxLength)
```

```
      throw Error(
        &nbsp;Provable comparison functions can only be used on Fields of size <= ${maxLength} bits, got
${actualLength} bits.&nbsp;
      );
    });
    let [, less, lessOrEqual] = Snarky.field.compare(maxLength, this.value, y);
    return { less: new Bool(less), lessOrEqual: new Bool(lessOrEqual) };
  }

  /**
   * Check if this {@link Field} is less than another "field-like" value.
   * Returns a {@link Bool}, which is a provable type and can be used prove to the validity of this statement.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * Field(2).lessThan(3).assertEquals(Bool(true));
   * &nbsp;&nbsp;&nbsp;
   *
   * **Warning**: Comparison methods only support Field elements of size <= 253 bits in provable code.
   * The method will throw if one of the inputs exceeds 253 bits.
   *
   * **Warning**: As this method compares the bigint value of a {@link Field}, it can result in unexpected
behavior when used with negative inputs or modular division.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * Field(1).div(Field(3)).lessThan(Field(1).div(Field(2))).assertEquals(Bool(true)); // This code will throw an
error
   * &nbsp;&nbsp;&nbsp;
   *
   * @param value - the "field-like" value to compare with this {@link Field}.
   *
   * @return A {@link Bool} representing if this {@link Field} is less than another "field-like" value.
   */
  lessThan(y: Field | bigint | number | string): Bool {
    if (this.isConstant() && isConstant(y)) {
      return new Bool(this.toBigInt() < toFp(y));
    }
    return this.#compare(Field.#toVar(y)).less;
  }

  /**
   * Check if this {@link Field} is less than or equal to another "field-like" value.
   * Returns a {@link Bool}, which is a provable type and can be used to prove the validity of this statement.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * Field(3).lessThanOrEqual(3).assertEquals(Bool(true));
   * &nbsp;&nbsp;&nbsp;
   *
   * **Warning**: Comparison methods only support Field elements of size <= 253 bits in provable code.
   * The method will throw if one of the inputs exceeds 253 bits.
   *
   * **Warning**: As this method compares the bigint value of a {@link Field}, it can result in unexpected
behaviour when used with negative inputs or modular division.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * Field(1).div(Field(3)).lessThanOrEqual(Field(1).div(Field(2))).assertEquals(Bool(true)); // This code will
throw an error
   * &nbsp;&nbsp;&nbsp;
   *
   * @param value - the "field-like" value to compare with this {@link Field}.
   *
   * @return A {@link Bool} representing if this {@link Field} is less than or equal another "field-like" value.
   */
  lessThanOrEqual(y: Field | bigint | number | string): Bool {
    if (this.isConstant() && isConstant(y)) {
      return new Bool(this.toBigInt() <= toFp(y));
    }
    return this.#compare(Field.#toVar(y)).lessOrEqual;
  }

  /**
   * Check if this {@link Field} is greater than another "field-like" value.
   * Returns a {@link Bool}, which is a provable type and can be used to prove the validity of this statement.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * Field(5).greaterThan(3).assertEquals(Bool(true));
   * &nbsp;&nbsp;&nbsp;
   *
   * **Warning**: Comparison methods currently only support Field elements of size <= 253 bits in provable
code.
   * The method will throw if one of the inputs exceeds 253 bits.
   *
   * **Warning**: As this method compares the bigint value of a {@link Field}, it can result in unexpected
behaviour when used with negative inputs or modular division.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * Field(1).div(Field(2)).greaterThan(Field(1).div(Field(3))).assertEquals(Bool(true)); // This code will throw
an error
   * &nbsp;&nbsp;&nbsp;
   *
   * @param value - the "field-like" value to compare with this {@link Field}.
   *
   * @return A {@link Bool} representing if this {@link Field} is greater than another "field-like" value.
   */
  greaterThan(y: Field | bigint | number | string) {
    return Field.from(y).lessThan(this);
  }

  /**
```

```
   * Check if this {@link Field} is greater than or equal another "field-like" value.
   * Returns a {@link Bool}, which is a provable type and can be used to prove the validity of this statement.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * Field(3).greaterThanOrEqual(3).assertEquals(Bool(true));
   * &nbsp;&nbsp;&nbsp;
   *
   * **Warning**: Comparison methods only support Field elements of size <= 253 bits in provable code.
   * The method will throw if one of the inputs exceeds 253 bits.
   *
   * **Warning**: As this method compares the bigint value of a {@link Field}, it can result in unexpected
behaviour when used with negative inputs or modular division.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * Field(1).div(Field(2)).greaterThanOrEqual(Field(1).div(Field(3))).assertEquals(Bool(true)); // This code
will throw an error
   * &nbsp;&nbsp;&nbsp;
   *
   * @param value - the "field-like" value to compare with this {@link Field}.
   *
   * @return A {@link Bool} representing if this {@link Field} is greater than or equal another "field-like" value.
   */
  greaterThanOrEqual(y: Field | bigint | number | string) {
    return Field.from(y).lessThanOrEqual(this);
  }

  /**
   * Assert that this {@link Field} is less than another "field-like" value.
   * Calling this function is equivalent to &nbsp;Field(...).lessThan(...).assertEquals(Bool(true))&nbsp;.
   * See {@link Field.lessThan} for more details.
   *
   * **Important**: If an assertion fails, the code throws an error.
   *
   * **Warning**: Comparison methods only support Field elements of size <= 253 bits in provable code.
   * The method will throw if one of the inputs exceeds 253 bits.
   *
   * @param value - the "field-like" value to compare & assert with this {@link Field}.
   * @param message? - a string error message to print if the assertion fails, optional.
   */
  assertLessThan(y: Field | bigint | number | string, message?: string) {
    try {
      if (this.isConstant() && isConstant(y)) {
        if (!(this.toBigInt() < toFp(y))) {
          throw Error(&nbsp;Field.assertLessThan(): expected ${this} < ${y}&nbsp;);
        }
        return;
      }
      let { less } = this.#compare(Field.#toVar(y));
      less.assertTrue();
    } catch (err) {
```

```
    throw withMessage(err, message);
  }
}

/**
 * Assert that this {@link Field} is less than or equal to another "field-like" value.
 * Calling this function is equivalent to
&nbsp;Field(...).lessThanOrEqual(...).assertEquals(Bool(true))&nbsp;.
 * See {@link Field.lessThanOrEqual} for more details.
 *
 * **Important**: If an assertion fails, the code throws an error.
 *
 * **Warning**: Comparison methods only support Field elements of size <= 253 bits in provable code.
 * The method will throw if one of the inputs exceeds 253 bits.
 *
 * @param value - the "field-like" value to compare & assert with this {@link Field}.
 * @param message? - a string error message to print if the assertion fails, optional.
 */
assertLessThanOrEqual(y: Field | bigint | number | string, message?: string) {
  try {
    if (this.isConstant() && isConstant(y)) {
      if (!(this.toBigInt() <= toFp(y))) {
        throw Error(&nbsp;Field.assertLessThan(): expected ${this} <= ${y}&nbsp;);
      }
      return;
    }
    let { lessOrEqual } = this.#compare(Field.#toVar(y));
    lessOrEqual.assertTrue();
  } catch (err) {
    throw withMessage(err, message);
  }
}

/**
 * Assert that this {@link Field} is greater than another "field-like" value.
 * Calling this function is equivalent to &nbsp;Field(...).greaterThan(...).assertEquals(Bool(true))&nbsp;.
 * See {@link Field.greaterThan} for more details.
 *
 * **Important**: If an assertion fails, the code throws an error.
 *
 * **Warning**: Comparison methods only support Field elements of size <= 253 bits in provable code.
 * The method will throw if one of the inputs exceeds 253 bits.
 *
 * @param value - the "field-like" value to compare & assert with this {@link Field}.
 * @param message? - a string error message to print if the assertion fails, optional.
 */
assertGreaterThan(y: Field | bigint | number | string, message?: string) {
  Field.from(y).assertLessThan(this, message);
}

/**
 * Assert that this {@link Field} is greater than or equal to another "field-like" value.
```

```
  * Calling this function is equivalent to
&nbsp;Field(...).greaterThanOrEqual(...).assertEquals(Bool(true))&nbsp;.
  * See {@link Field.greaterThanOrEqual} for more details.
  *
  * **Important**: If an assertion fails, the code throws an error.
  *
  * **Warning**: Comparison methods only support Field elements of size <= 253 bits in provable code.
  * The method will throw if one of the inputs exceeds 253 bits.
  *
  * @param value - the "field-like" value to compare & assert with this {@link Field}.
  * @param message? - a string error message to print if the assertion fails, optional.
  */
 assertGreaterThanOrEqual(
   y: Field | bigint | number | string,
   message?: string
 ) {
   Field.from(y).assertLessThanOrEqual(this, message);
 }

 /**
  * Assert that this {@link Field} does not equal another field-like value.
  *
  * Note: This uses fewer constraints than &nbsp;x.equals(y).assertFalse()&nbsp;.
  *
  * @example
  * &nbsp;&nbsp;&nbsp;ts
  * x.assertNotEquals(0, "expect x to be non-zero");
  * &nbsp;&nbsp;&nbsp;
  */
 assertNotEquals(y: Field | bigint | number | string, message?: string) {
   try {
     if (this.isConstant() && isConstant(y)) {
       if (this.toBigInt() === toFp(y)) {
         throw Error(&nbsp;Field.assertNotEquals(): ${this} = ${y}&nbsp;);
       }
       return;
     }
     // inv() proves that a field element is non-zero, using 1 constraint.
     // so this takes 1-2 generic gates, while x.equals(y).assertTrue() takes 3-5
     this.sub(y).inv();
   } catch (err) {
     throw withMessage(err, message);
   }
 }

 /**
  * Assert that this {@link Field} is equal to 1 or 0 as a "field-like" value.
  * Calling this function is equivalent to &nbsp;Bool.or(Field(...).equals(1),
Field(...).equals(0)).assertEquals(Bool(true))&nbsp;.
  *
  * **Important**: If an assertion fails, the code throws an error.
  *
  * @param value - the "field-like" value to compare & assert with this {@link Field}.
  * @param message? - a string error message to print if the assertion fails, optional.
  */
 assertBool(message?: string) {
   try {
     if (this.isConstant()) {
       let x = this.toBigInt();
       if (x !== 0n && x !== 1n) {
         throw Error(&nbsp;Field.assertBool(): expected ${x} to be 0 or 1&nbsp;);
       }
       return;
     }
     Snarky.field.assertBoolean(this.value);
   } catch (err) {
     throw withMessage(err, message);
   }
 }

 static #checkBitLength(name: string, length: number) {
   if (length > Fp.sizeInBits)
     throw Error(
       &nbsp;${name}: bit length must be ${Fp.sizeInBits} or less, got ${length}&nbsp;
     );
   if (length <= 0)
     throw Error(&nbsp;${name}: bit length must be positive, got ${length}&nbsp;);
 }

 /**
  * Returns an array of {@link Bool} elements representing [little endian binary representation]
(https://en.wikipedia.org/wiki/Endianness) of this {@link Field} element.
  *
  * If you use the optional &nbsp;length&nbsp; argument, proves that the field element fits in
&nbsp;length&nbsp; bits.
  * The &nbsp;length&nbsp; has to be between 0 and 255 and the method throws if it isn't.
  *
  * **Warning**: The cost of this operation in a zk proof depends on the &nbsp;length&nbsp; you specify,
  * which by default is 255 bits. Prefer to pass a smaller &nbsp;length&nbsp; if possible.
  *
  * @param length - the number of bits to fit the element. If the element does not fit in &nbsp;length&nbsp;
bits, the functions throws an error.
  *
  * @return An array of {@link Bool} element representing little endian binary representation of this {@link
Field}.
  */
 toBits(length?: number) {
   if (length !== undefined) Field.#checkBitLength('Field.toBits()', length);
   if (this.isConstant()) {
     let bits = Fp.toBits(this.toBigInt());
     if (length !== undefined) {
       if (bits.slice(length).some((bit) => bit))
         throw Error(&nbsp;Field.toBits(): ${this} does not fit in ${length} bits&nbsp;);
       return bits.slice(0, length).map((b) => new Bool(b));
```

```
    }
    return bits.map((b) => new Bool(b));
  }
  let [, ...bits] = Snarky.field.toBits(length ?? Fp.sizeInBits, this.value);
  return bits.map((b) => new Bool(b));
}

/**
 * Convert a bit array into a {@link Field} element using [little endian binary representation]
(https://en.wikipedia.org/wiki/Endianness)
 *
 * The method throws if the given bits do not fit in a single Field element. A Field element can be at most
255 bits.
 *
 * **Important**: If the given &nbsp;bytes&nbsp; array is an array of &nbsp;booleans&nbsp; or {@link Bool}
elements that all are &nbsp;constant&nbsp;, the resulting {@link Field} element will be a constant as well. Or
else, if the given array is a mixture of constants and variables of {@link Bool} type, the resulting {@link Field}
will be a variable as well.
 *
 * @param bytes - An array of {@link Bool} or &nbsp;boolean&nbsp; type.
 *
 * @return A {@link Field} element matching the [little endian binary representation]
(https://en.wikipedia.org/wiki/Endianness) of the given &nbsp;bytes&nbsp; array.
 */
static fromBits(bits: (Bool | boolean)[]) {
  let length = bits.length;
  Field.#checkBitLength('Field.fromBits()', length);
  if (bits.every((b) => typeof b === 'boolean' || b.toField().isConstant())) {
    let bits_ = bits
      .map((b) => (typeof b === 'boolean' ? b : b.toBoolean()))
      .concat(Array(Fp.sizeInBits - length).fill(false));
    return new Field(Fp.fromBits(bits_));
  }
  let bitsVars = bits.map((b): FieldVar => {
    if (typeof b === 'boolean') return b ? FieldVar[1] : FieldVar[0];
    return b.toField().value;
  });
  let x = Snarky.field.fromBits([0, ...bitsVars]);
  return new Field(x);
}

/**
 * Create a new {@link Field} element from the first &nbsp;length&nbsp; bits of this {@link Field} element.
 *
 * The &nbsp;length&nbsp; has to be a multiple of 16, and has to be between 0 and 255, otherwise the
method throws.
 *
 * As {@link Field} elements are represented using [little endian binary representation]
(https://en.wikipedia.org/wiki/Endianness),
 * the resulting {@link Field} element will equal the original one if it fits in &nbsp;length&nbsp; bits.
 *
 * @param length - The number of bits to take from this {@link Field} element.
```

```
   *
   * @return A {@link Field} element that is equal to the &nbsp;length&nbsp; of this {@link Field} element.
   */
  rangeCheckHelper(length: number) {
    Field.#checkBitLength('Field.rangeCheckHelper()', length);
    if (length % 16 !== 0)
      throw Error(
        'Field.rangeCheckHelper(): &nbsp;length&nbsp; has to be a multiple of 16.'
      );
    let lengthDiv16 = length / 16;
    if (this.isConstant()) {
      let bits = Fp.toBits(this.toBigInt())
        .slice(0, length)
        .concat(Array(Fp.sizeInBits - length).fill(false));
      return new Field(Fp.fromBits(bits));
    }
    let x = Snarky.field.truncateToBits16(lengthDiv16, this.value);
    return new Field(x);
  }

  /**
   * **Warning**: This function is mainly for internal use. Normally it is not intended to be used by a zkApp
developer.
   *
   * In o1js, addition and scaling (multiplication of variables by a constant) of variables is represented as an
AST - [abstract syntax tree](https://en.wikipedia.org/wiki/Abstract_syntax_tree). For example, the expression
&nbsp;x.add(y).mul(2)&nbsp; is represented as &nbsp;Scale(2, Add(x, y))&nbsp;.
   *
   *  A new internal variable is created only when the variable is needed in a multiplicative or any higher level
constraint (for example multiplication of two {@link Field} elements) to represent the operation.
   *
   * The &nbsp;seal()&nbsp; function tells o1js to stop building an AST and create a new variable right away.
   *
   * @return A {@link Field} element that is equal to the result of AST that was previously on this {@link
Field} element.
   */
  seal() {
    if (this.isConstant()) return this;
    let x = Snarky.field.seal(this.value);
    return new Field(x);
  }

  /**
   * A random {@link Field} element.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * console.log(Field.random().toBigInt()); // Run this code twice!
   * &nbsp;&nbsp;&nbsp;
   *
   * @return A random {@link Field} element.
   */
```

```
static random() {
  return new Field(Fp.random());
}

// internal stuff

// Provable<Field>

/**
 * This function is the implementation of {@link Provable.toFields} for the {@link Field} type.
 *
 * Static function to serializes a {@link Field} into an array of {@link Field} elements.
 * This will be always an array of length 1, where the first and only element equals the given parameter
itself.
 *
 * @param value - the {@link Field} element to cast the array from.
 *
 * @return A {@link Field} array of length 1 created from this {@link Field}.
 */
static toFields(x: Field) {
  return [x];
}

/**
 * This function is the implementation of {@link Provable.toAuxiliary} for the {@link Field} type.
 *
 * As the primitive {@link Field} type has no auxiliary data associated with it, this function will always return
an empty array.
 *
 * @param value - The {@link Field} element to get the auxiliary data of, optional. If not provided, the
function returns an empty array.
 */
static toAuxiliary(): [] {
  return [];
}

/**
 * This function is the implementation of {@link Provable.sizeInFields} for the {@link Field} type.
 *
 * Size of the {@link Field} type is 1, as it is the primitive type.
 * This function returns a regular number, so you cannot use it to prove something on chain. You can use it
during debugging or to understand the memory complexity of some type.
 *
 * @example
 * &nbsp;&nbsp;&nbsp;ts
 * console.log(Field.sizeInFields()); // Prints 1
 * &nbsp;&nbsp;&nbsp;
 *
 * @return A number representing the size of the {@link Field} type in terms of {@link Field} type itself.
 */
static sizeInFields() {
  return 1;
}

/**
 * Implementation of {@link Provable.fromFields} for the {@link Field} type.
 *
 * **Warning**: This function is designed for internal use. It is not intended to be used by a zkApp
developer.
 *
 * Creates a {@link Field} from an array of Fields of length 1.
 *
 * @param fields - an array of length 1 serialized from {@link Field} elements.
 *
 * @return The first {@link Field} element of the given array.
 */
static fromFields([x]: Field[]) {
  return x;
}

/**
 * This function is the implementation of {@link Provable.check} in {@link Field} type.
 *
 * As any field element can be a {@link Field}, this function does not create any assertions, so it does
nothing.
 *
 * @param value - the {@link Field} element to check.
 */
static check() {}

/**
 * This function is the implementation of {@link Provable.toFields} for the {@link Field} type.
 *
 * The result will be always an array of length 1, where the first and only element equals the {@link Field}
itself.
 *
 * @return A {@link Field} array of length 1 created from this {@link Field}.
 */
toFields() {
  return Field.toFields(this);
}

/**
 * This function is the implementation of {@link Provable.toAuxiliary} for the {@link Field} type.
 *
 * As the primitive {@link Field} type has no auxiliary data associated with it, this function will always return
an empty array.
 */
toAuxiliary() {
  return Field.toAuxiliary();
}

// ProvableExtended<Field>
/**
 * Serialize the {@link Field} to a JSON string, e.g. for printing. Trying to print a {@link Field} without this
function will directly stringify the Field object, resulting in unreadable output.
 *
 * **Warning**: This operation does _not_ affect the circuit and can't be used to prove anything about the
JSON string representation of the {@link Field}. Use the operation only during debugging.
 *
 * @example
 * &nbsp;&nbsp;&nbsp;ts
 * const someField = Field(42);
 * console.log(someField.toJSON());
 * &nbsp;&nbsp;&nbsp;
 *
 * @return A string equivalent to the JSON representation of the {@link Field}.
 */
toJSON() {
  return this.#toConstant('toJSON').toString();
}

/**
 * Serialize the given {@link Field} element to a JSON string, e.g. for printing. Trying to print a {@link Field}
without this function will directly stringify the Field object, resulting in unreadable output.
 *
 * **Warning**: This operation does _not_ affect the circuit and can't be used to prove anything about the
JSON string representation of the {@link Field}. Use the operation only during debugging.
 *
 * @example
 * &nbsp;&nbsp;&nbsp;ts
 * const someField = Field(42);
 * console.log(Field.toJSON(someField));
 * &nbsp;&nbsp;&nbsp;
 *
 * @param value - The JSON string to coerce the {@link Field} from.
 *
 * @return A string equivalent to the JSON representation of the given {@link Field}.
 */
static toJSON(x: Field) {
  return x.toJSON();
}

/**
 * Deserialize a JSON string containing a "field-like" value into a {@link Field} element.
 *
 * **Warning**: This operation does _not_ affect the circuit and can't be used to prove anything about the
string representation of the {@link Field}.
 *
 * @param value - the "field-like" value to coerce the {@link Field} from.
 *
 * @return A {@link Field} coerced from the given JSON string.
 */
static fromJSON(json: string) {
  return new Field(Fp.fromJSON(json));
```

```
  }

  /**
   * **Warning**: This function is mainly for internal use. Normally it is not intended to be used by a zkApp
developer.
   *
   * This function is the implementation of &nbsp;ProvableExtended.toInput()&nbsp; for the {@link Field}
type.
   *
   * @param value - The {@link Field} element to get the &nbsp;input&nbsp; array.
   *
   * @return An object where the &nbsp;fields&nbsp; key is a {@link Field} array of length 1 created from this
{@link Field}.
   *
   */
  static toInput(x: Field) {
    return { fields: [x] };
  }

  // Binable<Field>

  /**
   * Create an array of digits equal to the [little-endian](https://en.wikipedia.org/wiki/Endianness) byte order of
the given {@link Field} element.
   * Note that the array has always 32 elements as the {@link Field} is a &nbsp;finite-field&nbsp; in the order
of {@link Field.ORDER}.
   *
   * @param value - The {@link Field} element to generate the array of bytes of.
   *
   * @return An array of digits equal to the [little-endian](https://en.wikipedia.org/wiki/Endianness) byte order
of the given {@link Field} element.
   *
   */
  static toBytes(x: Field) {
    return FieldBinable.toBytes(x);
  }

  /**
   * Part of the &nbsp;Binable&nbsp; interface.
   *
   * **Warning**: This function is for internal use. It is not intended to be used by a zkApp developer.
   */
  static readBytes<N extends number>(
    bytes: number[],
    offset: NonNegativeInteger<N>
  ) {
    return FieldBinable.readBytes(bytes, offset);
  }

  /**
   * Coerce a new {@link Field} element using the [little-endian](https://en.wikipedia.org/wiki/Endianness)
representation of the given &nbsp;bytes&nbsp; array.
```

```
   * Note that the given &nbsp;bytes&nbsp; array may have at most 32 elements as the {@link Field} is a
&nbsp;finite-field&nbsp; in the order of {@link Field.ORDER}.
   *
   * **Warning**: This operation does _not_ affect the circuit and can't be used to prove anything about the
byte representation of the {@link Field}.
   *
   * @param bytes - The bytes array to coerce the {@link Field} from.
   *
   * @return A new {@link Field} element created using the [little-endian]
(https://en.wikipedia.org/wiki/Endianness) representation of the given &nbsp;bytes&nbsp; array.
   */
  static fromBytes(bytes: number[]) {
    return FieldBinable.fromBytes(bytes);
  }

  /**
   * **Warning**: This function is mainly for internal use. Normally it is not intended to be used by a zkApp
developer.
   *
   * As all {@link Field} elements have 32 bytes, this function returns 32.
   *
   * @return The size of a {@link Field} element - 32.
   */
  static sizeInBytes() {
    return Fp.sizeInBytes();
  }

  /**
   * **Warning**: This function is mainly for internal use. Normally it is not intended to be used by a zkApp
developer.
   *
   * As all {@link Field} elements have 255 bits, this function returns 255.
   *
   * @return The size of a {@link Field} element in bits - 255.
   */
  static sizeInBits() {
    return Fp.sizeInBits;
  }
}

const FieldBinable = defineBinable({
  toBytes(t: Field) {
    let t0 = toConstantField(t, 'toBytes').toBigInt();
    return Fp.toBytes(t0);
  },
  readBytes(bytes, offset) {
    let uint8array = new Uint8Array(32);
    uint8array.set(bytes.slice(offset, offset + 32));
    let x = Fp.fromBytes([...uint8array]);
    return [new Field(x), offset + 32];
  },
});
```

```
function isField(x: unknown): x is Field {
  return x instanceof Field;
}

function isConstant(
  x: bigint | number | string | Field
): x is bigint | number | string | ConstantField {
  let type = typeof x;
  if (type === 'bigint' || type === 'number' || type === 'string') {
    return true;
  }
  return (x as Field).isConstant();
}

function toFp(x: bigint | number | string | Field): Fp {
  let type = typeof x;
  if (type === 'bigint' || type === 'number' || type === 'string') {
    return Fp(x as bigint | number | string);
  }
  return (x as Field).toBigInt();
}

function withMessage(error: unknown, message?: string) {
  if (message === undefined || !(error instanceof Error)) return error;
  error.message = &nbsp;${message}\n${error.message}&nbsp;;
  return error;
}

function toConstantField(
  x: Field,
  methodName: string,
  varName = 'x',
  varDescription = 'field element'
): ConstantField {
  // if this is a constant, return it
  if (x.isConstant()) return x;

  // a non-constant can only appear inside a checked computation. everything else is a bug.
  assert(
    inCheckedComputation(),
    'variables only exist inside checked computations'
  );

  // if we are inside an asProver or witness block, read the variable's value and return it as constant
  if (Snarky.run.inProverBlock()) {
    let value = Snarky.field.readVar(x.value);
    return new Field(value) as ConstantField;
  }

  // otherwise, calling &nbsp;toConstant()&nbsp; is likely a mistake. throw a helpful error message.
  throw Error(readVarMessage(methodName, varName, varDescription));
}

function readVarMessage(
  methodName: string,
  varName: string,
  varDescription: string
) {
  return &nbsp;${varName}.${methodName}() was called on a variable ${varDescription}
\&nbsp;${varName}\&nbsp; in provable code.
This is not supported, because variables represent an abstract computation,
which only carries actual values during proving, but not during compiling.

Also, reading out JS values means that whatever you're doing with those values will no longer be
linked to the original variable in the proof, which makes this pattern prone to security holes.

You can check whether your ${varDescription} is a variable or a constant by using ${varName}.isConstant().

To inspect values for debugging, use Provable.log(${varName}). For more advanced use cases,
there is \&nbsp;Provable.asProver(() => { ... })\&nbsp; which allows you to use ${varName}.${methodName}
() inside the callback.
Warning: whatever happens inside asProver() will not be part of the zk proof.
&nbsp;;
}
```

</file>

<file>

## path: /src/lib/field.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/field.unit-test.ts

```
import { ProvablePure } from '../snarky.js';
import { Field } from './core.js';
import { Field as Fp } from '../provable/field-bigint.js';
import { test, Random } from './testing/property.js';
import { deepEqual, throws } from 'node:assert/strict';
import { Provable } from './provable.js';
import { Binable } from '../bindings/lib/binable.js';
import { ProvableExtended } from './circuit_value.js';
import { FieldType } from './field.js';
import {
  equivalentProvable as equivalent,
  oneOf,
  field,
  bigintField,
  throwError,
  unit,
  bool,
  Spec,
} from './testing/equivalent.js';
```

```
// types
Field satisfies Provable<Field>;
Field satisfies ProvablePure<Field>;
Field satisfies ProvableExtended<Field>;
Field satisfies Binable<Field>;

// constructor
test(Random.field, Random.json.field, (x, y, assert) => {
  let z = Field(x);
  assert(z instanceof Field);
  assert(z.toBigInt() === x);
  assert(z.toString() === x.toString());
  assert(z.isConstant());
  deepEqual(z.toConstant(), z);

  assert((z = new Field(x)) instanceof Field && z.toBigInt() === x);
  assert((z = Field(z)) instanceof Field && z.toBigInt() === x);
  assert((z = Field(z.value)) instanceof Field && z.toBigInt() === x);

  z = Field(y);
  assert(z instanceof Field);
  assert(z.toString() === y);
  deepEqual(Field.fromJSON(y), z);
  assert(z.toJSON() === y);
});

// handles small numbers
test(Random.nat(1000), (n, assert) => {
  assert(Field(n).toString() === String(n));
});
// handles large numbers 2^31 <= x < 2^53
test(Random.int(2 ** 31, Number.MAX_SAFE_INTEGER), (n, assert) => {
  assert(Field(n).toString() === String(n));
});
// handles negative numbers
test(Random.uint32, (n) => {
  deepEqual(Field(-n), Field(n).neg());
});
// throws on fractional numbers
test.negative(Random.int(-10, 10), Random.fraction(1), (x, f) => {
  Field(x + f);
});
// correctly overflows the field
test(Random.field, Random.int(-5, 5), (x, k) => {
  deepEqual(Field(x + BigInt(k) * Field.ORDER), Field(x));
});

// Field | bigint parameter
let fieldOrBigint = oneOf(field, bigintField);

// special generator
```

```
let SmallField = Random.reject(
  Random.field,
  (x) => x.toString(2).length > Fp.sizeInBits - 2
);
let smallField: Spec<bigint, Field> = { ...field, rng: SmallField };
let smallBigint: Spec<bigint, bigint> = { ...bigintField, rng: SmallField };
let smallFieldOrBigint = oneOf(smallField, smallBigint);

// arithmetic, both in- and outside provable code
let equivalent1 = equivalent({ from: [field], to: field });
let equivalent2 = equivalent({ from: [field, fieldOrBigint], to: field });

equivalent2(Fp.add, (x, y) => x.add(y));
equivalent1(Fp.negate, (x) => x.neg());
equivalent2(Fp.sub, (x, y) => x.sub(y));
equivalent2(Fp.mul, (x, y) => x.mul(y));
equivalent1(
  (x) => Fp.inverse(x) ?? throwError('division by 0'),
  (x) => x.inv()
);
equivalent2(
  (x, y) => Fp.div(x, y) ?? throwError('division by 0'),
  (x, y) => x.div(y)
);
equivalent1(Fp.square, (x) => x.square());
equivalent1(
  (x) => Fp.sqrt(x) ?? throwError('no sqrt'),
  (x) => x.sqrt()
);
equivalent({ from: [field, fieldOrBigint], to: bool })(
  (x, y) => x === y,
  (x, y) => x.equals(y)
);

equivalent({ from: [smallField, smallFieldOrBigint], to: bool })(
  (x, y) => x < y,
  (x, y) => x.lessThan(y)
);
equivalent({ from: [smallField, smallFieldOrBigint], to: bool })(
  (x, y) => x <= y,
  (x, y) => x.lessThanOrEqual(y)
);
equivalent({ from: [field, fieldOrBigint], to: unit })(
  (x, y) => x === y || throwError('not equal'),
  (x, y) => x.assertEquals(y)
);
equivalent({ from: [field, fieldOrBigint], to: unit })(
  (x, y) => x !== y || throwError('equal'),
  (x, y) => x.assertNotEquals(y)
);
equivalent({ from: [smallField, smallFieldOrBigint], to: unit })(
  (x, y) => x < y || throwError('not less than'),
  (x, y) => x.assertLessThan(y)
);
equivalent({ from: [smallField, smallFieldOrBigint], to: unit })(
  (x, y) => x <= y || throwError('not less than or equal'),
  (x, y) => x.assertLessThanOrEqual(y)
);
equivalent({ from: [field], to: unit })(
  (x) => x === 0n || x === 1n || throwError('not boolean'),
  (x) => x.assertBool()
);
equivalent({ from: [smallField], to: bool })(
  (x) => (x & 1n) === 0n,
  (x) => x.isEven()
);

// non-constant field vars
test(Random.field, (x0, assert) => {
  Provable.runAndCheck(() => {
    // Var
    let x = Provable.witness(Field, () => Field(x0));
    assert(x.value[0] === FieldType.Var);
    assert(typeof x.value[1] === 'number');
    throws(() => x.toConstant());
    throws(() => x.toBigInt());
    Provable.asProver(() => assert(x.toBigInt() === x0));

    // Scale
    let z = x.mul(2);
    assert(z.value[0] === FieldType.Scale);
    throws(() => x.toConstant());

    // Add
    let u = z.add(x);
    assert(u.value[0] === FieldType.Add);
    throws(() => x.toConstant());
    Provable.asProver(() => assert(u.toBigInt() === Fp.mul(x0, 3n)));

    // seal
    let v = u.seal();
    assert(v.value[0] === FieldType.Var);
    Provable.asProver(() => assert(v.toBigInt() === Fp.mul(x0, 3n)));

    // Provable.witness / assertEquals / assertNotEquals
    let w0 = Provable.witness(Field, () => v.mul(5).add(1));
    let w1 = x.mul(15).add(1);
    w0.assertEquals(w1);
    throws(() => w0.assertNotEquals(w1));

    let w2 = Provable.witness(Field, () => w0.add(1));
    w0.assertNotEquals(w2);
    throws(() => w0.assertEquals(w2));
  });
});

// some provable operations
test(Random.field, Random.field, (x0, y0, assert) => {
  Provable.runAndCheck(() => {
    // equals
    let x = Provable.witness(Field, () => Field(x0));
    let y = Provable.witness(Field, () => Field(y0));

    let b = x.equals(y);
    b.assertEquals(x0 === y0);
    Provable.asProver(() => assert(b.toBoolean() === (x0 === y0)));

    let c = x.equals(x0);
    c.assertEquals(true);
    Provable.asProver(() => assert(c.toBoolean()));

    // mul
    let z = x.mul(y);
    Provable.asProver(() => assert(z.toBigInt() === Fp.mul(x0, y0)));

    // toBits / fromBits
    let bits = Fp.toBits(x0);
    let x1 = Provable.witness(Field, () => Field.fromBits(bits));
    let bitsVars = x1.toBits();
    Provable.asProver(() =>
      assert(bitsVars.every((b, i) => b.toBoolean() === bits[i]))
    );
  });
});
```

</file>

<file>

## path: /src/lib/gadgets/bitwise.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/gadgets/bitwise.ts

```
import { Provable } from '../provable.js';
import { Field as Fp } from '../../provable/field-bigint.js';
import { Field } from '../field.js';
import * as Gates from '../gates.js';
import {
  MAX_BITS,
  assert,
  witnessSlice,
  witnessNextValue,
  divideWithRemainder,
} from './common.js';
import { rangeCheck64 } from './range-check.js';
```

```
export { xor, not, rotate, and, rightShift, leftShift };

function not(a: Field, length: number, checked: boolean = false) {
  // check that input length is positive
  assert(length > 0, &nbsp;Input length needs to be positive values.&nbsp;);

  // Check that length does not exceed maximum field size in bits
  assert(
    length < Field.sizeInBits(),
    &nbsp;Length ${length} exceeds maximum of ${Field.sizeInBits()} bits.&nbsp;
  );

  // obtain pad length until the length is a multiple of 16 for n-bit length lookup table
  let padLength = Math.ceil(length / 16) * 16;

  // handle constant case
  if (a.isConstant()) {
    let max = 1n << BigInt(padLength);
    assert(
      a.toBigInt() < max,
      &nbsp;${a.toBigInt()} does not fit into ${padLength} bits&nbsp;
    );
    return new Field(Fp.not(a.toBigInt(), length));
  }

  // create a bitmask with all ones
  let allOnesF = new Field(2n ** BigInt(length) - 1n);

  let allOnes = Provable.witness(Field, () => {
    return allOnesF;
  });

  allOnesF.assertEquals(allOnes);

  if (checked) {
    return xor(a, allOnes, length);
  } else {
    return allOnes.sub(a);
  }
}

function xor(a: Field, b: Field, length: number) {
  // check that both input lengths are positive
  assert(length > 0, &nbsp;Input lengths need to be positive values.&nbsp;);

  // check that length does not exceed maximum 254 size in bits
  assert(length <= 254, &nbsp;Length ${length} exceeds maximum of 254 bits.&nbsp;);

  // obtain pad length until the length is a multiple of 16 for n-bit length lookup table
  let padLength = Math.ceil(length / 16) * 16;
  // handle constant case
  if (a.isConstant() && b.isConstant()) {
    let max = 1n << BigInt(padLength);

    assert(
      a.toBigInt() < max,
      &nbsp;${a.toBigInt()} does not fit into ${padLength} bits&nbsp;
    );

    assert(
      b.toBigInt() < max,
      &nbsp;${b.toBigInt()} does not fit into ${padLength} bits&nbsp;
    );

    return new Field(a.toBigInt() ^ b.toBigInt());
  }

  // calculate expected xor output
  let outputXor = Provable.witness(
    Field,
    () => new Field(a.toBigInt() ^ b.toBigInt())
  );

  // builds the xor gadget chain
  buildXor(a, b, outputXor, padLength);

  // return the result of the xor operation
  return outputXor;
}

// builds a xor chain
function buildXor(
  a: Field,
  b: Field,
  expectedOutput: Field,
  padLength: number
) {
  // construct the chain of XORs until padLength is 0
  while (padLength !== 0) {
    // slices the inputs into 4x 4bit-sized chunks
    // slices of a
    let in1_0 = witnessSlice(a, 0, 4);
    let in1_1 = witnessSlice(a, 4, 4);
    let in1_2 = witnessSlice(a, 8, 4);
    let in1_3 = witnessSlice(a, 12, 4);

    // slices of b
    let in2_0 = witnessSlice(b, 0, 4);
    let in2_1 = witnessSlice(b, 4, 4);
    let in2_2 = witnessSlice(b, 8, 4);
    let in2_3 = witnessSlice(b, 12, 4);
    // slices of expected output
    let out0 = witnessSlice(expectedOutput, 0, 4);
    let out1 = witnessSlice(expectedOutput, 4, 4);
    let out2 = witnessSlice(expectedOutput, 8, 4);
    let out3 = witnessSlice(expectedOutput, 12, 4);

    // assert that the xor of the slices is correct, 16 bit at a time
    Gates.xor(
      a,
      b,
      expectedOutput,
      in1_0,
      in1_1,
      in1_2,
      in1_3,
      in2_0,
      in2_1,
      in2_2,
      in2_3,
      out0,
      out1,
      out2,
      out3
    );

    // update the values for the next loop iteration
    a = witnessNextValue(a);
    b = witnessNextValue(b);
    expectedOutput = witnessNextValue(expectedOutput);
    padLength = padLength - 16;
  }

  // inputs are zero and length is zero, add the zero check - we reached the end of our chain
  Gates.zero(a, b, expectedOutput);

  let zero = new Field(0);
  zero.assertEquals(a);
  zero.assertEquals(b);
  zero.assertEquals(expectedOutput);
}

function and(a: Field, b: Field, length: number) {
  // check that both input lengths are positive
  assert(length > 0, &nbsp;Input lengths need to be positive values.&nbsp;);

  // check that length does not exceed maximum field size in bits
  assert(
    length <= Field.sizeInBits(),
    &nbsp;Length ${length} exceeds maximum of ${Field.sizeInBits()} bits.&nbsp;
  );

  // obtain pad length until the length is a multiple of 16 for n-bit length lookup table
  let padLength = Math.ceil(length / 16) * 16;

  // handle constant case
  if (a.isConstant() && b.isConstant()) {
    let max = 1n << BigInt(padLength);

    assert(
      a.toBigInt() < max,
      &nbsp;${a.toBigInt()} does not fit into ${padLength} bits&nbsp;
    );

    assert(
      b.toBigInt() < max,
      &nbsp;${b.toBigInt()} does not fit into ${padLength} bits&nbsp;
    );

    return new Field(a.toBigInt() & b.toBigInt());
  }

  // calculate expect and output
  let outputAnd = Provable.witness(
    Field,
    () => new Field(a.toBigInt() & b.toBigInt())
  );

  // compute values for gate
  // explanation: https://o1-labs.github.io/proof-systems/specs/kimchi.html?highlight=gates#and
  let sum = a.add(b);
  let xorOutput = xor(a, b, length);
  outputAnd.mul(2).add(xorOutput).assertEquals(sum);

  // return the result of the and operation
  return outputAnd;
}

function rotate(
  field: Field,
  bits: number,
  direction: 'left' | 'right' = 'left'
) {
  // Check that the rotation bits are in range
  assert(
    bits >= 0 && bits <= MAX_BITS,
    &nbsp;rotation: expected bits to be between 0 and 64, got ${bits}&nbsp;
  );

  if (field.isConstant()) {
    assert(
      field.toBigInt() < 2n ** BigInt(MAX_BITS),
      &nbsp;rotation: expected field to be at most 64 bits, got ${field.toBigInt()}&nbsp;
    );
    return new Field(Fp.rot(field.toBigInt(), bits, direction));
  }
  const [rotated] = rot(field, bits, direction);
  return rotated;
}

function rot(
  field: Field,
  bits: number,
  direction: 'left' | 'right' = 'left'
): [Field, Field, Field] {
  const rotationBits = direction === 'right' ? MAX_BITS - bits : bits;
  const big2Power64 = 2n ** BigInt(MAX_BITS);
  const big2PowerRot = 2n ** BigInt(rotationBits);

  const [rotated, excess, shifted, bound] = Provable.witness(
    Provable.Array(Field, 4),
    () => {
      const f = field.toBigInt();

      // Obtain rotated output, excess, and shifted for the equation:
      // f * 2^rot = excess * 2^64 + shifted
      const { quotient: excess, remainder: shifted } = divideWithRemainder(
        f * big2PowerRot,
        big2Power64
      );

      // Compute rotated value as: rotated = excess + shifted
      const rotated = shifted + excess;
      // Compute bound to check excess < 2^rot
      const bound = excess + big2Power64 - big2PowerRot;
      return [rotated, excess, shifted, bound].map(Field.from);
    }
  );

  // Compute current row
  Gates.rotate(
    field,
    rotated,
    excess,
    [
      witnessSlice(bound, 52, 12), // bits 52-64
      witnessSlice(bound, 40, 12), // bits 40-52
      witnessSlice(bound, 28, 12), // bits 28-40
      witnessSlice(bound, 16, 12), // bits 16-28
    ],
    [
      witnessSlice(bound, 14, 2), // bits 14-16
      witnessSlice(bound, 12, 2), // bits 12-14
      witnessSlice(bound, 10, 2), // bits 10-12
      witnessSlice(bound, 8, 2), // bits 8-10
      witnessSlice(bound, 6, 2), // bits 6-8
      witnessSlice(bound, 4, 2), // bits 4-6
      witnessSlice(bound, 2, 2), // bits 2-4
      witnessSlice(bound, 0, 2), // bits 0-2
    ],
    big2PowerRot
  );
  // Compute next row
  rangeCheck64(shifted);
  // Compute following row
  rangeCheck64(excess);
  return [rotated, excess, shifted];
}

function rightShift(field: Field, bits: number) {
  assert(
    bits >= 0 && bits <= MAX_BITS,
    &nbsp;rightShift: expected bits to be between 0 and 64, got ${bits}&nbsp;
  );

  if (field.isConstant()) {
    assert(
      field.toBigInt() < 2n ** BigInt(MAX_BITS),
      &nbsp;rightShift: expected field to be at most 64 bits, got ${field.toBigInt()}&nbsp;
    );
    return new Field(Fp.rightShift(field.toBigInt(), bits));
  }
  const [, excess] = rot(field, bits, 'right');
  return excess;
}

function leftShift(field: Field, bits: number) {
  assert(
    bits >= 0 && bits <= MAX_BITS,
    &nbsp;rightShift: expected bits to be between 0 and 64, got ${bits}&nbsp;
  );

  if (field.isConstant()) {
    assert(
      field.toBigInt() < 2n ** BigInt(MAX_BITS),
      &nbsp;rightShift: expected field to be at most 64 bits, got ${field.toBigInt()}&nbsp;
    );
    return new Field(Fp.leftShift(field.toBigInt(), bits));
  }
  const [, , shifted] = rot(field, bits, 'left');
  return shifted;
}
```

</file>

<file>

## path: /src/lib/gadgets/bitwise.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/gadgets/bitwise.unit-test.ts

```
import { ZkProgram } from '../proof_system.js';
import {
  equivalent,
  equivalentAsync,
  field,
  fieldWithRng,
} from '../testing/equivalent.js';
import { Fp, mod } from '../../bindings/crypto/finite_field.js';
import { Field } from '../core.js';
import { Gadgets } from './gadgets.js';
import { Random } from '../testing/property.js';

const maybeField = {
  ...field,
  rng: Random.map(Random.oneOf(Random.field, Random.field.invalid), (x) =>
    mod(x, Field.ORDER)
  ),
};

let uint = (length: number) => fieldWithRng(Random.biguint(length));

let Bitwise = ZkProgram({
  name: 'bitwise',
  publicOutput: Field,
  methods: {
    xor: {
      privateInputs: [Field, Field],
      method(a: Field, b: Field) {
        return Gadgets.xor(a, b, 254);
      },
    },
    notUnchecked: {
      privateInputs: [Field],
      method(a: Field) {
        return Gadgets.not(a, 254, false);
      },
    },
    notChecked: {
      privateInputs: [Field],
      method(a: Field) {
        return Gadgets.not(a, 254, true);
      },
    },
    and: {
      privateInputs: [Field, Field],
      method(a: Field, b: Field) {
        return Gadgets.and(a, b, 64);
      },
    },
    rot: {
      privateInputs: [Field],
      method(a: Field) {
        return Gadgets.rotate(a, 12, 'left');
      },
    },
    leftShift: {
      privateInputs: [Field],
      method(a: Field) {
        return Gadgets.leftShift(a, 12);
      },
    },
    rightShift: {
      privateInputs: [Field],
      method(a: Field) {
        return Gadgets.rightShift(a, 12);
      },
    },
  },
});

await Bitwise.compile();

[2, 4, 8, 16, 32, 64, 128].forEach((length) => {
  equivalent({ from: [uint(length), uint(length)], to: field })(
    (x, y) => x ^ y,
    (x, y) => Gadgets.xor(x, y, length)
  );
  equivalent({ from: [uint(length), uint(length)], to: field })(
    (x, y) => x & y,
    (x, y) => Gadgets.and(x, y, length)
  );
  // NOT unchecked
  equivalent({ from: [uint(length)], to: field })(
    (x) => Fp.not(x, length),
    (x) => Gadgets.not(x, length, false)
  );
  // NOT checked
  equivalent({ from: [uint(length)], to: field })(
    (x) => Fp.not(x, length),
    (x) => Gadgets.not(x, length, true)
  );
});

[2, 4, 8, 16, 32, 64].forEach((length) => {
  equivalent({ from: [uint(length)], to: field })(
    (x) => Fp.rot(x, 12, 'left'),
    (x) => Gadgets.rotate(x, 12, 'left')
  );
  equivalent({ from: [uint(length)], to: field })(
    (x) => Fp.leftShift(x, 12),
    (x) => Gadgets.leftShift(x, 12)
  );
  equivalent({ from: [uint(length)], to: field })(
    (x) => Fp.rightShift(x, 12),
    (x) => Gadgets.rightShift(x, 12)
  );
});

await equivalentAsync({ from: [uint(64), uint(64)], to: field }, { runs: 3 })(
  (x, y) => {
    return x ^ y;
  },
  async (x, y) => {
    let proof = await Bitwise.xor(x, y);
    return proof.publicOutput;
  }
);

await equivalentAsync({ from: [maybeField], to: field }, { runs: 3 })(
  (x) => {
    return Fp.not(x, 254);
  },
  async (x) => {
    let proof = await Bitwise.notUnchecked(x);
    return proof.publicOutput;
  }
);
await equivalentAsync({ from: [maybeField], to: field }, { runs: 3 })(
  (x) => {
    if (x > 2n ** 254n) throw Error('Does not fit into 254 bit');
    return Fp.not(x, 254);
  },
  async (x) => {
    let proof = await Bitwise.notChecked(x);
    return proof.publicOutput;
  }
);

await equivalentAsync(
  { from: [maybeField, maybeField], to: field },
  { runs: 3 }
)(
  (x, y) => {
    if (x >= 2n ** 64n || y >= 2n ** 64n)
      throw Error('Does not fit into 64 bits');
    return x & y;
  },
  async (x, y) => {
    let proof = await Bitwise.and(x, y);
    return proof.publicOutput;
  }
);

await equivalentAsync({ from: [field], to: field }, { runs: 3 })(
  (x) => {
    if (x >= 2n ** 64n) throw Error('Does not fit into 64 bits');
    return Fp.rot(x, 12, 'left');
  },
  async (x) => {
    let proof = await Bitwise.rot(x);
    return proof.publicOutput;
  }
);

await equivalentAsync({ from: [field], to: field }, { runs: 3 })(
  (x) => {
    if (x >= 2n ** 64n) throw Error('Does not fit into 64 bits');
    return Fp.leftShift(x, 12);
  },
  async (x) => {
    let proof = await Bitwise.leftShift(x);
    return proof.publicOutput;
  }
);

await equivalentAsync({ from: [field], to: field }, { runs: 3 })(
  (x) => {
    if (x >= 2n ** 64n) throw Error('Does not fit into 64 bits');
    return Fp.rightShift(x, 12);
  },
  async (x) => {
    let proof = await Bitwise.rightShift(x);
    return proof.publicOutput;
  }
);
```

</file>

<file>

## path: /src/lib/gadgets/common.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/gadgets/common.ts

```
import { Provable } from '../provable.js';
import { Field, FieldConst } from '../field.js';
import { TupleN } from '../util/types.js';
import { Snarky } from '../../snarky.js';
import { MlArray } from '../ml/base.js';

const MAX_BITS = 64 as const;

export {
  MAX_BITS,
  exists,
  assert,
  bitSlice,
  witnessSlice,
  witnessNextValue,
  divideWithRemainder,
};

function exists<N extends number, C extends () => TupleN<bigint, N>>(
  n: N,
  compute: C
) {
  let varsMl = Snarky.exists(n, () =>
    MlArray.mapTo(compute(), FieldConst.fromBigint)
  );
  let vars = MlArray.mapFrom(varsMl, (v) => new Field(v));
  return TupleN.fromArray(n, vars);
}

function assert(stmt: boolean, message?: string) {
  if (!stmt) {
    throw Error(message ?? 'Assertion failed');
  }
}

function bitSlice(x: bigint, start: number, length: number) {
  return (x >> BigInt(start)) & ((1n << BigInt(length)) - 1n);
}

function witnessSlice(f: Field, start: number, length: number) {
  if (length <= 0) throw Error('Length must be a positive number');

  return Provable.witness(Field, () => {
    let n = f.toBigInt();
    return new Field((n >> BigInt(start)) & ((1n << BigInt(length)) - 1n));
  });
}

function witnessNextValue(current: Field) {
  return Provable.witness(Field, () => new Field(current.toBigInt() >> 16n));
}

function divideWithRemainder(numerator: bigint, denominator: bigint) {
  const quotient = numerator / denominator;
  const remainder = numerator - denominator * quotient;
  return { quotient, remainder };
}
```

</file>

<file>

path: /src/lib/gadgets/gadgets.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/gadgets/gadgets.ts

```
/**
 * Wrapper file for various gadgets, with a namespace and doccomments.
 */
import {
  compactMultiRangeCheck,
  multiRangeCheck,
  rangeCheck64,
} from './range-check.js';
import { not, rotate, xor, and, leftShift, rightShift } from './bitwise.js';
import { Field } from '../core.js';

export { Gadgets };

const Gadgets = {
  /**
   * Asserts that the input value is in the range [0, 2^64).
   *
   * This function proves that the provided field element can be represented with 64 bits.
   * If the field element exceeds 64 bits, an error is thrown.
   *
   * @param x - The value to be range-checked.
   *
   * @throws Throws an error if the input value exceeds 64 bits.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const x = Provable.witness(Field, () => Field(12345678n));
   * Gadgets.rangeCheck64(x); // successfully proves 64-bit range
   *
   * const xLarge = Provable.witness(Field, () => Field(12345678901234567890123456789012345678n));
   * Gadgets.rangeCheck64(xLarge); // throws an error since input exceeds 64 bits
   * &nbsp;&nbsp;&nbsp;
   *
   * **Note**: Small "negative" field element inputs are interpreted as large integers close to the field size,
   * and don't pass the 64-bit check. If you want to prove that a value lies in the int64 range [-2^63, 2^63),
   * you could use &nbsp;rangeCheck64(x.add(1n << 63n))&nbsp;.
   */
  rangeCheck64(x: Field) {
    return rangeCheck64(x);
  },
  /**
   * A (left and right) rotation operates similarly to the shift operation (&nbsp;<<&nbsp; for left and
&nbsp;>>&nbsp; for right) in JavaScript,
   * with the distinction that the bits are circulated to the opposite end of a 64-bit representation rather than
being discarded.
   * For a left rotation, this means that bits shifted off the left end reappear at the right end.
   * Conversely, for a right rotation, bits shifted off the right end reappear at the left end.
   *
   * It’s important to note that these operations are performed considering the big-endian 64-bit
representation of the number,
   * where the most significant (64th) bit is on the left end and the least significant bit is on the right end.
   * The &nbsp;direction&nbsp; parameter is a string that accepts either &nbsp;'left'&nbsp; or
&nbsp;'right'&nbsp;, determining the direction of the rotation.
   *
   * **Important:** The gadget assumes that its input is at most 64 bits in size.
   *
   * If the input exceeds 64 bits, the gadget is invalid and fails to prove correct execution of the rotation.
   * To safely use &nbsp;rotate()&nbsp;, you need to make sure that the value passed in is range-checked to
64 bits;
   * for example, using {@link Gadgets.rangeCheck64}.
   *
   * You can find more details about the implementation in the [Mina book](https://o1-labs.github.io/proof-
systems/specs/kimchi.html?highlight=gates#rotation)
   *
   * @param field {@link Field} element to rotate.
   * @param bits amount of bits to rotate this {@link Field} element with.
   * @param direction left or right rotation direction.
   *
   * @throws Throws an error if the input value exceeds 64 bits.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const x = Provable.witness(Field, () => Field(0b001100));
   * const y = Gadgets.rotate(x, 2, 'left'); // left rotation by 2 bits
   * const z = Gadgets.rotate(x, 2, 'right'); // right rotation by 2 bits
   * y.assertEquals(0b110000);
   * z.assertEquals(0b000011);
   *
   * const xLarge = Provable.witness(Field, () => Field(12345678901234567890123456789012345678n));
   * Gadgets.rotate(xLarge, 32, "left"); // throws an error since input exceeds 64 bits
   * &nbsp;&nbsp;&nbsp;
   */
  rotate(field: Field, bits: number, direction: 'left' | 'right' = 'left') {
    return rotate(field, bits, direction);
  },
  /**
   * Bitwise XOR gadget on {@link Field} elements. Equivalent to the [bitwise XOR &nbsp;^&nbsp; operator
in JavaScript](https://developer.mozilla.org/en-
US/docs/Web/JavaScript/Reference/Operators/Bitwise_XOR).
   * A XOR gate works by comparing two bits and returning &nbsp;1&nbsp; if two bits differ, and
&nbsp;0&nbsp; if two bits are equal.
   *
   * This gadget builds a chain of XOR gates recursively. Each XOR gate can verify 16 bit at most. If your
input elements exceed 16 bit, another XOR gate will be added to the chain.
   *
   * The &nbsp;length&nbsp; parameter lets you define how many bits should be compared.
&nbsp;length&nbsp; is rounded to the nearest multiple of 16, &nbsp;paddedLength = ceil(length / 16) *
16&nbsp;, and both input values are constrained to fit into &nbsp;paddedLength&nbsp; bits. The output is
guaranteed to have at most &nbsp;paddedLength&nbsp; bits as well.
```

```
   *
   * **Note:** Specifying a larger &nbsp;length&nbsp; parameter adds additional constraints.
   *
   * It is also important to mention that specifying a smaller &nbsp;length&nbsp; allows the verifier to infer the
length of the original input data (e.g. smaller than 16 bit if only one XOR gate has been used).
   * A zkApp developer should consider these implications when choosing the &nbsp;length&nbsp;
parameter and carefully weigh the trade-off between increased amount of constraints and security.
   *
   * **Important:** Both {@link Field} elements need to fit into &nbsp;2^paddedLength - 1&nbsp;. Otherwise,
an error is thrown and no proof can be generated.
   *
   * For example, with &nbsp;length = 2&nbsp; (&nbsp;paddedLength = 16&nbsp;), &nbsp;xor()&nbsp; will
fail for any input that is larger than &nbsp;2**16&nbsp;.
   *
   * You can find more details about the implementation in the [Mina book](https://o1-labs.github.io/proof-
systems/specs/kimchi.html?highlight=gates#xor-1)
   *
   * @param a {@link Field} element to compare.
   * @param b {@link Field} element to compare.
   * @param length amount of bits to compare.
   *
   * @throws Throws an error if the input values exceed &nbsp;2^paddedLength - 1&nbsp;.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * let a = Field(0b0101);
   * let b = Field(0b0011);
   *
   * let c = Gadgets.xor(a, b, 4); // xor-ing 4 bits
   * c.assertEquals(0b0110);
   * &nbsp;&nbsp;&nbsp;
   */
  xor(a: Field, b: Field, length: number) {
    return xor(a, b, length);
  },

  /**
   * Bitwise NOT gate on {@link Field} elements. Similar to the [bitwise
   * NOT &nbsp;~&nbsp; operator in JavaScript](https://developer.mozilla.org/en-US/docs/
   * Web/JavaScript/Reference/Operators/Bitwise_NOT).
   *
   * **Note:** The NOT gate only operates over the amount
   * of bits specified by the &nbsp;length&nbsp; parameter.
   *
   * A NOT gate works by returning &nbsp;1&nbsp; in each bit position if the
   * corresponding bit of the operand is &nbsp;0&nbsp;, and returning &nbsp;0&nbsp; if the
   * corresponding bit of the operand is &nbsp;1&nbsp;.
   *
   * The &nbsp;length&nbsp; parameter lets you define how many bits to NOT.
   *
   * **Note:** Specifying a larger &nbsp;length&nbsp; parameter adds additional constraints. The operation
will fail if the length or the input value is larger than 254.
```

```
   *
   * NOT is implemented in two different ways. If the &nbsp;checked&nbsp; parameter is set to
&nbsp;true&nbsp;
   * the {@link Gadgets.xor} gadget is reused with a second argument to be an
   * all one bitmask the same length. This approach needs as many rows as an XOR would need
   * for a single negation. If the &nbsp;checked&nbsp; parameter is set to &nbsp;false&nbsp;, NOT is
   * implemented as a subtraction of the input from the all one bitmask. This
   * implementation is returned by default if no &nbsp;checked&nbsp; parameter is provided.
   *
   * You can find more details about the implementation in the [Mina book](https://o1-labs.github.io/proof-
systems/specs/kimchi.html?highlight=gates#not)
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * // not-ing 4 bits with the unchecked version
   * let a = Field(0b0101);
   * let b = Gadgets.not(a,4,false);
   *
   * b.assertEquals(0b1010);
   *
   * // not-ing 4 bits with the checked version utilizing the xor gadget
   * let a = Field(0b0101);
   * let b = Gadgets.not(a,4,true);
   *
   * b.assertEquals(0b1010);
   * &nbsp;&nbsp;&nbsp;
   *
   * @param a - The value to apply NOT to. The operation will fail if the value is larger than 254.
   * @param length - The number of bits to be considered for the NOT operation.
   * @param checked - Optional boolean to determine if the checked or unchecked not implementation is
used. If it
   * is set to &nbsp;true&nbsp; the {@link Gadgets.xor} gadget is reused. If it is set to &nbsp;false&nbsp;,
NOT is implemented
   *  as a subtraction of the input from the all one bitmask. It is set to &nbsp;false&nbsp; by default if no
parameter is provided.
   *
   * @throws Throws an error if the input value exceeds 254 bits.
   */
  not(a: Field, length: number, checked: boolean = false) {
    return not(a, length, checked);
  },

  /**
   * Performs a left shift operation on the provided {@link Field} element.
   * This operation is similar to the &nbsp;<<&nbsp; shift operation in JavaScript,
   * where bits are shifted to the left, and the overflowing bits are discarded.
   *
   * It's important to note that these operations are performed considering the big-endian 64-bit
representation of the number,
   * where the most significant (64th) bit is on the left end and the least significant bit is on the right end.
   *
   * **Important:** The gadgets assumes that its input is at most 64 bits in size.
   *
   * If the input exceeds 64 bits, the gadget is invalid and fails to prove correct execution of the shift.
   * Therefore, to safely use &nbsp;leftShift()&nbsp;, you need to make sure that the values passed in are
range checked to 64 bits.
   * For example, this can be done with {@link Gadgets.rangeCheck64}.
   *
   * @param field {@link Field} element to shift.
   * @param bits Amount of bits to shift the {@link Field} element to the left. The amount should be between
0 and 64 (or else the shift will fail).
   *
   * @throws Throws an error if the input value exceeds 64 bits.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const x = Provable.witness(Field, () => Field(0b001100)); // 12 in binary
   * const y = Gadgets.leftShift(x, 2); // left shift by 2 bits
   * y.assertEquals(0b110000); // 48 in binary
   *
   * const xLarge = Provable.witness(Field, () => Field(12345678901234567890123456789012345678n));
   * leftShift(xLarge, 32); // throws an error since input exceeds 64 bits
   * &nbsp;&nbsp;&nbsp;
   */
  leftShift(field: Field, bits: number) {
    return leftShift(field, bits);
  },

  /**
   * Performs a right shift operation on the provided {@link Field} element.
   * This is similar to the &nbsp;>>&nbsp; shift operation in JavaScript, where bits are moved to the right.
   * The &nbsp;rightShift&nbsp; function utilizes the rotation method internally to implement this operation.
   *
   * * It’s important to note that these operations are performed considering the big-endian 64-bit
representation of the number,
   * where the most significant (64th) bit is on the left end and the least significant bit is on the right end.
   *
   * **Important:** The gadgets assumes that its input is at most 64 bits in size.
   *
   * If the input exceeds 64 bits, the gadget is invalid and fails to prove correct execution of the shift.
   * To safely use &nbsp;rightShift()&nbsp;, you need to make sure that the value passed in is range-
checked to 64 bits;
   * for example, using {@link Gadgets.rangeCheck64}.
   *
   * @param field {@link Field} element to shift.
   * @param bits Amount of bits to shift the {@link Field} element to the right. The amount should be
between 0 and 64 (or else the shift will fail).
   *
   * @throws Throws an error if the input value exceeds 64 bits.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const x = Provable.witness(Field, () => Field(0b001100)); // 12 in binary
   * const y = Gadgets.rightShift(x, 2); // right shift by 2 bits
   * y.assertEquals(0b000011); // 3 in binary
   *
   * const xLarge = Provable.witness(Field, () => Field(12345678901234567890123456789012345678n));
   * rightShift(xLarge, 32); // throws an error since input exceeds 64 bits
   * &nbsp;&nbsp;&nbsp;
   */
  rightShift(field: Field, bits: number) {
    return rightShift(field, bits);
  },
  /**
   * Bitwise AND gadget on {@link Field} elements. Equivalent to the [bitwise AND &nbsp;&&nbsp; operator
in JavaScript](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Bitwise_AND).
   * The AND gate works by comparing two bits and returning &nbsp;1&nbsp; if both bits are &nbsp;1&nbsp;,
and &nbsp;0&nbsp; otherwise.
   *
   * It can be checked by a double generic gate that verifies the following relationship between the values
below (in the process it also invokes the {@link Gadgets.xor} gadget which will create additional constraints
depending on &nbsp;length&nbsp;).
   *
   * The generic gate verifies:\
   * &nbsp;a + b = sum&nbsp; and the conjunction equation &nbsp;2 * and = sum - xor&nbsp;\
   * Where:\
   * &nbsp;a + b = sum&nbsp;\
   * &nbsp;a ^ b = xor&nbsp;\
   * &nbsp;a & b = and&nbsp;
   *
   * You can find more details about the implementation in the [Mina book](https://o1-labs.github.io/proof-
systems/specs/kimchi.html?highlight=gates#and)
   *
   * The &nbsp;length&nbsp; parameter lets you define how many bits should be compared.
&nbsp;length&nbsp; is rounded to the nearest multiple of 16, &nbsp;paddedLength = ceil(length / 16) *
16&nbsp;, and both input values are constrained to fit into &nbsp;paddedLength&nbsp; bits. The output is
guaranteed to have at most &nbsp;paddedLength&nbsp; bits as well.
   *
   * **Note:** Specifying a larger &nbsp;length&nbsp; parameter adds additional constraints.
   *
   * **Note:** Both {@link Field} elements need to fit into &nbsp;2^paddedLength - 1&nbsp;. Otherwise, an
error is thrown and no proof can be generated.
   * For example, with &nbsp;length = 2&nbsp; (&nbsp;paddedLength = 16&nbsp;), &nbsp;and()&nbsp; will
fail for any input that is larger than &nbsp;2**16&nbsp;.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;typescript
   * let a = Field(3);    // ... 000011
   * let b = Field(5);    // ... 000101
   *
   * let c = Gadgets.and(a, b, 2);    // ... 000001
   * c.assertEquals(1);
   * &nbsp;&nbsp;&nbsp;
   */
  and(a: Field, b: Field, length: number) {
    return and(a, b, length);
  },

  /**
   * Multi-range check.
   *
   * Proves that x, y, z are all in the range [0, 2^88).
   *
   * This takes 4 rows, so it checks 88*3/4 = 66 bits per row. This is slightly more efficient
   * than 64-bit range checks, which can do 64 bits in 1 row.
   *
   * In particular, the 3x88-bit range check supports bigints up to 264 bits, which in turn is enough
   * to support foreign field multiplication with moduli up to 2^259.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * Gadgets.multiRangeCheck([x, y, z]);
   * &nbsp;&nbsp;&nbsp;
   *
   * @throws Throws an error if one of the input values exceeds 88 bits.
   */
  multiRangeCheck(limbs: [Field, Field, Field]) {
    multiRangeCheck(limbs);
  },

  /**
   * Compact multi-range check
   *
   * This is a variant of {@link multiRangeCheck} where the first two variables are passed in
   * combined form xy = x + 2^88*y.
   *
   * The gadget
   * - splits up xy into x and y
   * - proves that xy = x + 2^88*y
   * - proves that x, y, z are all in the range [0, 2^88).
   *
   * The split form [x, y, z] is returned.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * let [x, y] = Gadgets.compactMultiRangeCheck([xy, z]);
   * &nbsp;&nbsp;&nbsp;
   *
   * @throws Throws an error if &nbsp;xy&nbsp; exceeds 2*88 = 176 bits, or if z exceeds 88 bits.
   */
  compactMultiRangeCheck(xy: Field, z: Field) {
    return compactMultiRangeCheck(xy, z);
  },
};
```

</file>

<file>

## path: /src/lib/gadgets/range-check.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/gadgets/range-check.ts

```
import { Field } from '../field.js';
import * as Gates from '../gates.js';
import { bitSlice, exists } from './common.js';

export { rangeCheck64, multiRangeCheck, compactMultiRangeCheck, L };

/**
 * Asserts that x is in the range [0, 2^64)
 */
function rangeCheck64(x: Field) {
  if (x.isConstant()) {
    if (x.toBigInt() >= 1n << 64n) {
      throw Error(&nbsp;rangeCheck64: expected field to fit in 64 bits, got ${x}&nbsp;);
    }
    return;
  }

  // crumbs (2-bit limbs)
  let [x0, x2, x4, x6, x8, x10, x12, x14] = exists(8, () => {
    let xx = x.toBigInt();
    return [
      bitSlice(xx, 0, 2),
      bitSlice(xx, 2, 2),
      bitSlice(xx, 4, 2),
      bitSlice(xx, 6, 2),
      bitSlice(xx, 8, 2),
      bitSlice(xx, 10, 2),
      bitSlice(xx, 12, 2),
      bitSlice(xx, 14, 2),
    ];
  });

  // 12-bit limbs
  let [x16, x28, x40, x52] = exists(4, () => {
    let xx = x.toBigInt();
    return [
      bitSlice(xx, 16, 12),
      bitSlice(xx, 28, 12),
      bitSlice(xx, 40, 12),
      bitSlice(xx, 52, 12),
    ];
  });

  Gates.rangeCheck0(
    x,
    [new Field(0), new Field(0), x52, x40, x28, x16],
    [x14, x12, x10, x8, x6, x4, x2, x0],
    false // not using compact mode
  );
}

// default bigint limb size
const L = 88n;
const twoL = 2n * L;
const lMask = (1n << L) - 1n;

/**
 * Asserts that x, y, z \in [0, 2^88)
 */
function multiRangeCheck([x, y, z]: [Field, Field, Field]) {
  if (x.isConstant() && y.isConstant() && z.isConstant()) {
    if (x.toBigInt() >> L || y.toBigInt() >> L || z.toBigInt() >> L) {
      throw Error(&nbsp;Expected fields to fit in ${L} bits, got ${x}, ${y}, ${z}&nbsp;);
    }
    return;
  }

  let [x64, x76] = rangeCheck0Helper(x);
  let [y64, y76] = rangeCheck0Helper(y);
  rangeCheck1Helper({ x64, x76, y64, y76, z, yz: new Field(0) });
}

/**
 * Compact multi-range-check - checks
 * - xy = x + 2^88*y
 * - x, y, z \in [0, 2^88)
 *
 * Returns the full limbs x, y, z
 */
function compactMultiRangeCheck(xy: Field, z: Field): [Field, Field, Field] {
  // constant case
  if (xy.isConstant() && z.isConstant()) {
    if (xy.toBigInt() >> twoL || z.toBigInt() >> L) {
      throw Error(
        &nbsp;Expected fields to fit in ${twoL} and ${L} bits respectively, got ${xy}, ${z}&nbsp;
      );
    }
    let [x, y] = splitCompactLimb(xy.toBigInt());
    return [new Field(x), new Field(y), z];
  }

  let [x, y] = exists(2, () => splitCompactLimb(xy.toBigInt()));

  let [z64, z76] = rangeCheck0Helper(z, false);
  let [x64, x76] = rangeCheck0Helper(x, true);
  rangeCheck1Helper({ x64: z64, x76: z76, y64: x64, y76: x76, z: y, yz: xy });
  return [x, y, z];
}

function splitCompactLimb(x01: bigint): [bigint, bigint] {
  return [x01 & lMask, x01 >> L];
}

function rangeCheck0Helper(x: Field, isCompact = false): [Field, Field] {
  // crumbs (2-bit limbs)
  let [x0, x2, x4, x6, x8, x10, x12, x14] = exists(8, () => {
    let xx = x.toBigInt();
    return [
      bitSlice(xx, 0, 2),
      bitSlice(xx, 2, 2),
      bitSlice(xx, 4, 2),
      bitSlice(xx, 6, 2),
      bitSlice(xx, 8, 2),
      bitSlice(xx, 10, 2),
      bitSlice(xx, 12, 2),
      bitSlice(xx, 14, 2),
    ];
  });

  // 12-bit limbs
  let [x16, x28, x40, x52, x64, x76] = exists(6, () => {
    let xx = x.toBigInt();
    return [
      bitSlice(xx, 16, 12),
      bitSlice(xx, 28, 12),
      bitSlice(xx, 40, 12),
      bitSlice(xx, 52, 12),
      bitSlice(xx, 64, 12),
      bitSlice(xx, 76, 12),
    ];
  });

  Gates.rangeCheck0(
    x,
    [x76, x64, x52, x40, x28, x16],
    [x14, x12, x10, x8, x6, x4, x2, x0],
    isCompact
  );

  // the two highest 12-bit limbs are returned because another gate
  // is needed to add lookups for them
  return [x64, x76];
}

function rangeCheck1Helper(inputs: {
  x64: Field;
  x76: Field;
  y64: Field;
  y76: Field;
  z: Field;
  yz: Field;
}) {
  let { x64, x76, y64, y76, z, yz } = inputs;

  // create limbs for current row
  let [z22, z24, z26, z28, z30, z32, z34, z36, z38, z50, z62, z74, z86] =
    exists(13, () => {
      let zz = z.toBigInt();
      return [
        bitSlice(zz, 22, 2),
        bitSlice(zz, 24, 2),
        bitSlice(zz, 26, 2),
        bitSlice(zz, 28, 2),
        bitSlice(zz, 30, 2),
        bitSlice(zz, 32, 2),
        bitSlice(zz, 34, 2),
        bitSlice(zz, 36, 2),
        bitSlice(zz, 38, 12),
        bitSlice(zz, 50, 12),
        bitSlice(zz, 62, 12),
        bitSlice(zz, 74, 12),
        bitSlice(zz, 86, 2),
      ];
    });

  // create limbs for next row
  let [z0, z2, z4, z6, z8, z10, z12, z14, z16, z18, z20] = exists(11, () => {
    let zz = z.toBigInt();
    return [
      bitSlice(zz, 0, 2),
      bitSlice(zz, 2, 2),
      bitSlice(zz, 4, 2),
      bitSlice(zz, 6, 2),
      bitSlice(zz, 8, 2),
      bitSlice(zz, 10, 2),
      bitSlice(zz, 12, 2),
      bitSlice(zz, 14, 2),
      bitSlice(zz, 16, 2),
      bitSlice(zz, 18, 2),
      bitSlice(zz, 20, 2),
    ];
  });

  Gates.rangeCheck1(
    z,
    yz,
    [z86, z74, z62, z50, z38, z36, z34, z32, z30, z28, z26, z24, z22],
    [z20, z18, z16, x76, x64, y76, y64, z14, z12, z10, z8, z6, z4, z2, z0]
  );
}
```

</file>

<file>

## path: /src/lib/gadgets/range-check.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/gadgets/range-check.unit-test.ts

```
import type { Gate } from '../../snarky.js';
import { mod } from '../../bindings/crypto/finite_field.js';
import { Field } from '../../lib/core.js';
import { ZkProgram } from '../proof_system.js';
import { Provable } from '../provable.js';
import {
  Spec,
  boolean,
  equivalentAsync,
  fieldWithRng,
} from '../testing/equivalent.js';
import { Random } from '../testing/property.js';
import { assert, exists } from './common.js';
import { Gadgets } from './gadgets.js';
import { L } from './range-check.js';
import { expect } from 'expect';

let uint = (n: number | bigint): Spec<bigint, Field> => {
  let uint = Random.bignat((1n << BigInt(n)) - 1n);
  return fieldWithRng(uint);
};

let maybeUint = (n: number | bigint): Spec<bigint, Field> => {
  let uint = Random.bignat((1n << BigInt(n)) - 1n);
  return fieldWithRng(
    Random.map(Random.oneOf(uint, uint.invalid), (x) => mod(x, Field.ORDER))
  );
};

// constraint system sanity check

function csWithoutGenerics(gates: Gate[]) {
  return gates.map((g) => g.type).filter((type) => type !== 'Generic');
}

let check64 = Provable.constraintSystem(() => {
  let [x] = exists(1, () => [0n]);
  Gadgets.rangeCheck64(x);
});
let multi = Provable.constraintSystem(() => {
  let x = exists(3, () => [0n, 0n, 0n]);
  Gadgets.multiRangeCheck(x);
```

```
});
let compact = Provable.constraintSystem(() => {
  let [xy, z] = exists(2, () => [0n, 0n]);
  Gadgets.compactMultiRangeCheck(xy, z);
});

let expectedLayout64 = ['RangeCheck0'];
let expectedLayoutMulti = ['RangeCheck0', 'RangeCheck0', 'RangeCheck1', 'Zero'];

expect(csWithoutGenerics(check64.gates)).toEqual(expectedLayout64);
expect(csWithoutGenerics(multi.gates)).toEqual(expectedLayoutMulti);
expect(csWithoutGenerics(compact.gates)).toEqual(expectedLayoutMulti);

// TODO: make a ZkFunction or something that doesn't go through Pickles
// --------------------------
// RangeCheck64 Gate
// --------------------------

let RangeCheck = ZkProgram({
  name: 'range-check',
  methods: {
    check64: {
      privateInputs: [Field],
      method(x) {
        Gadgets.rangeCheck64(x);
      },
    },
    checkMulti: {
      privateInputs: [Field, Field, Field],
      method(x, y, z) {
        Gadgets.multiRangeCheck([x, y, z]);
      },
    },
    checkCompact: {
      privateInputs: [Field, Field],
      method(xy, z) {
        let [x, y] = Gadgets.compactMultiRangeCheck(xy, z);
        x.add(y.mul(1n << L)).assertEquals(xy);
      },
    },
  },
});

await RangeCheck.compile();

// TODO: we use this as a test because there's no way to check custom gates quickly :(

await equivalentAsync({ from: [maybeUint(64)], to: boolean }, { runs: 3 })(
  (x) => {
    assert(x < 1n << 64n);
    return true;
  },
  async (x) => {
    let proof = await RangeCheck.check64(x);
    return await RangeCheck.verify(proof);
  }
);

await equivalentAsync(
  { from: [maybeUint(L), uint(L), uint(L)], to: boolean },
  { runs: 3 }
)(
  (x, y, z) => {
    assert(!(x >> L) && !(y >> L) && !(z >> L), 'multi: not out of range');
    return true;
  },
  async (x, y, z) => {
    let proof = await RangeCheck.checkMulti(x, y, z);
    return await RangeCheck.verify(proof);
  }
);

await equivalentAsync(
  { from: [maybeUint(2n * L), uint(L)], to: boolean },
  { runs: 3 }
)(
  (xy, z) => {
    assert(!(xy >> (2n * L)) && !(z >> L), 'compact: not out of range');
    return true;
  },
  async (xy, z) => {
    let proof = await RangeCheck.checkCompact(xy, z);
    return await RangeCheck.verify(proof);
  }
);
```

</file>

<file>

## path: /src/lib/gates.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/gates.ts

```
import { Snarky } from '../snarky.js';
import { FieldConst, type Field } from './field.js';
import { MlArray, MlTuple } from './ml/base.js';
import { TupleN } from './util/types.js';

export { rangeCheck0, rangeCheck1, xor, zero, rotate, generic };

function rangeCheck0(
  x: Field,
  xLimbs12: TupleN<Field, 6>,
  xLimbs2: TupleN<Field, 8>,
  isCompact: boolean
) {
  Snarky.gates.rangeCheck0(
    x.value,
    MlTuple.mapTo(xLimbs12, (x) => x.value),
    MlTuple.mapTo(xLimbs2, (x) => x.value),
    isCompact ? FieldConst[1] : FieldConst[0]
  );
}

/**
 * the rangeCheck1 gate is used in combination with the rangeCheck0,
 * for doing a 3x88-bit range check
 */
function rangeCheck1(
  v2: Field,
  v12: Field,
  vCurr: TupleN<Field, 13>,
  vNext: TupleN<Field, 15>
) {
  Snarky.gates.rangeCheck1(
    v2.value,
    v12.value,
    MlTuple.mapTo(vCurr, (x) => x.value),
    MlTuple.mapTo(vNext, (x) => x.value)
  );
}

function rotate(
  field: Field,
  rotated: Field,
  excess: Field,
  limbs: [Field, Field, Field, Field],
  crumbs: [Field, Field, Field, Field, Field, Field, Field, Field],
  two_to_rot: bigint
) {
  Snarky.gates.rotate(
    field.value,
    rotated.value,
    excess.value,
    MlArray.to(limbs.map((x) => x.value)),
    MlArray.to(crumbs.map((x) => x.value)),
    FieldConst.fromBigint(two_to_rot)
  );
}

/**
 * Asserts that 16 bit limbs of input two elements are the correct XOR output
 */
function xor(
  input1: Field,
  input2: Field,
  outputXor: Field,
  in1_0: Field,
  in1_1: Field,
  in1_2: Field,
  in1_3: Field,
  in2_0: Field,
  in2_1: Field,
  in2_2: Field,
  in2_3: Field,
  out0: Field,
  out1: Field,
  out2: Field,
  out3: Field
) {
  Snarky.gates.xor(
    input1.value,
    input2.value,
    outputXor.value,
    in1_0.value,
    in1_1.value,
    in1_2.value,
    in1_3.value,
    in2_0.value,
    in2_1.value,
    in2_2.value,
    in2_3.value,
    out0.value,
    out1.value,
    out2.value,
    out3.value
  );
}

/**
 * [Generic gate](https://o1-labs.github.io/proof-systems/specs/kimchi.html?highlight=foreignfield#double-generic-gate)
 * The vanilla PLONK gate that allows us to do operations like:
 * * addition of two registers (into an output register)
 * * multiplication of two registers
 * * equality of a register with a constant
 *
 * More generally, the generic gate controls the coefficients (denoted &nbsp;c_&nbsp;) in the equation:
 *
 * &nbsp;c_l*l + c_r*r + c_o*o + c_m*l*r + c_c === 0&nbsp;
 */
function generic(
  coefficients: {
    left: bigint;
    right: bigint;
    out: bigint;
    mul: bigint;
    const: bigint;
  },
  inputs: { left: Field; right: Field; out: Field }
) {
  Snarky.gates.generic(
    FieldConst.fromBigint(coefficients.left),
    inputs.left.value,
    FieldConst.fromBigint(coefficients.right),
    inputs.right.value,
    FieldConst.fromBigint(coefficients.out),
    inputs.out.value,
    FieldConst.fromBigint(coefficients.mul),
    FieldConst.fromBigint(coefficients.const)
  );
}

function zero(a: Field, b: Field, c: Field) {
  Snarky.gates.zero(a.value, b.value, c.value);
}
```

</file>

<file>

## path: /src/lib/global-context.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/global-context.ts

```
export { Context };

namespace Context {
  export type id = number;

  export type t<Context> = (() => Context | undefined) & {
    data: { context: Context; id: id }[];
    allowsNesting: boolean;

    get(): Context;
    has(): boolean;
    runWith<C extends Context, Result>(
      context: Context,
      func: (context: C) => Result
    ): [C, Result];
    runWithAsync<Result>(
      context: Context,
      func: (context: Context) => Promise<Result>
    ): Promise<[Context, Result]>;
    enter(context: Context): id;
    leave(id: id): Context;
    id: () => id;
  };
}
const Context = { create };

function create<C>(
  options = {
    allowsNesting: true,
    default: undefined,
  } as { allowsNesting?: boolean; default?: C }
): Context.t<C> {
  let t: Context.t<C> = Object.assign(
    function (): C | undefined {
      return t.data[t.data.length - 1]?.context;
    },
    {
      data: [],
      allowsNesting: options.allowsNesting ?? true,
      get: () => get(t),
      has: () => t.data.length !== 0,
      runWith<C0 extends C, Result>(
        context: C0,
        func: (context: C0) => Result
      ): [C0, Result] {
        let id = enter(t, context);
        let result: Result;
        let resultContext: C;
        try {
          result = func(context);
        } finally {
          resultContext = leave(t, id);
        }
        return [resultContext as C0, result];
      },
      async runWithAsync<Result>(
        context: C,
        func: (context: C) => Promise<Result>
      ): Promise<[C, Result]> {
        let id = enter(t, context);
        let result: Result;
        let resultContext: C;
        try {
          result = await func(context);
        } finally {
          resultContext = leave(t, id);
        }
        return [resultContext, result];
      },
      enter: (context: C) => enter(t, context),
      leave: (id: Context.id) => leave(t, id),
      id: () => {
        if (t.data.length === 0) throw Error(contextConflictMessage);
        return t.data[t.data.length - 1].id;
      },
    }
  );
  if (options.default !== undefined) enter(t, options.default);
  return t;
 }

 function enter<C>(t: Context.t<C>, context: C): Context.id {
  if (t.data.length > 0 && !t.allowsNesting) {
    throw Error(contextConflictMessage);
  }
  let id = Math.random();
  t.data.push({ context, id });
  return id;
 }

 function leave<C>(t: Context.t<C>, id: Context.id): C {
  let current = t.data.pop();
  if (current === undefined) throw Error(contextConflictMessage);
  if (current.id !== id) throw Error(contextConflictMessage);
  return current.context;
 }

 function get<C>(t: Context.t<C>): C {
  if (t.data.length === 0) throw Error(contextConflictMessage);
  let current = t.data[t.data.length - 1];
  return current.context;
 }

 let contextConflictMessage =
  "It seems you're running multiple provers concurrently within" +
  ' the same JavaScript thread, which, at the moment, is not supported and would lead to bugs.';
```

</file>

<file>

## path: /src/lib/group.test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/group.test.ts

```
import { Bool, Group, Scalar, Provable } from 'o1js';

describe('group', () => {
 let g = Group({
   x: -1,
   y: 2,
 });

 describe('Inside circuit', () => {
   describe('group membership', () => {
```

```
  it('valid element does not throw', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        Provable.witness(Group, () => g);
      });
    }).not.toThrow();
  });

  it('valid element does not throw', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        Provable.witness(Group, () => Group.generator);
      });
    }).not.toThrow();
  });

  it('Group.zero element does not throw', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        Provable.witness(Group, () => Group.zero);
      });
    }).not.toThrow();
  });

  it('invalid group element throws', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        Provable.witness(Group, () => Group({ x: 2, y: 2 }));
      });
    }).toThrow();
  });
});

describe('add', () => {
  it('g+g does not throw', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(Group, () => g);
        const y = Provable.witness(Group, () => g);
        x.add(y);
      });
    }).not.toThrow();
  });

  it('g+zero = g', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(Group, () => g);
        const zero = Provable.witness(Group, () => Group.zero);
        x.add(zero).assertEquals(x);
      });
    }).not.toThrow();
  });

  it('zero+g = g', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(Group, () => g);
        const zero = Provable.witness(Group, () => Group.zero);
        zero.add(x).assertEquals(x);
      });
    }).not.toThrow();
  });

  it('g+(-g) = zero', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(Group, () => g);
        const zero = Provable.witness(Group, () => Group.zero);
        x.add(x.neg()).assertEquals(zero);
      });
    }).not.toThrow();
  });

  it('(-g)+g = zero', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(Group, () => g);
        const zero = Provable.witness(Group, () => Group.zero);
        x.neg().add(x).assertEquals(zero);
      });
    }).not.toThrow();
  });

  it('zero + zero = zero', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const zero = Provable.witness(Group, () => Group.zero);
        zero.add(zero).assertEquals(zero);
      });
    }).not.toThrow();
  });
});

describe('sub', () => {
  it('g-g does not throw', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        Provable.runAndCheck(() => {
          const x = Provable.witness(Group, () => g);
          const y = Provable.witness(Group, () => g);
          x.sub(y);
        });
      });
    }).not.toThrow();
  });

  it('g-zero = g', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(Group, () => g);
        const zero = Provable.witness(Group, () => Group.zero);
        x.sub(zero).assertEquals(x);
      });
    }).not.toThrow();
  });

  it('zero - g = -g', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(Group, () => g);
        const zero = Provable.witness(Group, () => Group.zero);
        zero.sub(x).assertEquals(x.neg());
      });
    }).not.toThrow();
  });

  it('zero - zero = zero', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const zero = Provable.witness(Group, () => Group.zero);
        zero.sub(zero).assertEquals(zero);
      });
    }).not.toThrow();
  });
});

describe('neg', () => {
  it('neg(g) not to throw', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(Group, () => g);
        x.neg();
      });
    }).not.toThrow();
  });

  it('neg(zero) = zero', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const zero = Provable.witness(Group, () => Group.zero);
        zero.neg().assertEquals(zero);
      });
    }).not.toThrow();
  });
});
```

```
describe('scale', () => {
  it('scaling with random Scalar does not throw', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(Group, () => g);
        x.scale(Scalar.random());
      });
    }).not.toThrow();
  });

  it('x*g+y*g = (x+y)*g', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Scalar.from(2);
        const y = Scalar.from(3);
        const left = g.scale(x).add(g.scale(y));
        const right = g.scale(x.add(y));
        left.assertEquals(right);
      });
    }).not.toThrow();
  });

  it('x*(y*g) = (x*y)*g', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Scalar.from(2);
        const y = Scalar.from(3);
        const left = g.scale(y).scale(x);
        const right = g.scale(y.mul(x));
        left.assertEquals(right);
      });
    }).not.toThrow();
  });
});

describe('equals', () => {
  it('should equal true with same group', () => {
    Provable.runAndCheck(() => {
      const x = Provable.witness(Group, () => Group.generator);
      let isEqual = x.equals(Group.generator);
      Provable.asProver(() => {
        expect(isEqual.toBoolean()).toEqual(true);
      });
    });
  });

  it('should equal false with different group', () => {
    Provable.runAndCheck(() => {
      const x = Provable.witness(Group, () => Group.generator);
      let isEqual = x.equals(g);
      Provable.asProver(() => {
        expect(isEqual.toBoolean()).toEqual(false);
      });
    });
  });
});

describe('assertEquals', () => {
  it('should not throw with same group', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(Group, () => Group.generator);
        x.assertEquals(Group.generator);
      });
    }).not.toThrow();
  });

  it('should throw with different group', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(Group, () => Group.generator);
        x.assertEquals(g);
      });
    }).toThrow();
  });
});

describe('toJSON', () => {
  it('fromJSON(g.toJSON) should be the same as g', () => {
    Provable.runAndCheck(() => {
      const x = Provable.witness(
        Group,
        () => Group.fromJSON(Group.generator.toJSON())!
      );
      Provable.asProver(() => {
        expect(x.equals(Group.generator).toBoolean()).toEqual(true);
      });
    });
  });
});
});

describe('Outside circuit', () => {
  describe('neg', () => {
    it('neg not to throw', () => {
      expect(() => {
        g.neg();
      }).not.toThrow();
    });

    it('zero.neg = zero', () => {
      expect(() => {
        const zero = Group.zero;
      zero.neg().assertEquals(zero);
    }).not.toThrow();
  });
});

describe('add', () => {
  it('(-1,2)+(-1,2) does not throw', () => {
    expect(() => {
      g.add(g);
    }).not.toThrow();
  });

  it('g + zero = g', () => {
    expect(() => {
      const zero = Group.zero;
      g.add(zero).assertEquals(g);
    }).not.toThrow();
  });

  it('zero + g = g', () => {
    expect(() => {
      const zero = Group.zero;
      zero.add(g).assertEquals(g);
    }).not.toThrow();
  });

  it('g + (-g) = zero', () => {
    expect(() => {
      const zero = Group.zero;
      g.add(g.neg()).assertEquals(zero);
    }).not.toThrow();
  });

  it('(-g) + g = zero', () => {
    expect(() => {
      const zero = Group.zero;
      g.neg().add(g).assertEquals(zero);
    }).not.toThrow();
  });

  it('zero + zero = zero', () => {
    expect(() => {
      const zero = Group.zero;
      zero.add(zero).assertEquals(zero);
    }).not.toThrow();
  });
});

describe('sub', () => {
  it('generator-(-1,2) does not throw', () => {
    expect(() => {
      Group.generator.sub(g);
    }).not.toThrow();
  });

  it('g - zero = g', () => {
    expect(() => {
      const zero = Group.zero;
      g.sub(zero).assertEquals(g);
    }).not.toThrow();
  });

  it('zero - g = -g', () => {
    expect(() => {
      const zero = Group.zero;
      zero.sub(g).assertEquals(g.neg());
    }).not.toThrow();
  });

  it('zero - zero = -zero', () => {
    expect(() => {
      const zero = Group.zero;
      zero.sub(zero).assertEquals(zero);
    }).not.toThrow();
  });
});

describe('scale', () => {
  it('scaling with random Scalar does not throw', () => {
    expect(() => {
      g.scale(Scalar.random());
    }).not.toThrow();
  });

  it('x*g+y*g = (x+y)*g', () => {
    const x = Scalar.from(2);
    const y = Scalar.from(3);
    const left = g.scale(x).add(g.scale(y));
    const right = g.scale(x.add(y));
    expect(left).toEqual(right);
  });

  it('x*(y*g) = (x*y)*g', () => {
    const x = Scalar.from(2);
    const y = Scalar.from(3);
    const left = g.scale(y).scale(x);
    const right = g.scale(y.mul(x));
    expect(left).toEqual(right);
  });
});

describe('equals', () => {
  it('should equal true with same group', () => {
    expect(g.equals(g)).toEqual(Bool(true));
```

```
      });

      it('should equal false with different group', () => {
        expect(g.equals(Group.generator)).toEqual(Bool(false));
      });
    });

    describe('toJSON', () => {
      it("fromJSON('1','1') should be the same as Group(1,1)", () => {
        const x = Group.fromJSON({ x: -1, y: 2 });
        expect(x).toEqual(g);
      });
    });
  });

  describe('Variable/Constant circuit equality ', () => {
    it('add', () => {
      Provable.runAndCheck(() => {
        let y = Provable.witness(Group, () => g).add(
          Provable.witness(Group, () => Group.generator)
        );
        let z = g.add(Group.generator);
        y.assertEquals(z);
      });
    });

    it('sub', () => {
      let y = Provable.witness(Group, () => g).sub(
        Provable.witness(Group, () => Group.generator)
      );
      let z = g.sub(Group.generator);
      y.assertEquals(z);
    });

    it('sub', () => {
      let y = Provable.witness(Group, () => g).assertEquals(
        Provable.witness(Group, () => g)
      );
      g.assertEquals(g);
    });
  });
});
```

</file>

<file>

## path: /src/lib/group.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/group.ts

```
import { Field, FieldVar, isField } from './field.js';
import { Scalar } from './scalar.js';
import { Snarky } from '../snarky.js';
import { Field as Fp } from '../provable/field-bigint.js';
import { Pallas } from '../bindings/crypto/elliptic_curve.js';
import { Provable } from './provable.js';
import { Bool } from './bool.js';

export { Group };

/**
 * An element of a Group.
 */
class Group {
  x: Field;
  y: Field;

  /**
   * The generator &nbsp;g&nbsp; of the Group.
   */
  static get generator() {
    return new Group({ x: Pallas.one.x, y: Pallas.one.y });
  }

  /**
   * Unique representation of the &nbsp;zero&nbsp; element of the Group (the identity element of addition in
this Group).
   *
   * **Note**: The &nbsp;zero&nbsp; element is represented as &nbsp;(0, 0)&nbsp;.
   *
   * &nbsp;&nbsp;&nbsp;typescript
   * // g + -g = 0
   * g.add(g.neg()).assertEquals(zero);
   * // g + 0 = g
   * g.add(zero).assertEquals(g);
   * &nbsp;&nbsp;&nbsp;
   */
  static get zero() {
    return new Group({ x: 0, y: 0 });
  }

  /**
   * Coerces anything group-like to a {@link Group}.
   */
  constructor({
    x,
    y,
  }: {
    x: FieldVar | Field | number | string | bigint;
    y: FieldVar | Field | number | string | bigint;
  }) {
```

```
  this.x = isField(x) ? x : new Field(x);
  this.y = isField(y) ? y : new Field(y);

  if (this.#isConstant()) {
    // we also check the zero element (0, 0) here
    if (this.x.equals(0).and(this.y.equals(0)).toBoolean()) return;

    const { add, mul, square } = Fp;

    let x_bigint = this.x.toBigInt();
    let y_bigint = this.y.toBigInt();

    let onCurve =
      add(mul(x_bigint, mul(x_bigint, x_bigint)), Pallas.b) ===
      square(y_bigint);

    if (!onCurve) {
      throw Error(
        &nbsp;(x: ${x_bigint}, y: ${y_bigint}) is not a valid group element&nbsp;
      );
    }
  }
}

// helpers
static #fromAffine({
  x,
  y,
  infinity,
}: {
  x: bigint;
  y: bigint;
  infinity: boolean;
}) {
  return infinity ? Group.zero : new Group({ x, y });
}

static #fromProjective({ x, y, z }: { x: bigint; y: bigint; z: bigint }) {
  return this.#fromAffine(Pallas.toAffine({ x, y, z }));
}

#toTuple(): [0, FieldVar, FieldVar] {
  return [0, this.x.value, this.y.value];
}

#isConstant() {
  return this.x.isConstant() && this.y.isConstant();
}

#toProjective() {
  return Pallas.fromAffine({
    x: this.x.toBigInt(),
    y: this.y.toBigInt(),
    infinity: false,
  });
}

/**
 * Checks if this element is the &nbsp;zero&nbsp; element &nbsp;{x: 0, y: 0}&nbsp;.
 */
isZero() {
  // only the zero element can have x = 0, there are no other (valid) group elements with x = 0
  return this.x.equals(0);
}

/**
 * Adds this {@link Group} element to another {@link Group} element.
 *
 * &nbsp;&nbsp;&nbsp;ts
 * let g1 = Group({ x: -1, y: 2})
 * let g2 = g1.add(g1)
 * &nbsp;&nbsp;&nbsp;
 */
add(g: Group) {
  if (this.#isConstant() && g.#isConstant()) {
    // we check if either operand is zero, because adding zero to g just results in g (and vise versa)
    if (this.isZero().toBoolean()) {
      return g;
    } else if (g.isZero().toBoolean()) {
      return this;
    } else {
      let g_proj = Pallas.add(this.#toProjective(), g.#toProjective());
      return Group.#fromProjective(g_proj);
    }
  } else {
    const { x: x1, y: y1 } = this;
    const { x: x2, y: y2 } = g;

    let zero = new Field(0);

    let same_x = Provable.witness(Field, () => x1.equals(x2).toField());

    let inf = Provable.witness(Bool, () =>
      x1.equals(x2).and(y1.equals(y2).not())
    );

    let inf_z = Provable.witness(Field, () => {
      if (y1.equals(y2).toBoolean()) return zero;
      else if (x1.equals(x2).toBoolean()) return y2.sub(y1).inv();
      else return zero;
    });

    let x21_inv = Provable.witness(Field, () => {
      if (x1.equals(x2).toBoolean()) return zero;
      else return x2.sub(x1).inv();
    });

    let s = Provable.witness(Field, () => {
      if (x1.equals(x2).toBoolean()) {
        let x1_squared = x1.square();
        return x1_squared.add(x1_squared).add(x1_squared).div(y1.add(y1));
      } else return y2.sub(y1).div(x2.sub(x1));
    });

    let x3 = Provable.witness(Field, () => {
      return s.square().sub(x1.add(x2));
    });

    let y3 = Provable.witness(Field, () => {
      return s.mul(x1.sub(x3)).sub(y1);
    });

    let [, x, y] = Snarky.gates.ecAdd(
      Group.from(x1.seal(), y1.seal()).#toTuple(),
      Group.from(x2.seal(), y2.seal()).#toTuple(),
      Group.from(x3, y3).#toTuple(),
      inf.toField().value,
      same_x.value,
      s.value,
      inf_z.value,
      x21_inv.value
    );

    // similarly to the constant implementation, we check if either operand is zero
    // and the implementation above (original OCaml implementation) returns something wild -> g + 0 != g
where it should be g + 0 = g
    let gIsZero = g.isZero();
    let onlyThisIsZero = this.isZero().and(gIsZero.not());
    let isNegation = inf;
    let isNormalAddition = gIsZero.or(onlyThisIsZero).or(isNegation).not();

    // note: gIsZero and isNegation are not mutually exclusive, but if both are true, we add 1*0 + 1*0 = 0
which is correct
    return Provable.switch(
      [gIsZero, onlyThisIsZero, isNegation, isNormalAddition],
      Group,
      [this, g, Group.zero, new Group({ x, y })]
    );
  }
 }

 /**
  * Subtracts another {@link Group} element from this one.
  */
 sub(g: Group) {
  return this.add(g.neg());
 }

 /**
  * Negates this {@link Group}. Under the hood, it simply negates the &nbsp;y&nbsp; coordinate and leaves
the &nbsp;x&nbsp; coordinate as is.
  */
 neg() {
   let { x, y } = this;

   return new Group({ x, y: y.neg() });
 }

 /**
  * Elliptic curve scalar multiplication. Scales the {@link Group} element &nbsp;n&nbsp;-times by itself,
where &nbsp;n&nbsp; is the {@link Scalar}.
  *
  * &nbsp;&nbsp;&nbsp;typescript
  * let s = Scalar(5);
  * let 5g = g.scale(s);
  * &nbsp;&nbsp;&nbsp;
  */
 scale(s: Scalar | number | bigint) {
   let scalar = Scalar.from(s);

   if (this.#isConstant() && scalar.isConstant()) {
     let g_proj = Pallas.scale(this.#toProjective(), scalar.toBigInt());
     return Group.#fromProjective(g_proj);
   } else {
     let [, ...bits] = scalar.value;
     bits.reverse();
     let [, x, y] = Snarky.group.scale(this.#toTuple(), [0, ...bits]);
     return new Group({ x, y });
   }
 }

 /**
  * Assert that this {@link Group} element equals another {@link Group} element.
  * Throws an error if the assertion fails.
  *
  * &nbsp;&nbsp;&nbsp;ts
  * g1.assertEquals(g2);
  * &nbsp;&nbsp;&nbsp;
  */
 assertEquals(g: Group, message?: string) {
   let { x: x1, y: y1 } = this;
   let { x: x2, y: y2 } = g;

   x1.assertEquals(x2, message);
   y1.assertEquals(y2, message);
 }

 /**
```

```
   * Check if this {@link Group} element equals another {@link Group} element.
   * Returns a {@link Bool}.
   *
   * &nbsp;&nbsp;&nbsp;ts
   * g1.equals(g1); // Bool(true)
   * &nbsp;&nbsp;&nbsp;
   */
  equals(g: Group) {
    let { x: x1, y: y1 } = this;
    let { x: x2, y: y2 } = g;

    return x1.equals(x2).and(y1.equals(y2));
  }

  /**
   * Serializes this {@link Group} element to a JSON object.
   *
   * This operation does NOT affect the circuit and can't be used to prove anything about the representation
of the element.
   */
  toJSON(): {
    x: string;
    y: string;
  } {
    return {
      x: this.x.toString(),
      y: this.y.toString(),
    };
  }

  /**
   * Part of the {@link Provable} interface.
   *
   * Returns an array containing this {@link Group} element as an array of {@link Field} elements.
   */
  toFields() {
    return [this.x, this.y];
  }

  /**
   * Coerces two x and y coordinates into a {@link Group} element.
   */
  static from(
    x: FieldVar | Field | number | string | bigint,
    y: FieldVar | Field | number | string | bigint
  ) {
    return new Group({ x, y });
  }

  /**
   * @deprecated Please use the method &nbsp;.add&nbsp; on the instance instead
   *
```

```
 * Adds a {@link Group} element to another one.
 */
static add(g1: Group, g2: Group) {
  return g1.add(g2);
}

/**
 * @deprecated Please use the method &nbsp;.sub&nbsp; on the instance instead
 *
 * Subtracts a {@link Group} element from another one.
 */
static sub(g1: Group, g2: Group) {
  return g1.sub(g2);
}

/**
 * @deprecated Please use the method &nbsp;.neg&nbsp; on the instance instead
 *
 * Negates a {@link Group} element. Under the hood, it simply negates the &nbsp;y&nbsp; coordinate and
leaves the &nbsp;x&nbsp; coordinate as is.
 *
 * &nbsp;&nbsp;&nbsp;typescript
 * let gNeg = Group.neg(g);
 * &nbsp;&nbsp;&nbsp;
 */
static neg(g: Group) {
  return g.neg();
}

/**
 * @deprecated Please use the method &nbsp;.scale&nbsp; on the instance instead
 *
 * Elliptic curve scalar multiplication. Scales a {@link Group} element &nbsp;n&nbsp;-times by itself, where
&nbsp;n&nbsp; is the {@link Scalar}.
 *
 * &nbsp;&nbsp;&nbsp;typescript
 * let s = Scalar(5);
 * let 5g = Group.scale(g, s);
 * &nbsp;&nbsp;&nbsp;
 */
static scale(g: Group, s: Scalar) {
  return g.scale(s);
}

/**
 * @deprecated Please use the method &nbsp;.assertEqual&nbsp; on the instance instead.
 *
 * Assert that two {@link Group} elements are equal to another.
 * Throws an error if the assertion fails.
 *
 * &nbsp;&nbsp;&nbsp;ts
 * Group.assertEquals(g1, g2);
 * &nbsp;&nbsp;&nbsp;
 */
static assertEqual(g1: Group, g2: Group) {
  g1.assertEquals(g2);
}

/**
 * @deprecated Please use the method &nbsp;.equals&nbsp; on the instance instead.
 *
 * Checks if a {@link Group} element is equal to another {@link Group} element.
 * Returns a {@link Bool}.
 *
 * &nbsp;&nbsp;&nbsp;ts
 * Group.equal(g1, g2); // Bool(true)
 * &nbsp;&nbsp;&nbsp;
 */
static equal(g1: Group, g2: Group) {
  return g1.equals(g2);
}

/**
 * Part of the {@link Provable} interface.
 *
 * Returns an array containing a {@link Group} element as an array of {@link Field} elements.
 */
static toFields(g: Group) {
  return g.toFields();
}

/**
 * Part of the {@link Provable} interface.
 *
 * Returns an empty array.
 */
static toAuxiliary(g?: Group) {
  return [];
}

/**
 * Part of the {@link Provable} interface.
 *
 * Deserializes a {@link Group} element from a list of field elements.
 */
static fromFields([x, y]: Field[]) {
  return new Group({ x, y });
}

/**
 * Part of the {@link Provable} interface.
 *
 * Returns 2.
 */
```

```
static sizeInFields() {
  return 2;
}

/**
 * Serializes a {@link Group} element to a JSON object.
 *
 * This operation does NOT affect the circuit and can't be used to prove anything about the representation
of the element.
 */
static toJSON(g: Group) {
  return g.toJSON();
}

/**
 * Deserializes a JSON-like structure to a {@link Group} element.
 *
 * This operation does NOT affect the circuit and can't be used to prove anything about the representation
of the element.
 */
static fromJSON({
  x,
  y,
}: {
  x: string | number | bigint | Field | FieldVar;
  y: string | number | bigint | Field | FieldVar;
}) {
  return new Group({ x, y });
}

/**
 * Checks that a {@link Group} element is constraint properly by checking that the element is on the curve.
 */
static check(g: Group) {
  try {
    const { x, y } = g;

    let x2 = x.square();
    let x3 = x2.mul(x);
    let ax = x.mul(Pallas.a); // this will obviously be 0, but just for the sake of correctness

    // we also check the zero element (0, 0) here
    let isZero = x.equals(0).and(y.equals(0));

    isZero.or(x3.add(ax).add(Pallas.b).equals(y.square())).assertTrue();
  } catch (error) {
    if (!(error instanceof Error)) return error;
    throw &nbsp;${&nbsp;Element (x: ${g.x}, y: ${g.y}) is not an element of the group.&nbsp;}\n${
      error.message
    }&nbsp;;
  }
}
```

```
}
```

</file>

<file>

## path: /src/lib/group.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/group.unit-test.ts

```
import { Group } from './core.js';
import { test, Random } from './testing/property.js';
import { Provable } from './provable.js';
import { Poseidon } from '../provable/poseidon-bigint.js';

console.log('group consistency tests');

// tests consistency between in- and out-circuit implementations
test(Random.field, Random.field, (a, b, assert) => {
  const {
    x: x1,
    y: { x0: y1 },
  } = Poseidon.hashToGroup([a])!;

  const {
    x: x2,
    y: { x0: y2 },
  } = Poseidon.hashToGroup([b])!;

  const zero = Group.zero;
  const g1 = Group.from(x1, y1);
  const g2 = Group.from(x2, y2);

  run(g1, g2, (x, y) => x.add(y), assert);
  run(g1.neg(), g2.neg(), (x, y) => x.add(y), assert);
  run(g1, g1.neg(), (x, y) => x.add(y), assert);
  run(g1, zero, (x, y) => x.add(y), assert);
  run(g1, zero.neg(), (x, y) => x.add(y), assert);
  run(g1.neg(), zero, (x, y) => x.add(y), assert);

  run(zero, zero, (x, y) => x.add(y), assert);
  run(zero, zero.neg(), (x, y) => x.add(y), assert);
  run(zero.neg(), zero, (x, y) => x.add(y), assert);
  run(zero.neg(), zero.neg(), (x, y) => x.add(y), assert);

  run(g1, g2, (x, y) => x.sub(y), assert);
  run(g1.neg(), g2.neg(), (x, y) => x.sub(y), assert);
  run(g1, g1.neg(), (x, y) => x.sub(y), assert);
  run(g1, zero, (x, y) => x.sub(y), assert);
  run(g1, zero.neg(), (x, y) => x.sub(y), assert);
  run(g1.neg(), zero, (x, y) => x.sub(y), assert);
  run(zero, zero, (x, y) => x.sub(y), assert);
  run(zero, zero.neg(), (x, y) => x.sub(y), assert);
  run(zero.neg(), zero, (x, y) => x.sub(y), assert);
  run(zero.neg(), zero.neg(), (x, y) => x.sub(y), assert);
});

function run(
  g1: Group,
  g2: Group,
  f: (g1: Group, g2: Group) => Group,
  assert: (b: boolean, message?: string | undefined) => void
) {
  let result_out_circuit = f(g1, g2);

  Provable.runAndCheck(() => {
    let result_in_circuit = f(
      Provable.witness(Group, () => g1),
      Provable.witness(Group, () => g2)
    );

    Provable.asProver(() => {
      assert(
        result_out_circuit.equals(result_in_circuit).toBoolean(),
        &nbsp;Result for x does not match. g1: ${JSON.stringify(
          g1
        )}, g2: ${JSON.stringify(g2)}&nbsp;
      );
    });
  });
}
```

</file>

<file>

## path: /src/lib/hash-generic.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/hash-generic.ts

```
import { GenericField } from '../bindings/lib/generic.js';
import { prefixToField } from '../bindings/lib/binable.js';

export { createHashHelpers, HashHelpers };

type Hash<Field> = {
  initialState(): Field[];
  update(state: Field[], input: Field[]): Field[];
};

type HashHelpers<Field> = ReturnType<typeof createHashHelpers<Field>>;

function createHashHelpers<Field>(
  Field: GenericField<Field>,
  Hash: Hash<Field>
) {
  function salt(prefix: string) {
    return Hash.update(Hash.initialState(), [prefixToField(Field, prefix)]);
  }
  function emptyHashWithPrefix(prefix: string) {
    return salt(prefix)[0];
  }
  function hashWithPrefix(prefix: string, input: Field[]) {
    let init = salt(prefix);
    return Hash.update(init, input)[0];
  }
  return { salt, emptyHashWithPrefix, hashWithPrefix };
}
```

</file>

<file>

## path: /src/lib/hash-input.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/hash-input.unit-test.ts

```
import {
  isReady,
  AccountUpdate,
  Types,
  Permissions,
  shutdown,
  ProvableExtended,
} from '../index.js';
import { expect } from 'expect';
import { jsLayout } from '../bindings/mina-transaction/gen/js-layout.js';
import {
  Json,
  provableFromLayout,
} from '../bindings/mina-transaction/gen/transaction.js';
```

```
import { packToFields } from './hash.js';
import { Random, test } from './testing/property.js';
import { MlHashInput } from './ml/conversion.js';
import { MlFieldConstArray } from './ml/fields.js';
import { Test } from '../snarky.js';

let { hashInputFromJson } = Test;

await isReady;

// types
type Body = Types.AccountUpdate['body'];
type Update = Body['update'];
type Timing = Update['timing']['value'];
type AccountPrecondition = Body['preconditions']['account'];
type NetworkPrecondition = Body['preconditions']['network'];

// provables
let bodyLayout = jsLayout.AccountUpdate.entries.body;
let Timing = provableFromLayout<Timing, any>(
  bodyLayout.entries.update.entries.timing.inner as any
);
let Permissions_ = provableFromLayout<Permissions, any>(
  bodyLayout.entries.update.entries.permissions.inner as any
);
let Update = provableFromLayout<Update, any>(bodyLayout.entries.update as any);
let AccountPrecondition = provableFromLayout<AccountPrecondition, any>(
  bodyLayout.entries.preconditions.entries.account as any
);
let NetworkPrecondition = provableFromLayout<NetworkPrecondition, any>(
  bodyLayout.entries.preconditions.entries.network as any
);
let Body = provableFromLayout<Body, any>(bodyLayout as any);

// test with random account udpates
test(Random.json.accountUpdate, (accountUpdateJson) => {
  fixVerificationKey(accountUpdateJson);
  let accountUpdate = AccountUpdate.fromJSON(accountUpdateJson);

  // timing
  let timing = accountUpdate.body.update.timing.value;
  testInput(Timing, hashInputFromJson.timing, timing);

  // permissions
  let permissions = accountUpdate.body.update.permissions.value;
  testInput(Permissions_, hashInputFromJson.permissions, permissions);

  // update
  // TODO non ascii strings in zkapp uri and token symbol fail
  let update = accountUpdate.body.update;
  testInput(Update, hashInputFromJson.update, update);
  // account precondition
  let account = accountUpdate.body.preconditions.account;
  testInput(
    AccountPrecondition,
    hashInputFromJson.accountPrecondition,
    account
  );

  // network precondition
  let network = accountUpdate.body.preconditions.network;
  testInput(
    NetworkPrecondition,
    hashInputFromJson.networkPrecondition,
    network
  );

  // body
  let body = accountUpdate.body;
  testInput(Body, hashInputFromJson.body, body);

  // accountUpdate (should be same as body)
  testInput(
    Types.AccountUpdate,
    (accountUpdateJson) =>
      hashInputFromJson.body(
        JSON.stringify(JSON.parse(accountUpdateJson).body)
      ),
    accountUpdate
  );
});

console.log('all hash inputs are consistent!      ');
shutdown();

function testInput<T, TJson>(
  Module: ProvableExtended<T, TJson>,
  toInputOcaml: (json: string) => MlHashInput,
  value: T
) {
  let json = Module.toJSON(value);
  // console.log('json', json);
  let input1 = MlHashInput.from(toInputOcaml(JSON.stringify(json)));
  let input2 = Module.toInput(value);
  let input1Json = JSON.stringify(input1);
  let input2Json = JSON.stringify(input2);
  // console.log('o1js', input2Json);
  // console.log();
  // console.log('protocol', input1Json);
  let ok1 = input1Json === input2Json;
  expect(input2Json).toEqual(input1Json);
  // console.log('ok?', ok1);
  let fields1 = MlFieldConstArray.from(
    hashInputFromJson.packInput(MlHashInput.to(input1))
  );
  let fields2 = packToFields(input2);
  let ok2 = JSON.stringify(fields1) === JSON.stringify(fields2);
  // console.log('packed ok?', ok2);
  // console.log();
  if (!ok1 || !ok2) {
    throw Error('inconsistent toInput');
  }
}

function fixVerificationKey(accountUpdate: Json.AccountUpdate) {
  // TODO we set vk to null since we can't generate a valid random one
  accountUpdate.body.update.verificationKey = null;
}
```

</file>

<file>

## path: /src/lib/hash.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/hash.ts

```
import { HashInput, ProvableExtended, Struct } from './circuit_value.js';
import { Snarky } from '../snarky.js';
import { Field } from './core.js';
import { createHashHelpers } from './hash-generic.js';
import { Provable } from './provable.js';
import { MlFieldArray } from './ml/fields.js';
import { Poseidon as PoseidonBigint } from '../bindings/crypto/poseidon.js';
import { assert } from './errors.js';

// external API
export { Poseidon, TokenSymbol };

// internal API
export {
  HashInput,
  Hash,
  emptyHashWithPrefix,
  hashWithPrefix,
  salt,
  packToFields,
  emptyReceiptChainHash,
  hashConstant,
};

class Sponge {
  private sponge: unknown;
  constructor() {
    let isChecked = Provable.inCheckedComputation();
    this.sponge = Snarky.poseidon.sponge.create(isChecked);
  }

  absorb(x: Field) {
    Snarky.poseidon.sponge.absorb(this.sponge, x.value);
  }

  squeeze(): Field {
    return Field(Snarky.poseidon.sponge.squeeze(this.sponge));
  }
}

const Poseidon = {
  hash(input: Field[]) {
    if (isConstant(input)) {
      return Field(PoseidonBigint.hash(toBigints(input)));
    }
    return Poseidon.update(this.initialState(), input)[0];
  },

  update(state: [Field, Field, Field], input: Field[]) {
    if (isConstant(state) && isConstant(input)) {
      let newState = PoseidonBigint.update(toBigints(state), toBigints(input));
      return newState.map(Field);
    }

    let newState = Snarky.poseidon.update(
      MlFieldArray.to(state),
      MlFieldArray.to(input)
    );
    return MlFieldArray.from(newState) as [Field, Field, Field];
  },

  hashToGroup(input: Field[]) {
    if (isConstant(input)) {
      let result = PoseidonBigint.hashToGroup(toBigints(input));
      assert(result !== undefined, 'hashToGroup works on all inputs');
      let { x, y } = result;
      return {
        x: Field(x),
        y: { x0: Field(y.x0), x1: Field(y.x1) },
      };
    }

    // y = sqrt(y^2)
    let [, xv, yv] = Snarky.poseidon.hashToGroup(MlFieldArray.to(input));

    let x = Field(xv);
    let y = Field(yv);
    let x0 = Provable.witness(Field, () => {
      // the even root of y^2 will become x0, so the APIs are uniform
      let isEven = y.toBigInt() % 2n === 0n;

      // we just change the order so the even root is x0
      // y.mul(-1); is the second root of sqrt(y^2)
      return isEven ? y : y.mul(-1);
    });

    let x1 = x0.mul(-1);

    // we check that either x0 or x1 match the original root y
    y.equals(x0).or(y.equals(x1)).assertTrue();

    return { x, y: { x0, x1 } };
  },

  initialState(): [Field, Field, Field] {
    return [Field(0), Field(0), Field(0)];
  },

  Sponge,
};

function hashConstant(input: Field[]) {
  return Field(PoseidonBigint.hash(toBigints(input)));
}

const Hash = createHashHelpers(Field, Poseidon);
let { salt, emptyHashWithPrefix, hashWithPrefix } = Hash;

// same as Random_oracle.prefix_to_field in OCaml
function prefixToField(prefix: string) {
  if (prefix.length * 8 >= 255) throw Error('prefix too long');
  let bits = [...prefix]
    .map((char) => {
      // convert char to 8 bits
      let bits = [];
      for (let j = 0, c = char.charCodeAt(0); j < 8; j++, c >>= 1) {
        bits.push(!!(c & 1));
      }
      return bits;
    })
    .flat();
  return Field.fromBits(bits);
}

/**
 * Convert the {fields, packed} hash input representation to a list of field elements
 * Random_oracle_input.Chunked.pack_to_fields
 */
function packToFields({ fields = [], packed = [] }: HashInput) {
  if (packed.length === 0) return fields;
  let packedBits = [];
  let currentPackedField = Field(0);
  let currentSize = 0;
  for (let [field, size] of packed) {
    currentSize += size;
    if (currentSize < 255) {
      currentPackedField = currentPackedField
        .mul(Field(1n << BigInt(size)))
        .add(field);
    } else {
      packedBits.push(currentPackedField);
      currentSize = size;
      currentPackedField = field;
    }
  }
  packedBits.push(currentPackedField);
  return fields.concat(packedBits);
}

const TokenSymbolPure: ProvableExtended<
  { symbol: string; field: Field },
  string
> = {
  toFields({ field }) {
    return [field];
  },
  toAuxiliary(value) {
    return [value?.symbol ?? ''];
  },
  fromFields([field], [symbol]) {
    return { symbol, field };
  },
  sizeInFields() {
    return 1;
  },
  check({ field }: TokenSymbol) {
    let actual = field.rangeCheckHelper(48);
    actual.assertEquals(field);
  },
  toJSON({ symbol }) {
    return symbol;
  },
  fromJSON(symbol: string) {
    let field = prefixToField(symbol);
    return { symbol, field };
  },
  toInput({ field }) {
    return { packed: [[field, 48]] };
  },
};
class TokenSymbol extends Struct(TokenSymbolPure) {
  static get empty() {
    return { symbol: '', field: Field(0) };
  }

  static from(symbol: string): TokenSymbol {
    let bytesLength = new TextEncoder().encode(symbol).length;
    if (bytesLength > 6)
      throw Error(
        &nbsp;Token symbol ${symbol} should be a maximum of 6 bytes, but is ${bytesLength}&nbsp;
      );
    let field = prefixToField(symbol);
    return { symbol, field };
  }
}

function emptyReceiptChainHash() {
  return emptyHashWithPrefix('CodaReceiptEmpty');
}

function isConstant(fields: Field[]) {
  return fields.every((x) => x.isConstant());
}
function toBigints(fields: Field[]) {
  return fields.map((x) => x.toBigInt());
}
```

</file>

<file>

## path: /src/lib/int.test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/int.test.ts

```
import {
  isReady,
  Provable,
  shutdown,
  Int64,
  UInt64,
  UInt32,
  Field,
  Bool,
  Sign,
} from 'o1js';

describe('int', () => {
  beforeAll(async () => {
    await isReady;
  });

```

```
afterAll(async () => {
  // Use a timeout to defer the execution of &nbsp;shutdown()&nbsp; until Jest processes all tests.
  // &nbsp;shutdown()&nbsp; exits the process when it's done cleanup so we want to delay it's execution
until Jest is done
  setTimeout(async () => {
    await shutdown();
  }, 0);
});

const NUMBERMAX = 2 ** 53 - 1; //  JavaScript numbers can only safely store integers in the range -(2^53
– 1) to 2^53 – 1

describe('Int64', () => {
  describe('toString', () => {
    it('should be the same as Field(0)', async () => {
      const int = new Int64(UInt64.zero, Sign.one);
      const field = Field(0);
      expect(int.toString()).toEqual(field.toString());
    });

    it('should be -1', async () => {
      const int = new Int64(UInt64.one).neg();
      expect(int.toString()).toEqual('-1');
    });

    it('should be the same as 2^53-1', async () => {
      const int = Int64.fromField(Field(String(NUMBERMAX)));
      const field = Field(String(NUMBERMAX));
      expect(int.toString()).toEqual(field.toString());
    });
  });

  describe('zero', () => {
    it('should be the same as Field(0)', async () => {
      expect(Int64.zero.magnitude.value).toEqual(Field(0));
    });
  });

  describe('fromUnsigned', () => {
    it('should be the same as UInt64.zero', async () => {
      expect(new Int64(UInt64.zero, Sign.one)).toEqual(
        Int64.fromUnsigned(UInt64.zero)
      );
    });

    it('should be the same as UInt64.MAXINT', async () => {
      expect(Int64.from((1n << 64n) - 1n)).toEqual(
        Int64.fromUnsigned(UInt64.MAXINT())
      );
    });
  });
```

```
describe('neg', () => {
  it('neg(1)=-1', () => {
    const int = Int64.one;
    expect(int.neg().toField()).toEqual(Field(-1));
  });
  it('neg(2^53-1)=-2^53-1', () => {
    const int = Int64.from(NUMBERMAX);
    expect(int.neg().toString()).toEqual(&nbsp;${-NUMBERMAX}&nbsp;);
  });
});

describe('add', () => {
  it('1+1=2', () => {
    expect(Int64.one.add(Int64.from('1')).toString()).toEqual('2');
  });

  it('5000+(-4000)=1000', () => {
    expect(
      Int64.from(5000)
        .add(Int64.fromField(Field(-4000)))
        .toString()
    ).toEqual('1000');
  });

  it('(MAXINT/2+MAXINT/2) adds to MAXINT', () => {
    const value = ((1n << 64n) - 2n) / 2n;
    expect(
      Int64.from(value).add(Int64.from(value)).add(Int64.one).toString()
    ).toEqual(UInt64.MAXINT().toString());
  });

  it('should throw on overflow', () => {
    expect(() => {
      Int64.from(1n << 64n);
    }).toThrow();
    expect(() => {
      Int64.from(-(1n << 64n));
    }).toThrow();
    expect(() => {
      Int64.from(100).add(1n << 64n);
    }).toThrow();
    expect(() => {
      Int64.from(100).sub('1180591620717411303424');
    }).toThrow();
    expect(() => {
      Int64.from(100).mul(UInt64.from(Field(1n << 100n)));
    }).toThrow();
  });

  // TODO - should we make these throw?
  // These are edge cases, where one of two inputs is out of the Int64 range,
  // but the result of an operation with a proper Int64 moves it into the range.
  // They would only get caught if we'd also check the range in the Int64 / UInt64 constructors,
  // which breaks out current practice of having a dumb constructor that only stores variables
  it.skip('operations should throw on overflow of any input', () => {
    expect(() => {
      new Int64(new UInt64(Field(1n << 64n))).sub(1);
    }).toThrow();
    expect(() => {
      new Int64(new UInt64(Field(-(1n << 64n)))).add(5);
    }).toThrow();
    expect(() => {
      Int64.from(20).sub(new UInt64(Field((1n << 64n) + 10n)));
    }).toThrow();
    expect(() => {
      Int64.from(6).add(new UInt64(Field(-(1n << 64n) - 5n)));
    }).toThrow();
  });

  it('should throw on overflow addition', () => {
    expect(() => {
      Int64.from((1n << 64n) - 1n).add(1);
    }).toThrow();
    expect(() => {
      Int64.one.add((1n << 64n) - 1n);
    }).toThrow();
  });
  it('should not throw on non-overflowing addition', () => {
    expect(() => {
      Int64.from((1n << 64n) - 1n).add(Int64.zero);
    }).not.toThrow();
  });
});

describe('sub', () => {
  it('1-1=0', () => {
    expect(Int64.one.sub(Int64.from(1)).toString()).toEqual('0');
  });

  it('10000-5000=5000', () => {
    expect(
      Int64.fromField(Field(10000)).sub(Int64.from('5000')).toString()
    ).toEqual('5000');
  });

  it('0-1=-1', () => {
    expect(Int64.zero.sub(Int64.one).toString()).toEqual('-1');
  });

  it('(0-MAXINT) subs to -MAXINT', () => {
    expect(Int64.zero.sub(UInt64.MAXINT()).toString()).toEqual(
      '-' + UInt64.MAXINT().toString()
    );
  });
  });

  describe('toFields', () => {
    it('toFields(1) should be the same as [Field(1), Field(1)]', () => {
      expect(Int64.toFields(Int64.one)).toEqual([Field(1), Field(1)]);
    });

    it('toFields(2^53-1) should be the same as Field(2^53-1)', () => {
      expect(Int64.toFields(Int64.from(NUMBERMAX))).toEqual([
        Field(String(NUMBERMAX)),
        Field(1),
      ]);
    });
  });
  describe('fromFields', () => {
    it('fromFields([1, 1]) should be the same as Int64.one', () => {
      expect(Int64.fromFields([Field(1), Field(1)])).toEqual(Int64.one);
    });

    it('fromFields(2^53-1) should be the same as Field(2^53-1)', () => {
      expect(Int64.fromFields([Field(String(NUMBERMAX)), Field(1)])).toEqual(
        Int64.from(NUMBERMAX)
      );
    });
  });

  describe('mul / div / mod', () => {
    it('mul, div and mod work', () => {
      // 2 ** 6 === 64
      let x = Int64.fromField(Field(2))
        .mul(2)
        .mul('2')
        .mul(2n)
        .mul(UInt32.from(2))
        .mul(UInt64.from(2));
      expect(&nbsp;${x}&nbsp;).toBe('64');

      // 64 * (-64) === -64**2
      let y = Int64.from(-64);
      expect(&nbsp;${x.mul(y)}&nbsp;).toEqual(&nbsp;${-(64 ** 2)}&nbsp;);
      // (-64) // 64 === -1
      expect(y.div(x).toString()).toEqual('-1');
      // (-64) // 65 === 0
      expect(y.div(65).toString()).toEqual('0');
      // 64 % 3 === 1
      expect(x.mod(3).toString()).toEqual('1');
      // (-64) % 3 === 2
      expect(y.mod(3).toString()).toEqual('2');
    });
  });
});
```

```
describe('UInt64', () => {
  describe('Inside circuit', () => {
    describe('add', () => {
      it('1+1=2', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const x = Provable.witness(UInt64, () => new UInt64(Field(1)));
            const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
            x.add(y).assertEquals(new UInt64(Field(2)));
          });
        }).not.toThrow();
      });

      it('5000+5000=10000', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const x = Provable.witness(UInt64, () => new UInt64(Field(5000)));
            const y = Provable.witness(UInt64, () => new UInt64(Field(5000)));
            x.add(y).assertEquals(new UInt64(Field(10000)));
          });
        }).not.toThrow();
      });

      it('(MAXINT/2+MAXINT/2) adds to MAXINT', () => {
        const n = Field((((1n << 64n) - 2n) / 2n).toString());
        expect(() => {
          Provable.runAndCheck(() => {
            const x = Provable.witness(UInt64, () => new UInt64(n));
            const y = Provable.witness(UInt64, () => new UInt64(n));
            x.add(y).add(1).assertEquals(UInt64.MAXINT());
          });
        }).not.toThrow();
      });

      it('should throw on overflow addition', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const x = Provable.witness(UInt64, () => UInt64.MAXINT());
            const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
            x.add(y);
          });
        }).toThrow();
      });
    });

    describe('sub', () => {
      it('1-1=0', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const x = Provable.witness(UInt64, () => new UInt64(Field(1)));
            const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
            x.sub(y).assertEquals(new UInt64(Field(0)));
      });
    }).not.toThrow();
  });

  it('10000-5000=5000', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(
          UInt64,
          () => new UInt64(Field(10000))
        );
        const y = Provable.witness(UInt64, () => new UInt64(Field(5000)));
        x.sub(y).assertEquals(new UInt64(Field(5000)));
      });
    }).not.toThrow();
  });

  it('should throw on sub if results in negative number', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(0)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
        x.sub(y);
      });
    }).toThrow();
  });
});

describe('mul', () => {
  it('1x2=2', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(1)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(2)));
        x.mul(y).assertEquals(new UInt64(Field(2)));
      });
    }).not.toThrow();
  });

  it('1x0=0', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(1)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(0)));
        x.mul(y).assertEquals(new UInt64(Field(0)));
      });
    }).not.toThrow();
  });

  it('1000x1000=1000000', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(1000)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(1000)));
        x.mul(y).assertEquals(new UInt64(Field(1000000)));
      });
    }).not.toThrow();
  });

  it('MAXINTx1=MAXINT', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => UInt64.MAXINT());
        const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
        x.mul(y).assertEquals(UInt64.MAXINT());
      });
    }).not.toThrow();
  });

  it('should throw on overflow multiplication', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => UInt64.MAXINT());
        const y = Provable.witness(UInt64, () => new UInt64(Field(2)));
        x.mul(y);
      });
    }).toThrow();
  });
});

describe('div', () => {
  it('2/1=2', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(2)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
        x.div(y).assertEquals(new UInt64(Field(2)));
      });
    }).not.toThrow();
  });

  it('0/1=0', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(0)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
        x.div(y).assertEquals(new UInt64(Field(0)));
      });
    }).not.toThrow();
  });

  it('2000/1000=2', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(2000)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(1000)));
        x.div(y).assertEquals(new UInt64(Field(2)));
      });
    }).not.toThrow();
  });

  it('MAXINT/1=MAXINT', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => UInt64.MAXINT());
        const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
        x.div(y).assertEquals(UInt64.MAXINT());
      });
    }).not.toThrow();
  });

  it('should throw on division by zero', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => UInt64.MAXINT());
        const y = Provable.witness(UInt64, () => new UInt64(Field(0)));
        x.div(y);
      });
    }).toThrow();
  });
});

describe('mod', () => {
  it('1%1=0', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(1)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
        x.mod(y).assertEquals(new UInt64(Field(0)));
      });
    }).not.toThrow();
  });

  it('500%32=20', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(500)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(32)));
        x.mod(y).assertEquals(new UInt64(Field(20)));
      });
    }).not.toThrow();
  });

  it('MAXINT%7=1', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => UInt64.MAXINT());
        const y = Provable.witness(UInt64, () => new UInt64(Field(7)));
        x.mod(y).assertEquals(new UInt64(Field(1)));
      });
    }).not.toThrow();
  });

  it('should throw on mod by zero', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => UInt64.MAXINT());
        const y = Provable.witness(UInt64, () => new UInt64(Field(0)));
        x.mod(y).assertEquals(new UInt64(Field(1)));
      });
    }).toThrow();
  });
});

describe('assertLt', () => {
  it('1<2=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(1)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(2)));
        x.assertLessThan(y);
      });
    }).not.toThrow();
  });

  it('1<1=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(1)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
        x.assertLessThan(y);
      });
    }).toThrow();
  });

  it('2<1=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(2)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
        x.assertLessThan(y);
      });
    }).toThrow();
  });

  it('1000<100000=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(1000)));
        const y = Provable.witness(
          UInt64,
          () => new UInt64(Field(100000))
        );
        x.assertLessThan(y);
      });
    }).not.toThrow();
  });

  it('100000<1000=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(
          UInt64,
          () => new UInt64(Field(100000))
        );
        const y = Provable.witness(UInt64, () => new UInt64(Field(1000)));
        x.assertLessThan(y);
      });
    }).toThrow();
  });

  it('MAXINT<MAXINT=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => UInt64.MAXINT());
        const y = Provable.witness(UInt64, () => UInt64.MAXINT());
        x.assertLessThan(y);
      });
    }).toThrow();
  });
});

describe('assertLte', () => {
  it('1<=1=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(1)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
        x.assertLessThanOrEqual(y);
      });
    }).not.toThrow();
  });

  it('2<=1=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(2)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
        x.assertLessThanOrEqual(y);
      });
    }).toThrow();
  });

  it('1000<=100000=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(1000)));
        const y = Provable.witness(
          UInt64,
          () => new UInt64(Field(100000))
        );
        x.assertLessThanOrEqual(y);
      });
    }).not.toThrow();
  });

  it('100000<=1000=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(
          UInt64,
          () => new UInt64(Field(100000))
        );
        const y = Provable.witness(UInt64, () => new UInt64(Field(1000)));
        x.assertLessThanOrEqual(y);
      });
    }).toThrow();
  });

  it('MAXINT<=MAXINT=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => UInt64.MAXINT());
        const y = Provable.witness(UInt64, () => UInt64.MAXINT());
        x.assertLessThanOrEqual(y);
      });
    }).not.toThrow();
  });
});

describe('assertGreaterThan', () => {
  it('2>1=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(2)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
        x.assertGreaterThan(y);
      });
    }).not.toThrow();
  });

  it('1>1=false', () => {
```

```
  expect(() => {
    Provable.runAndCheck(() => {
      const x = Provable.witness(UInt64, () => new UInt64(Field(1)));
      const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
      x.assertGreaterThan(y);
    });
  }).toThrow();
});

it('1>2=false', () => {
  expect(() => {
    Provable.runAndCheck(() => {
      const x = Provable.witness(UInt64, () => new UInt64(Field(1)));
      const y = Provable.witness(UInt64, () => new UInt64(Field(2)));
      x.assertGreaterThan(y);
    });
  }).toThrow();
});

it('100000>1000=true', () => {
  expect(() => {
    Provable.runAndCheck(() => {
      const x = Provable.witness(
        UInt64,
        () => new UInt64(Field(100000))
      );
      const y = Provable.witness(UInt64, () => new UInt64(Field(1000)));
      x.assertGreaterThan(y);
    });
  }).not.toThrow();
});

it('1000>100000=false', () => {
  expect(() => {
    Provable.runAndCheck(() => {
      const x = Provable.witness(UInt64, () => new UInt64(Field(1000)));
      const y = Provable.witness(
        UInt64,
        () => new UInt64(Field(100000))
      );
      x.assertGreaterThan(y);
    });
  }).toThrow();
});

it('MAXINT>MAXINT=false', () => {
  expect(() => {
    Provable.runAndCheck(() => {
      const x = Provable.witness(UInt64, () => UInt64.MAXINT());
      const y = Provable.witness(UInt64, () => UInt64.MAXINT());
      x.assertGreaterThan(y);
    });
    }).toThrow();
  });
});

describe('assertGreaterThanOrEqual', () => {
  it('1<=1=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(1)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
        x.assertGreaterThanOrEqual(y);
      });
    }).not.toThrow();
  });

  it('1>=2=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(1)));
        const y = Provable.witness(UInt64, () => new UInt64(Field(2)));
        x.assertGreaterThanOrEqual(y);
      });
    }).toThrow();
  });

  it('100000>=1000=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(
          UInt64,
          () => new UInt64(Field(100000))
        );
        const y = Provable.witness(UInt64, () => new UInt64(Field(1000)));
        x.assertGreaterThanOrEqual(y);
      });
    }).not.toThrow();
  });

  it('1000>=100000=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt64, () => new UInt64(Field(1000)));
        const y = Provable.witness(
          UInt64,
          () => new UInt64(Field(100000))
        );
        x.assertGreaterThanOrEqual(y);
      });
    }).toThrow();
  });

  it('MAXINT>=MAXINT=true', () => {
      expect(() => {
        Provable.runAndCheck(() => {
          const x = Provable.witness(UInt64, () => UInt64.MAXINT());
          const y = Provable.witness(UInt64, () => UInt64.MAXINT());
          x.assertGreaterThanOrEqual(y);
        });
      }).not.toThrow();
    });
  });

  describe('from() ', () => {
    describe('fromNumber()', () => {
      it('should be the same as Field(1)', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const x = Provable.witness(UInt64, () => UInt64.from(1));
            const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
            x.assertEquals(y);
          });
        }).not.toThrow();
      });

      it('should be the same as 2^53-1', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const x = Provable.witness(UInt64, () =>
              UInt64.from(NUMBERMAX)
            );
            const y = Provable.witness(
              UInt64,
              () => new UInt64(Field(String(NUMBERMAX)))
            );
            x.assertEquals(y);
          });
        }).not.toThrow();
      });
    });
    describe('fromString()', () => {
      it('should be the same as Field(1)', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const x = Provable.witness(UInt64, () => UInt64.from('1'));
            const y = Provable.witness(UInt64, () => new UInt64(Field(1)));
            x.assertEquals(y);
          });
        }).not.toThrow();
      });

      it('should be the same as 2^53-1', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const x = Provable.witness(UInt64, () =>
          UInt64.from(String(NUMBERMAX))
        );
        const y = Provable.witness(
          UInt64,
          () => new UInt64(Field(String(NUMBERMAX)))
        );
        x.assertEquals(y);
      });
    }).not.toThrow();
  });
  });
 });
});

describe('Outside of circuit', () => {
 describe('add', () => {
  it('1+1=2', () => {
   expect(new UInt64(Field(1)).add(1).toString()).toEqual('2');
  });

  it('5000+5000=10000', () => {
   expect(new UInt64(Field(5000)).add(5000).toString()).toEqual('10000');
  });

  it('(MAXINT/2+MAXINT/2) adds to MAXINT', () => {
   const value = Field((((1n << 64n) - 2n) / 2n).toString());
   expect(
    new UInt64(value)
     .add(new UInt64(value))
     .add(new UInt64(Field(1)))
     .toString()
   ).toEqual(UInt64.MAXINT().toString());
  });

  it('should throw on overflow addition', () => {
   expect(() => {
    UInt64.MAXINT().add(1);
   }).toThrow();
  });
 });

 describe('sub', () => {
  it('1-1=0', () => {
   expect(new UInt64(Field(1)).sub(1).toString()).toEqual('0');
  });

  it('10000-5000=5000', () => {
   expect(new UInt64(Field(10000)).sub(5000).toString()).toEqual('5000');
  });

  it('should throw on sub if results in negative number', () => {
   expect(() => {
      UInt64.from(0).sub(1);
    }).toThrow();
  });
});

describe('mul', () => {
  it('1x2=2', () => {
    expect(new UInt64(Field(1)).mul(2).toString()).toEqual('2');
  });

  it('1x0=0', () => {
    expect(new UInt64(Field(1)).mul(0).toString()).toEqual('0');
  });

  it('1000x1000=1000000', () => {
    expect(new UInt64(Field(1000)).mul(1000).toString()).toEqual(
      '1000000'
    );
  });

  it('MAXINTx1=MAXINT', () => {
    expect(UInt64.MAXINT().mul(1).toString()).toEqual(
      UInt64.MAXINT().toString()
    );
  });

  it('should throw on overflow multiplication', () => {
    expect(() => {
      UInt64.MAXINT().mul(2);
    }).toThrow();
  });
});

describe('div', () => {
  it('2/1=2', () => {
    expect(new UInt64(Field(2)).div(1).toString()).toEqual('2');
  });

  it('0/1=0', () => {
    expect(new UInt64(Field(0)).div(1).toString()).toEqual('0');
  });

  it('2000/1000=2', () => {
    expect(new UInt64(Field(2000)).div(1000).toString()).toEqual('2');
  });

  it('MAXINT/1=MAXINT', () => {
    expect(UInt64.MAXINT().div(1).toString()).toEqual(
      UInt64.MAXINT().toString()
    );
  });
  it('should throw on division by zero', () => {
    expect(() => {
      UInt64.MAXINT().div(0);
    }).toThrow();
  });
});

describe('mod', () => {
  it('1%1=0', () => {
    expect(new UInt64(Field(1)).mod(1).toString()).toEqual('0');
  });

  it('500%32=20', () => {
    expect(new UInt64(Field(500)).mod(32).toString()).toEqual('20');
  });

  it('MAXINT%7=1', () => {
    expect(UInt64.MAXINT().mod(7).toString()).toEqual('1');
  });

  it('should throw on mod by zero', () => {
    expect(() => {
      UInt64.MAXINT().mod(0);
    }).toThrow();
  });
});

describe('lt', () => {
  it('1<2=true', () => {
    expect(new UInt64(Field(1)).lessThan(new UInt64(Field(2)))).toEqual(
      Bool(true)
    );
  });

  it('1<1=false', () => {
    expect(new UInt64(Field(1)).lessThan(new UInt64(Field(1)))).toEqual(
      Bool(false)
    );
  });

  it('2<1=false', () => {
    expect(new UInt64(Field(2)).lessThan(new UInt64(Field(1)))).toEqual(
      Bool(false)
    );
  });

  it('1000<100000=true', () => {
    expect(
      new UInt64(Field(1000)).lessThan(new UInt64(Field(100000)))
    ).toEqual(Bool(true));
  });
```

```
    it('100000<1000=false', () => {
      expect(
        new UInt64(Field(100000)).lessThan(new UInt64(Field(1000)))
      ).toEqual(Bool(false));
    });

    it('MAXINT<MAXINT=false', () => {
      expect(UInt64.MAXINT().lessThan(UInt64.MAXINT())).toEqual(
        Bool(false)
      );
    });
  });

  describe('lte', () => {
    it('1<=1=true', () => {
      expect(
        new UInt64(Field(1)).lessThanOrEqual(new UInt64(Field(1)))
      ).toEqual(Bool(true));
    });

    it('2<=1=false', () => {
      expect(
        new UInt64(Field(2)).lessThanOrEqual(new UInt64(Field(1)))
      ).toEqual(Bool(false));
    });

    it('1000<=100000=true', () => {
      expect(
        new UInt64(Field(1000)).lessThanOrEqual(new UInt64(Field(100000)))
      ).toEqual(Bool(true));
    });

    it('100000<=1000=false', () => {
      expect(
        new UInt64(Field(100000)).lessThanOrEqual(new UInt64(Field(1000)))
      ).toEqual(Bool(false));
    });

    it('MAXINT<=MAXINT=true', () => {
      expect(UInt64.MAXINT().lessThanOrEqual(UInt64.MAXINT())).toEqual(
        Bool(true)
      );
    });
  });

  describe('assertLessThanOrEqual', () => {
    it('1<=1=true', () => {
      expect(() => {
        new UInt64(Field(1)).assertLessThanOrEqual(new UInt64(Field(1)));
      }).not.toThrow();
    });
    it('2<=1=false', () => {
      expect(() => {
        new UInt64(Field(2)).assertLessThanOrEqual(new UInt64(Field(1)));
      }).toThrow();
    });

    it('1000<=100000=true', () => {
      expect(() => {
        new UInt64(Field(1000)).assertLessThanOrEqual(
          new UInt64(Field(100000))
        );
      }).not.toThrow();
    });

    it('100000<=1000=false', () => {
      expect(() => {
        new UInt64(Field(100000)).assertLessThanOrEqual(
          new UInt64(Field(1000))
        );
      }).toThrow();
    });

    it('MAXINT<=MAXINT=true', () => {
      expect(() => {
        UInt64.MAXINT().assertLessThanOrEqual(UInt64.MAXINT());
      }).not.toThrow();
    });
  });

  describe('greaterThan', () => {
    it('2>1=true', () => {
      expect(
        new UInt64(Field(2)).greaterThan(new UInt64(Field(1)))
      ).toEqual(Bool(true));
    });

    it('1>1=false', () => {
      expect(
        new UInt64(Field(1)).greaterThan(new UInt64(Field(1)))
      ).toEqual(Bool(false));
    });

    it('1>2=false', () => {
      expect(
        new UInt64(Field(1)).greaterThan(new UInt64(Field(2)))
      ).toEqual(Bool(false));
    });

    it('100000>1000=true', () => {
      expect(
        new UInt64(Field(100000)).greaterThan(new UInt64(Field(1000)))
      ).toEqual(Bool(true));
    });

    it('1000>100000=false', () => {
      expect(
        new UInt64(Field(1000)).greaterThan(new UInt64(Field(100000)))
      ).toEqual(Bool(false));
    });

    it('MAXINT>MAXINT=false', () => {
      expect(UInt64.MAXINT().greaterThan(UInt64.MAXINT())).toEqual(
        Bool(false)
      );
    });
  });

  describe('greaterThanOrEqual', () => {
    it('2>=1=true', () => {
      expect(
        new UInt64(Field(2)).greaterThanOrEqual(new UInt64(Field(1)))
      ).toEqual(Bool(true));
    });

    it('1>=1=true', () => {
      expect(
        new UInt64(Field(1)).greaterThanOrEqual(new UInt64(Field(1)))
      ).toEqual(Bool(true));
    });

    it('1>=2=false', () => {
      expect(
        new UInt64(Field(1)).greaterThanOrEqual(new UInt64(Field(2)))
      ).toEqual(Bool(false));
    });

    it('100000>=1000=true', () => {
      expect(
        new UInt64(Field(100000)).greaterThanOrEqual(
          new UInt64(Field(1000))
        )
      ).toEqual(Bool(true));
    });

    it('1000>=100000=false', () => {
      expect(
        new UInt64(Field(1000)).greaterThanOrEqual(
          new UInt64(Field(100000))
        )
      ).toEqual(Bool(false));
    });

    it('MAXINT>=MAXINT=true', () => {
      expect(UInt64.MAXINT().greaterThanOrEqual(UInt64.MAXINT())).toEqual(
        Bool(true)
      );
    });
  });

  describe('assertGreaterThan', () => {
    it('1>1=false', () => {
      expect(() => {
        new UInt64(Field(1)).assertGreaterThan(new UInt64(Field(1)));
      }).toThrow();
    });

    it('2>1=true', () => {
      expect(() => {
        new UInt64(Field(2)).assertGreaterThan(new UInt64(Field(1)));
      }).not.toThrow();
    });

    it('1000>100000=false', () => {
      expect(() => {
        new UInt64(Field(1000)).assertGreaterThan(
          new UInt64(Field(100000))
        );
      }).toThrow();
    });

    it('100000>1000=true', () => {
      expect(() => {
        new UInt64(Field(100000)).assertGreaterThan(
          new UInt64(Field(1000))
        );
      }).not.toThrow();
    });

    it('MAXINT>MAXINT=false', () => {
      expect(() => {
        UInt64.MAXINT().assertGreaterThan(UInt64.MAXINT());
      }).toThrow();
    });
  });

  describe('assertGreaterThanOrEqual', () => {
    it('1>=1=true', () => {
      expect(() => {
        new UInt64(Field(1)).assertGreaterThanOrEqual(new UInt64(Field(1)));
      }).not.toThrow();
    });

    it('2>=1=true', () => {
      expect(() => {
        new UInt64(Field(2)).assertGreaterThanOrEqual(new UInt64(Field(1)));
      }).not.toThrow();
    });

    it('1000>=100000=false', () => {
      expect(() => {
        new UInt64(Field(1000)).assertGreaterThanOrEqual(
          new UInt64(Field(100000))
        );
      }).toThrow();
    });

    it('100000>=1000=true', () => {
      expect(() => {
        new UInt64(Field(100000)).assertGreaterThanOrEqual(
          new UInt64(Field(1000))
        );
      }).not.toThrow();
    });

    it('MAXINT>=MAXINT=true', () => {
      expect(() => {
        UInt64.MAXINT().assertGreaterThanOrEqual(UInt64.MAXINT());
      }).not.toThrow();
    });
  });

  describe('toString()', () => {
    it('should be the same as Field(0)', async () => {
      const uint64 = new UInt64(Field(0));
      const field = Field(0);
      expect(uint64.toString()).toEqual(field.toString());
    });
    it('should be the same as 2^53-1', async () => {
      const uint64 = new UInt64(Field(String(NUMBERMAX)));
      const field = Field(String(NUMBERMAX));
      expect(uint64.toString()).toEqual(field.toString());
    });
  });

  describe('check()', () => {
    it('should pass checking a MAXINT', () => {
      expect(() => {
        UInt64.check(UInt64.MAXINT());
      }).not.toThrow();
    });

    it('should throw checking over MAXINT', () => {
      const aboveMax = new UInt64(Field((1n << 64n).toString())); // This number is defined in
UInt64.MAXINT()
      expect(() => {
        UInt64.check(aboveMax);
      }).toThrow();
    });
    });

    describe('from() ', () => {
      describe('fromNumber()', () => {
        it('should be the same as Field(1)', () => {
          const uint = UInt64.from(1);
          expect(uint.value).toEqual(new UInt64(Field(1)).value);
        });

        it('should be the same as 2^53-1', () => {
          const uint = UInt64.from(NUMBERMAX);
          expect(uint.value).toEqual(Field(String(NUMBERMAX)));
        });
      });
      describe('fromString()', () => {
        it('should be the same as Field(1)', () => {
          const uint = UInt64.from('1');
          expect(uint.value).toEqual(new UInt64(Field(1)).value);
        });

        it('should be the same as 2^53-1', () => {
          const uint = UInt64.from(String(NUMBERMAX));
          expect(uint.value).toEqual(Field(String(NUMBERMAX)));
        });
      });
    });
  });
});

describe('UInt32', () => {
  const NUMBERMAX = 2 ** 32 - 1;

  describe('Inside circuit', () => {
    describe('add', () => {
      it('1+1=2', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const x = Provable.witness(UInt32, () => new UInt32(Field(1)));
            const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
            x.add(y).assertEquals(new UInt32(Field(2)));
          });
        }).not.toThrow();
      });

      it('5000+5000=10000', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const x = Provable.witness(UInt32, () => new UInt32(Field(5000)));
            const y = Provable.witness(UInt32, () => new UInt32(Field(5000)));
            x.add(y).assertEquals(new UInt32(Field(10000)));
          });
        }).not.toThrow();
    });

    it('(MAXINT/2+MAXINT/2) adds to MAXINT', () => {
      const n = Field((((1n << 32n) - 2n) / 2n).toString());
      expect(() => {
        Provable.runAndCheck(() => {
          const x = Provable.witness(UInt32, () => new UInt32(n));
          const y = Provable.witness(UInt32, () => new UInt32(n));
          x.add(y).add(1).assertEquals(UInt32.MAXINT());
        });
      }).not.toThrow();
    });

    it('should throw on overflow addition', () => {
      expect(() => {
        Provable.runAndCheck(() => {
          const x = Provable.witness(UInt32, () => UInt32.MAXINT());
          const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
          x.add(y);
        });
      }).toThrow();
    });
  });

  describe('sub', () => {
    it('1-1=0', () => {
      expect(() => {
        Provable.runAndCheck(() => {
          const x = Provable.witness(UInt32, () => new UInt32(Field(1)));
          const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
          x.sub(y).assertEquals(new UInt32(Field(0)));
        });
      }).not.toThrow();
    });

    it('10000-5000=5000', () => {
      expect(() => {
        Provable.runAndCheck(() => {
          const x = Provable.witness(
            UInt32,
            () => new UInt32(Field(10000))
          );
          const y = Provable.witness(UInt32, () => new UInt32(Field(5000)));
          x.sub(y).assertEquals(new UInt32(Field(5000)));
        });
      }).not.toThrow();
    });

    it('should throw on sub if results in negative number', () => {
      expect(() => {
        Provable.runAndCheck(() => {
          const x = Provable.witness(UInt32, () => new UInt32(Field(0)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
        x.sub(y);
      });
    }).toThrow();
  });
});

describe('mul', () => {
  it('1x2=2', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(1)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(2)));
        x.mul(y).assertEquals(new UInt32(Field(2)));
      });
    }).not.toThrow();
  });

  it('1x0=0', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(1)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(0)));
        x.mul(y).assertEquals(new UInt32(Field(0)));
      });
    }).not.toThrow();
  });

  it('1000x1000=1000000', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(1000)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(1000)));
        x.mul(y).assertEquals(new UInt32(Field(1000000)));
      });
    }).not.toThrow();
  });

  it('MAXINTx1=MAXINT', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => UInt32.MAXINT());
        const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
        x.mul(y).assertEquals(UInt32.MAXINT());
      });
    }).not.toThrow();
  });

  it('should throw on overflow multiplication', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => UInt32.MAXINT());
        const y = Provable.witness(UInt32, () => new UInt32(Field(2)));
        x.mul(y);
      });
    }).toThrow();
  });
});

describe('div', () => {
  it('2/1=2', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(2)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
        x.div(y).assertEquals(new UInt32(Field(2)));
      });
    }).not.toThrow();
  });

  it('0/1=0', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(0)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
        x.div(y).assertEquals(new UInt32(Field(0)));
      });
    }).not.toThrow();
  });

  it('2000/1000=2', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(2000)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(1000)));
        x.div(y).assertEquals(new UInt32(Field(2)));
      });
    }).not.toThrow();
  });

  it('MAXINT/1=MAXINT', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => UInt32.MAXINT());
        const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
        x.div(y).assertEquals(UInt32.MAXINT());
      });
    }).not.toThrow();
  });

  it('should throw on division by zero', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => UInt32.MAXINT());
        const y = Provable.witness(UInt32, () => new UInt32(Field(0)));
        x.div(y);
      });
    }).toThrow();
  });
});

describe('mod', () => {
  it('1%1=0', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(1)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
        x.mod(y).assertEquals(new UInt32(Field(0)));
      });
    }).not.toThrow();
  });

  it('500%32=20', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(500)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(32)));
        x.mod(y).assertEquals(new UInt32(Field(20)));
      });
    }).not.toThrow();
  });

  it('MAXINT%7=3', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => UInt32.MAXINT());
        const y = Provable.witness(UInt32, () => new UInt32(Field(7)));
        x.mod(y).assertEquals(new UInt32(Field(3)));
      });
    }).not.toThrow();
  });

  it('should throw on mod by zero', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => UInt32.MAXINT());
        const y = Provable.witness(UInt32, () => new UInt32(Field(0)));
        x.mod(y).assertEquals(new UInt32(Field(1)));
      });
    }).toThrow();
  });
});

describe('assertLt', () => {
  it('1<2=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(1)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(2)));
        x.assertLessThan(y);
      });
    }).not.toThrow();
  });

  it('1<1=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(1)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
        x.assertLessThan(y);
      });
    }).toThrow();
  });

  it('2<1=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(2)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
        x.assertLessThan(y);
      });
    }).toThrow();
  });

  it('1000<100000=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(1000)));
        const y = Provable.witness(
          UInt32,
          () => new UInt32(Field(100000))
        );
        x.assertLessThan(y);
      });
    }).not.toThrow();
  });

  it('100000<1000=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(
          UInt32,
          () => new UInt32(Field(100000))
        );
        const y = Provable.witness(UInt32, () => new UInt32(Field(1000)));
        x.assertLessThan(y);
      });
    }).toThrow();
  });

  it('MAXINT<MAXINT=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => UInt32.MAXINT());
        const y = Provable.witness(UInt32, () => UInt32.MAXINT());
        x.assertLessThan(y);
      });
    }).toThrow();
  });
});

describe('assertLessThanOrEqual', () => {
  it('1<=1=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(1)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
        x.assertLessThanOrEqual(y);
      });
    }).not.toThrow();
  });

  it('2<=1=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(2)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
        x.assertLessThanOrEqual(y);
      });
    }).toThrow();
  });

  it('1000<=100000=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(1000)));
        const y = Provable.witness(
          UInt32,
          () => new UInt32(Field(100000))
        );
        x.assertLessThanOrEqual(y);
      });
    }).not.toThrow();
  });

  it('100000<=1000=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(
          UInt32,
          () => new UInt32(Field(100000))
        );
        const y = Provable.witness(UInt32, () => new UInt32(Field(1000)));
        x.assertLessThanOrEqual(y);
      });
    }).toThrow();
  });

  it('MAXINT<=MAXINT=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => UInt32.MAXINT());
        const y = Provable.witness(UInt32, () => UInt32.MAXINT());
        x.assertLessThanOrEqual(y);
      });
    }).not.toThrow();
  });
});

describe('assertGreaterThan', () => {
  it('2>1=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(2)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
        x.assertGreaterThan(y);
      });
    }).not.toThrow();
  });

  it('1>1=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(1)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
        x.assertGreaterThan(y);
      });
    }).toThrow();
  });

  it('1>2=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(1)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(2)));
        x.assertGreaterThan(y);
      });
    }).toThrow();
  });

  it('100000>1000=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(
          UInt32,
          () => new UInt32(Field(100000))
        );
        const y = Provable.witness(UInt32, () => new UInt32(Field(1000)));
        x.assertGreaterThan(y);
      });
    }).not.toThrow();
  });

  it('1000>100000=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(1000)));
        const y = Provable.witness(
          UInt32,
          () => new UInt32(Field(100000))
        );
        x.assertGreaterThan(y);
      });
    }).toThrow();
  });

  it('MAXINT>MAXINT=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => UInt32.MAXINT());
        const y = Provable.witness(UInt32, () => UInt32.MAXINT());
        x.assertGreaterThan(y);
      });
    }).toThrow();
  });
});

describe('assertGreaterThanOrEqual', () => {
  it('1<=1=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(1)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
        x.assertGreaterThanOrEqual(y);
      });
    }).not.toThrow();
  });

  it('1>=2=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(1)));
        const y = Provable.witness(UInt32, () => new UInt32(Field(2)));
        x.assertGreaterThanOrEqual(y);
      });
    }).toThrow();
  });

  it('100000>=1000=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(
          UInt32,
          () => new UInt32(Field(100000))
        );
        const y = Provable.witness(UInt32, () => new UInt32(Field(1000)));
        x.assertGreaterThanOrEqual(y);
      });
    }).not.toThrow();
  });

  it('1000>=100000=false', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => new UInt32(Field(1000)));
        const y = Provable.witness(
          UInt32,
          () => new UInt32(Field(100000))
        );
        x.assertGreaterThanOrEqual(y);
      });
    }).toThrow();
  });

  it('MAXINT>=MAXINT=true', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(UInt32, () => UInt32.MAXINT());
        const y = Provable.witness(UInt32, () => UInt32.MAXINT());
        x.assertGreaterThanOrEqual(y);
      });
    }).not.toThrow();
  });
});

describe('from() ', () => {
  describe('fromNumber()', () => {
    it('should be the same as Field(1)', () => {
      expect(() => {
        Provable.runAndCheck(() => {
          const x = Provable.witness(UInt32, () => UInt32.from(1));
          const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
          x.assertEquals(y);
        });
      }).not.toThrow();
    });
```

```
      it('should be the same as 2^53-1', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const x = Provable.witness(UInt32, () =>
              UInt32.from(NUMBERMAX)
            );
            const y = Provable.witness(
              UInt32,
              () => new UInt32(Field(String(NUMBERMAX)))
            );
            x.assertEquals(y);
          });
        }).not.toThrow();
      });
    });
    describe('fromString()', () => {
      it('should be the same as Field(1)', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const x = Provable.witness(UInt32, () => UInt32.from('1'));
            const y = Provable.witness(UInt32, () => new UInt32(Field(1)));
            x.assertEquals(y);
          });
        }).not.toThrow();
      });

      it('should be the same as 2^53-1', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const x = Provable.witness(UInt32, () =>
              UInt32.from(String(NUMBERMAX))
            );
            const y = Provable.witness(
              UInt32,
              () => new UInt32(Field(String(NUMBERMAX)))
            );
            x.assertEquals(y);
          });
        }).not.toThrow();
      });
    });
  });
});

describe('Outside of circuit', () => {
  describe('add', () => {
    it('1+1=2', () => {
      expect(new UInt32(Field(1)).add(1).toString()).toEqual('2');
    });

    it('5000+5000=10000', () => {
```

```
    expect(new UInt32(Field(5000)).add(5000).toString()).toEqual('10000');
  });

  it('(MAXINT/2+MAXINT/2) adds to MAXINT', () => {
    const value = Field((((1n << 32n) - 2n) / 2n).toString());
    expect(
      new UInt32(value)
        .add(new UInt32(value))
        .add(new UInt32(Field(1)))
        .toString()
    ).toEqual(UInt32.MAXINT().toString());
  });

  it('should throw on overflow addition', () => {
    expect(() => {
      UInt32.MAXINT().add(1);
    }).toThrow();
  });
});

describe('sub', () => {
  it('1-1=0', () => {
    expect(new UInt32(Field(1)).sub(1).toString()).toEqual('0');
  });

  it('10000-5000=5000', () => {
    expect(new UInt32(Field(10000)).sub(5000).toString()).toEqual('5000');
  });

  it('should throw on sub if results in negative number', () => {
    expect(() => {
      UInt32.from(0).sub(1);
    }).toThrow();
  });
});

describe('mul', () => {
  it('1x2=2', () => {
    expect(new UInt32(Field(1)).mul(2).toString()).toEqual('2');
  });

  it('1x0=0', () => {
    expect(new UInt32(Field(1)).mul(0).toString()).toEqual('0');
  });

  it('1000x1000=1000000', () => {
    expect(new UInt32(Field(1000)).mul(1000).toString()).toEqual(
      '1000000'
    );
  });

  it('MAXINTx1=MAXINT', () => {
    expect(UInt32.MAXINT().mul(1).toString()).toEqual(
      UInt32.MAXINT().toString()
    );
  });

  it('should throw on overflow multiplication', () => {
    expect(() => {
      UInt32.MAXINT().mul(2);
    }).toThrow();
  });
});

describe('div', () => {
  it('2/1=2', () => {
    expect(new UInt32(Field(2)).div(1).toString()).toEqual('2');
  });

  it('0/1=0', () => {
    expect(new UInt32(Field(0)).div(1).toString()).toEqual('0');
  });

  it('2000/1000=2', () => {
    expect(new UInt32(Field(2000)).div(1000).toString()).toEqual('2');
  });

  it('MAXINT/1=MAXINT', () => {
    expect(UInt32.MAXINT().div(1).toString()).toEqual(
      UInt32.MAXINT().toString()
    );
  });

  it('should throw on division by zero', () => {
    expect(() => {
      UInt32.MAXINT().div(0);
    }).toThrow();
  });
});

describe('mod', () => {
  it('1%1=0', () => {
    expect(new UInt32(Field(1)).mod(1).toString()).toEqual('0');
  });

  it('500%32=20', () => {
    expect(new UInt32(Field(500)).mod(32).toString()).toEqual('20');
  });

  it('MAXINT%7=3', () => {
    expect(UInt32.MAXINT().mod(7).toString()).toEqual('3');
  });

  it('should throw on mod by zero', () => {
    expect(() => {
      UInt32.MAXINT().mod(0);
    }).toThrow();
  });
});

describe('lessThan', () => {
  it('1<2=true', () => {
    expect(new UInt32(Field(1)).lessThan(new UInt32(Field(2)))).toEqual(
      Bool(true)
    );
  });

  it('1<1=false', () => {
    expect(new UInt32(Field(1)).lessThan(new UInt32(Field(1)))).toEqual(
      Bool(false)
    );
  });

  it('2<1=false', () => {
    expect(new UInt32(Field(2)).lessThan(new UInt32(Field(1)))).toEqual(
      Bool(false)
    );
  });

  it('1000<100000=true', () => {
    expect(
      new UInt32(Field(1000)).lessThan(new UInt32(Field(100000)))
    ).toEqual(Bool(true));
  });

  it('100000<1000=false', () => {
    expect(
      new UInt32(Field(100000)).lessThan(new UInt32(Field(1000)))
    ).toEqual(Bool(false));
  });

  it('MAXINT<MAXINT=false', () => {
    expect(UInt32.MAXINT().lessThan(UInt32.MAXINT())).toEqual(
      Bool(false)
    );
  });
});

describe('lessThanOrEqual', () => {
  it('1<=1=true', () => {
    expect(
      new UInt32(Field(1)).lessThanOrEqual(new UInt32(Field(1)))
    ).toEqual(Bool(true));
  });

  it('2<=1=false', () => {
    expect(
      new UInt32(Field(2)).lessThanOrEqual(new UInt32(Field(1)))
    ).toEqual(Bool(false));
  });

  it('1000<=100000=true', () => {
    expect(
      new UInt32(Field(1000)).lessThanOrEqual(new UInt32(Field(100000)))
    ).toEqual(Bool(true));
  });

  it('100000<=1000=false', () => {
    expect(
      new UInt32(Field(100000)).lessThanOrEqual(new UInt32(Field(1000)))
    ).toEqual(Bool(false));
  });

  it('MAXINT<=MAXINT=true', () => {
    expect(UInt32.MAXINT().lessThanOrEqual(UInt32.MAXINT())).toEqual(
      Bool(true)
    );
  });
});

describe('assertLessThanOrEqual', () => {
  it('1<=1=true', () => {
    expect(() => {
      new UInt32(Field(1)).assertLessThanOrEqual(new UInt32(Field(1)));
    }).not.toThrow();
  });

  it('2<=1=false', () => {
    expect(() => {
      new UInt32(Field(2)).assertLessThanOrEqual(new UInt32(Field(1)));
    }).toThrow();
  });

  it('1000<=100000=true', () => {
    expect(() => {
      new UInt32(Field(1000)).assertLessThanOrEqual(
        new UInt32(Field(100000))
      );
    }).not.toThrow();
  });

  it('100000<=1000=false', () => {
    expect(() => {
      new UInt32(Field(100000)).assertLessThanOrEqual(
        new UInt32(Field(1000))
      );
    }).toThrow();
  });
  it('MAXINT<=MAXINT=true', () => {
    expect(() => {
      UInt32.MAXINT().assertLessThanOrEqual(UInt32.MAXINT());
    }).not.toThrow();
  });
});

describe('greaterThan', () => {
  it('2>1=true', () => {
    expect(
      new UInt32(Field(2)).greaterThan(new UInt32(Field(1)))
    ).toEqual(Bool(true));
  });

  it('1>1=false', () => {
    expect(
      new UInt32(Field(1)).greaterThan(new UInt32(Field(1)))
    ).toEqual(Bool(false));
  });

  it('1>2=false', () => {
    expect(
      new UInt32(Field(1)).greaterThan(new UInt32(Field(2)))
    ).toEqual(Bool(false));
  });

  it('100000>1000=true', () => {
    expect(
      new UInt32(Field(100000)).greaterThan(new UInt32(Field(1000)))
    ).toEqual(Bool(true));
  });

  it('1000>100000=false', () => {
    expect(
      new UInt32(Field(1000)).greaterThan(new UInt32(Field(100000)))
    ).toEqual(Bool(false));
  });

  it('MAXINT>MAXINT=false', () => {
    expect(UInt32.MAXINT().greaterThan(UInt32.MAXINT())).toEqual(
      Bool(false)
    );
  });
});

describe('assertGreaterThan', () => {
  it('1>1=false', () => {
    expect(() => {
      new UInt32(Field(1)).assertGreaterThan(new UInt32(Field(1)));
    }).toThrow();
  });
```

```
    it('2>1=true', () => {
      expect(() => {
        new UInt32(Field(2)).assertGreaterThan(new UInt32(Field(1)));
      }).not.toThrow();
    });

    it('1000>100000=false', () => {
      expect(() => {
        new UInt32(Field(1000)).assertGreaterThan(
          new UInt32(Field(100000))
        );
      }).toThrow();
    });

    it('100000>1000=true', () => {
      expect(() => {
        new UInt32(Field(100000)).assertGreaterThan(
          new UInt32(Field(1000))
        );
      }).not.toThrow();
    });

    it('MAXINT>MAXINT=false', () => {
      expect(() => {
        UInt32.MAXINT().assertGreaterThan(UInt32.MAXINT());
      }).toThrow();
    });
  });

  describe('greaterThanOrEqual', () => {
    it('2>=1=true', () => {
      expect(
        new UInt32(Field(2)).greaterThanOrEqual(new UInt32(Field(1)))
      ).toEqual(Bool(true));
    });

    it('1>=1=true', () => {
      expect(
        new UInt32(Field(1)).greaterThanOrEqual(new UInt32(Field(1)))
      ).toEqual(Bool(true));
    });

    it('1>=2=false', () => {
      expect(
        new UInt32(Field(1)).greaterThanOrEqual(new UInt32(Field(2)))
      ).toEqual(Bool(false));
    });

    it('100000>=1000=true', () => {
      expect(
        new UInt32(Field(100000)).greaterThanOrEqual(
          new UInt32(Field(1000))
        )
      ).toEqual(Bool(true));
    });

    it('1000>=100000=false', () => {
      expect(
        new UInt32(Field(1000)).greaterThanOrEqual(
          new UInt32(Field(100000))
        )
      ).toEqual(Bool(false));
    });

    it('MAXINT>=MAXINT=true', () => {
      expect(UInt32.MAXINT().greaterThanOrEqual(UInt32.MAXINT())).toEqual(
        Bool(true)
      );
    });
  });

  describe('assertGreaterThanOrEqual', () => {
    it('1>=1=true', () => {
      expect(() => {
        new UInt32(Field(1)).assertGreaterThanOrEqual(new UInt32(Field(1)));
      }).not.toThrow();
    });

    it('2>=1=true', () => {
      expect(() => {
        new UInt32(Field(2)).assertGreaterThanOrEqual(new UInt32(Field(1)));
      }).not.toThrow();
    });

    it('1000>=100000=false', () => {
      expect(() => {
        new UInt32(Field(1000)).assertGreaterThanOrEqual(
          new UInt32(Field(100000))
        );
      }).toThrow();
    });

    it('100000>=1000=true', () => {
      expect(() => {
        new UInt32(Field(100000)).assertGreaterThanOrEqual(
          new UInt32(Field(1000))
        );
      }).not.toThrow();
    });

    it('MAXINT>=MAXINT=true', () => {
      expect(() => {
        UInt32.MAXINT().assertGreaterThanOrEqual(UInt32.MAXINT());
      }).not.toThrow();
    });
  });

  describe('toString()', () => {
    it('should be the same as Field(0)', async () => {
      const x = new UInt32(Field(0));
      const y = Field(0);
      expect(x.toString()).toEqual(y.toString());
    });
    it('should be the same as 2^32-1', async () => {
      const x = new UInt32(Field(String(NUMBERMAX)));
      const y = Field(String(NUMBERMAX));
      expect(x.toString()).toEqual(y.toString());
    });
  });

  describe('check()', () => {
    it('should pass checking a MAXINT', () => {
      expect(() => {
        UInt32.check(UInt32.MAXINT());
      }).not.toThrow();
    });

    it('should throw checking over MAXINT', () => {
      const x = new UInt32(Field((1n << 32n).toString())); // This number is defined in UInt32.MAXINT()
      expect(() => {
        UInt32.check(x);
      }).toThrow();
    });
  });

  describe('from() ', () => {
    describe('fromNumber()', () => {
      it('should be the same as Field(1)', () => {
        const x = UInt32.from(1);
        expect(x.value).toEqual(new UInt32(Field(1)).value);
      });

      it('should be the same as 2^53-1', () => {
        const x = UInt32.from(NUMBERMAX);
        expect(x.value).toEqual(Field(String(NUMBERMAX)));
      });
    });
    describe('fromString()', () => {
      it('should be the same as Field(1)', () => {
        const x = UInt32.from('1');
        expect(x.value).toEqual(new UInt32(Field(1)).value);
      });

      it('should be the same as 2^53-1', () => {
        const x = UInt32.from(String(NUMBERMAX));
```

```
          expect(x.value).toEqual(Field(String(NUMBERMAX)));
        });
      });
    });
  });
 });
});
```

</file>

<file>

## path: /src/lib/int.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/int.ts

```
import { Field, Bool } from './core.js';
import { AnyConstructor, CircuitValue, prop } from './circuit_value.js';
import { Types } from '../bindings/mina-transaction/types.js';
import { HashInput } from './hash.js';
import { Provable } from './provable.js';

// external API
export { UInt32, UInt64, Int64, Sign };

/**
 * A 64 bit unsigned integer with values ranging from 0 to 18,446,744,073,709,551,615.
 */
class UInt64 extends CircuitValue {
  @prop value: Field;
  static NUM_BITS = 64;

  /**
   * Static method to create a {@link UInt64} with value &nbsp;0&nbsp;.
   */
  static get zero() {
    return new UInt64(Field(0));
  }
  /**
   * Static method to create a {@link UInt64} with value &nbsp;1&nbsp;.
   */
  static get one() {
    return new UInt64(Field(1));
  }
  /**
   * Turns the {@link UInt64} into a string.
   * @returns
   */
  toString() {
    return this.value.toString();
  }
```

```
/**
 * Turns the {@link UInt64} into a {@link BigInt}.
 * @returns
 */
toBigInt() {
  return this.value.toBigInt();
}

/**
 * Turns the {@link UInt64} into a {@link UInt32}, asserting that it fits in 32 bits.
 */
toUInt32() {
  let uint32 = new UInt32(this.value);
  UInt32.check(uint32);
  return uint32;
}

/**
 * Turns the {@link UInt64} into a {@link UInt32}, clamping to the 32 bits range if it's too large.
 * &nbsp;&nbsp;&nbsp;ts
 * UInt64.from(4294967296).toUInt32Clamped().toString(); // "4294967295"
 * &nbsp;&nbsp;&nbsp;
 */
toUInt32Clamped() {
  let max = (1n << 32n) - 1n;
  return Provable.if(
    this.greaterThan(UInt64.from(max)),
    UInt32.from(max),
    new UInt32(this.value)
  );
}

static check(x: UInt64) {
  let actual = x.value.rangeCheckHelper(64);
  actual.assertEquals(x.value);
}

static toInput(x: UInt64): HashInput {
  return { packed: [[x.value, 64]] };
}

/**
 * Encodes this structure into a JSON-like object.
 */
static toJSON(x: UInt64) {
  return x.value.toString();
}

/**
 * Decodes a JSON-like object into this structure.
 */
static fromJSON<T extends AnyConstructor>(x: string): InstanceType<T> {
  return this.from(x) as any;
}

private static checkConstant(x: Field) {
  if (!x.isConstant()) return x;
  let xBig = x.toBigInt();
  if (xBig < 0n || xBig >= 1n << BigInt(this.NUM_BITS)) {
    throw Error(
      &nbsp;UInt64: Expected number between 0 and 2^64 - 1, got ${xBig}&nbsp;
    );
  }
  return x;
}

// this checks the range if the argument is a constant
/**
 * Creates a new {@link UInt64}.
 */
static from(x: UInt64 | UInt32 | Field | number | string | bigint) {
  if (x instanceof UInt64 || x instanceof UInt32) x = x.value;
  return new this(this.checkConstant(Field(x)));
}

/**
 * Creates a {@link UInt64} with a value of 18,446,744,073,709,551,615.
 */
static MAXINT() {
  return new UInt64(Field((1n << 64n) - 1n));
}

/**
 * Integer division with remainder.
 *
 * &nbsp;x.divMod(y)&nbsp; returns the quotient and the remainder.
 */
divMod(y: UInt64 | number | string) {
  let x = this.value;
  let y_ = UInt64.from(y).value;

  if (this.value.isConstant() && y_.isConstant()) {
    let xn = x.toBigInt();
    let yn = y_.toBigInt();
    let q = xn / yn;
    let r = xn - q * yn;
    return {
      quotient: new UInt64(Field(q)),
      rest: new UInt64(Field(r)),
    };
  }

  y_ = y_.seal();
  let q = Provable.witness(
    Field,
    () => new Field(x.toBigInt() / y_.toBigInt())
  );

  q.rangeCheckHelper(UInt64.NUM_BITS).assertEquals(q);

  // TODO: Could be a bit more efficient
  let r = x.sub(q.mul(y_)).seal();
  r.rangeCheckHelper(UInt64.NUM_BITS).assertEquals(r);

  let r_ = new UInt64(r);
  let q_ = new UInt64(q);

  r_.assertLessThan(new UInt64(y_));

  return { quotient: q_, rest: r_ };
}

/**
 * Integer division.
 *
 * &nbsp;x.div(y)&nbsp; returns the floor of &nbsp;x / y&nbsp;, that is, the greatest
 * &nbsp;z&nbsp; such that &nbsp;z * y <= x&nbsp;.
 *
 */
div(y: UInt64 | number) {
  return this.divMod(y).quotient;
}

/**
 * Integer remainder.
 *
 * &nbsp;x.mod(y)&nbsp; returns the value &nbsp;z&nbsp; such that &nbsp;0 <= z < y&nbsp; and
 * &nbsp;x - z&nbsp; is divisble by &nbsp;y&nbsp;.
 */
mod(y: UInt64 | number) {
  return this.divMod(y).rest;
}

/**
 * Multiplication with overflow checking.
 */
mul(y: UInt64 | number) {
  let z = this.value.mul(UInt64.from(y).value);
  z.rangeCheckHelper(UInt64.NUM_BITS).assertEquals(z);
  return new UInt64(z);
}

/**
 * Addition with overflow checking.
 */
```

```
add(y: UInt64 | number) {
  let z = this.value.add(UInt64.from(y).value);
  z.rangeCheckHelper(UInt64.NUM_BITS).assertEquals(z);
  return new UInt64(z);
}

/**
 * Subtraction with underflow checking.
 */
sub(y: UInt64 | number) {
  let z = this.value.sub(UInt64.from(y).value);
  z.rangeCheckHelper(UInt64.NUM_BITS).assertEquals(z);
  return new UInt64(z);
}

/**
 * @deprecated Use {@link lessThanOrEqual} instead.
 *
 * Checks if a {@link UInt64} is less than or equal to another one.
 */
lte(y: UInt64) {
  if (this.value.isConstant() && y.value.isConstant()) {
    return Bool(this.value.toBigInt() <= y.value.toBigInt());
  } else {
    let xMinusY = this.value.sub(y.value).seal();
    let yMinusX = xMinusY.neg();
    let xMinusYFits = xMinusY
      .rangeCheckHelper(UInt64.NUM_BITS)
      .equals(xMinusY);
    let yMinusXFits = yMinusX
      .rangeCheckHelper(UInt64.NUM_BITS)
      .equals(yMinusX);
    xMinusYFits.or(yMinusXFits).assertEquals(true);
    // x <= y if y - x fits in 64 bits
    return yMinusXFits;
  }
}

/**
 * Checks if a {@link UInt64} is less than or equal to another one.
 */
lessThanOrEqual(y: UInt64) {
  if (this.value.isConstant() && y.value.isConstant()) {
    return Bool(this.value.toBigInt() <= y.value.toBigInt());
  } else {
    let xMinusY = this.value.sub(y.value).seal();
    let yMinusX = xMinusY.neg();
    let xMinusYFits = xMinusY
      .rangeCheckHelper(UInt64.NUM_BITS)
      .equals(xMinusY);
    let yMinusXFits = yMinusX
      .rangeCheckHelper(UInt64.NUM_BITS)
      .equals(yMinusX);
    xMinusYFits.or(yMinusXFits).assertEquals(true);
    // x <= y if y - x fits in 64 bits
    return yMinusXFits;
  }
}

/**
 * @deprecated Use {@link assertLessThanOrEqual} instead.
 *
 * Asserts that a {@link UInt64} is less than or equal to another one.
 */
assertLte(y: UInt64, message?: string) {
  this.assertLessThanOrEqual(y, message);
}

/**
 * Asserts that a {@link UInt64} is less than or equal to another one.
 */
assertLessThanOrEqual(y: UInt64, message?: string) {
  if (this.value.isConstant() && y.value.isConstant()) {
    let x0 = this.value.toBigInt();
    let y0 = y.value.toBigInt();
    if (x0 > y0) {
      if (message !== undefined) throw Error(message);
      throw Error(&nbsp;UInt64.assertLessThanOrEqual: expected ${x0} <= ${y0}&nbsp;);
    }
    return;
  }
  let yMinusX = y.value.sub(this.value).seal();
  yMinusX.rangeCheckHelper(UInt64.NUM_BITS).assertEquals(yMinusX, message);
}

/**
 * @deprecated Use {@link lessThan} instead.
 *
 * Checks if a {@link UInt64} is less than another one.
 */
lt(y: UInt64) {
  return this.lessThanOrEqual(y).and(this.value.equals(y.value).not());
}

/**
 *
 * Checks if a {@link UInt64} is less than another one.
 */
lessThan(y: UInt64) {
  return this.lessThanOrEqual(y).and(this.value.equals(y.value).not());
}

/**
 *
 * @deprecated Use {@link assertLessThan} instead.
 *
 * Asserts that a {@link UInt64} is less than another one.
 */
assertLt(y: UInt64, message?: string) {
  this.lessThan(y).assertEquals(true, message);
}

/**
 * Asserts that a {@link UInt64} is less than another one.
 */
assertLessThan(y: UInt64, message?: string) {
  this.lessThan(y).assertEquals(true, message);
}

/**
 * @deprecated Use {@link greaterThan} instead.
 *
 * Checks if a {@link UInt64} is greater than another one.
 */
gt(y: UInt64) {
  return y.lessThan(this);
}

/**
 * Checks if a {@link UInt64} is greater than another one.
 */
greaterThan(y: UInt64) {
  return y.lessThan(this);
}

/**
 * @deprecated Use {@link assertGreaterThan} instead.
 *
 * Asserts that a {@link UInt64} is greater than another one.
 */
assertGt(y: UInt64, message?: string) {
  y.assertLessThan(this, message);
}

/**
 * Asserts that a {@link UInt64} is greater than another one.
 */
assertGreaterThan(y: UInt64, message?: string) {
  y.assertLessThan(this, message);
}

/**
 * @deprecated Use {@link greaterThanOrEqual} instead.
 *
 * Checks if a {@link UInt64} is greater than or equal to another one.
 */
```

```
  gte(y: UInt64) {
    return this.lessThan(y).not();
  }

  /**
   * Checks if a {@link UInt64} is greater than or equal to another one.
   */
  greaterThanOrEqual(y: UInt64) {
    return this.lessThan(y).not();
  }

  /**
   * @deprecated Use {@link assertGreaterThanOrEqual} instead.
   *
   * Asserts that a {@link UInt64} is greater than or equal to another one.
   */
  assertGte(y: UInt64, message?: string) {
    y.assertLessThanOrEqual(this, message);
  }

  /**
   * Asserts that a {@link UInt64} is greater than or equal to another one.
   */
  assertGreaterThanOrEqual(y: UInt64, message?: string) {
    y.assertLessThanOrEqual(this, message);
  }
}
/**
 * A 32 bit unsigned integer with values ranging from 0 to 4,294,967,295.
 */
class UInt32 extends CircuitValue {
  @prop value: Field;
  static NUM_BITS = 32;

  /**
   * Static method to create a {@link UInt32} with value &nbsp;0&nbsp;.
   */
  static get zero(): UInt32 {
    return new UInt32(Field(0));
  }

  /**
   * Static method to create a {@link UInt32} with value &nbsp;0&nbsp;.
   */
  static get one(): UInt32 {
    return new UInt32(Field(1));
  }
  /**
   * Turns the {@link UInt32} into a string.
   */
  toString(): string {
    return this.value.toString();
```

```
}
/**
 * Turns the {@link UInt32} into a {@link BigInt}.
 */
toBigint() {
  return this.value.toBigInt();
}
/**
 * Turns the {@link UInt32} into a {@link UInt64}.
 */
toUInt64(): UInt64 {
  // this is safe, because the UInt32 range is included in the UInt64 range
  return new UInt64(this.value);
}

static check(x: UInt32) {
  let actual = x.value.rangeCheckHelper(32);
  actual.assertEquals(x.value);
}
static toInput(x: UInt32): HashInput {
  return { packed: [[x.value, 32]] };
}
/**
 * Encodes this structure into a JSON-like object.
 */
static toJSON(x: UInt32) {
  return x.value.toString();
}

/**
 * Decodes a JSON-like object into this structure.
 */
static fromJSON<T extends AnyConstructor>(x: string): InstanceType<T> {
  return this.from(x) as any;
}

private static checkConstant(x: Field) {
  if (!x.isConstant()) return x;
  let xBig = x.toBigInt();
  if (xBig < 0n || xBig >= 1n << BigInt(this.NUM_BITS)) {
    throw Error(
      &nbsp;UInt32: Expected number between 0 and 2^32 - 1, got ${xBig}&nbsp;
    );
  }
  return x;
}

// this checks the range if the argument is a constant
/**
 * Creates a new {@link UInt32}.
 */
static from(x: UInt32 | Field | number | string | bigint) {
  if (x instanceof UInt32) x = x.value;
  return new this(this.checkConstant(Field(x)));
}
/**
 * Creates a {@link UInt32} with a value of 4,294,967,295.
 */
static MAXINT() {
  return new UInt32(Field((1n << 32n) - 1n));
}

/**
 * Integer division with remainder.
 *
 * &nbsp;x.divMod(y)&nbsp; returns the quotient and the remainder.
 */
divMod(y: UInt32 | number | string) {
  let x = this.value;
  let y_ = UInt32.from(y).value;

  if (x.isConstant() && y_.isConstant()) {
    let xn = x.toBigInt();
    let yn = y_.toBigInt();
    let q = xn / yn;
    let r = xn - q * yn;
    return {
      quotient: new UInt32(new Field(q.toString())),
      rest: new UInt32(new Field(r.toString())),
    };
  }

  y_ = y_.seal();

  let q = Provable.witness(
    Field,
    () => new Field(x.toBigInt() / y_.toBigInt())
  );

  q.rangeCheckHelper(UInt32.NUM_BITS).assertEquals(q);

  // TODO: Could be a bit more efficient
  let r = x.sub(q.mul(y_)).seal();
  r.rangeCheckHelper(UInt32.NUM_BITS).assertEquals(r);

  let r_ = new UInt32(r);
  let q_ = new UInt32(q);

  r_.assertLessThan(new UInt32(y_));

  return { quotient: q_, rest: r_ };
}
/**
 * Integer division.
 *
 * &nbsp;x.div(y)&nbsp; returns the floor of &nbsp;x / y&nbsp;, that is, the greatest
 * &nbsp;z&nbsp; such that &nbsp;x * y <= x&nbsp;.
 *
 */
div(y: UInt32 | number) {
  return this.divMod(y).quotient;
}
/**
 * Integer remainder.
 *
 * &nbsp;x.mod(y)&nbsp; returns the value &nbsp;z&nbsp; such that &nbsp;0 <= z < y&nbsp; and
 * &nbsp;x - z&nbsp; is divisble by &nbsp;y&nbsp;.
 */
mod(y: UInt32 | number) {
  return this.divMod(y).rest;
}
/**
 * Multiplication with overflow checking.
 */
mul(y: UInt32 | number) {
  let z = this.value.mul(UInt32.from(y).value);
  z.rangeCheckHelper(UInt32.NUM_BITS).assertEquals(z);
  return new UInt32(z);
}
/**
 * Addition with overflow checking.
 */
add(y: UInt32 | number) {
  let z = this.value.add(UInt32.from(y).value);
  z.rangeCheckHelper(UInt32.NUM_BITS).assertEquals(z);
  return new UInt32(z);
}
/**
 * Subtraction with underflow checking.
 */
sub(y: UInt32 | number) {
  let z = this.value.sub(UInt32.from(y).value);
  z.rangeCheckHelper(UInt32.NUM_BITS).assertEquals(z);
  return new UInt32(z);
}
/**
 * @deprecated Use {@link lessThanOrEqual} instead.
 *
 * Checks if a {@link UInt32} is less than or equal to another one.
 */
lte(y: UInt32) {
  if (this.value.isConstant() && y.value.isConstant()) {
    return Bool(this.value.toBigInt() <= y.value.toBigInt());
  } else {
    let xMinusY = this.value.sub(y.value).seal();
    let yMinusX = xMinusY.neg();
    let xMinusYFits = xMinusY
      .rangeCheckHelper(UInt32.NUM_BITS)
      .equals(xMinusY);
    let yMinusXFits = yMinusX
      .rangeCheckHelper(UInt32.NUM_BITS)
      .equals(yMinusX);
    xMinusYFits.or(yMinusXFits).assertEquals(true);
    // x <= y if y - x fits in 64 bits
    return yMinusXFits;
  }
}

/**
 * Checks if a {@link UInt32} is less than or equal to another one.
 */
lessThanOrEqual(y: UInt32) {
  if (this.value.isConstant() && y.value.isConstant()) {
    return Bool(this.value.toBigInt() <= y.value.toBigInt());
  } else {
    let xMinusY = this.value.sub(y.value).seal();
    let yMinusX = xMinusY.neg();
    let xMinusYFits = xMinusY
      .rangeCheckHelper(UInt32.NUM_BITS)
      .equals(xMinusY);
    let yMinusXFits = yMinusX
      .rangeCheckHelper(UInt32.NUM_BITS)
      .equals(yMinusX);
    xMinusYFits.or(yMinusXFits).assertEquals(true);
    // x <= y if y - x fits in 64 bits
    return yMinusXFits;
  }
}

/**
 * @deprecated Use {@link assertLessThanOrEqual} instead.
 *
 * Asserts that a {@link UInt32} is less than or equal to another one.
 */
assertLte(y: UInt32, message?: string) {
  this.assertLessThanOrEqual(y, message);
}

/**
 * Asserts that a {@link UInt32} is less than or equal to another one.
 */
assertLessThanOrEqual(y: UInt32, message?: string) {
  if (this.value.isConstant() && y.value.isConstant()) {
    let x0 = this.value.toBigInt();
    let y0 = y.value.toBigInt();
    if (x0 > y0) {
      if (message !== undefined) throw Error(message);
      throw Error(&nbsp;UInt32.assertLessThanOrEqual: expected ${x0} <= ${y0}&nbsp;);
    }
    return;
  }
  let yMinusX = y.value.sub(this.value).seal();
  yMinusX.rangeCheckHelper(UInt32.NUM_BITS).assertEquals(yMinusX, message);
}

/**
 * @deprecated Use {@link lessThan} instead.
 *
 * Checks if a {@link UInt32} is less than another one.
 */
lt(y: UInt32) {
  return this.lessThanOrEqual(y).and(this.value.equals(y.value).not());
}

/**
 * Checks if a {@link UInt32} is less than another one.
 */
lessThan(y: UInt32) {
  return this.lessThanOrEqual(y).and(this.value.equals(y.value).not());
}

/**
 * @deprecated Use {@link assertLessThan} instead.
 *
 * Asserts that a {@link UInt32} is less than another one.
 */
assertLt(y: UInt32, message?: string) {
  this.lessThan(y).assertEquals(true, message);
}

/**
 * Asserts that a {@link UInt32} is less than another one.
 */
assertLessThan(y: UInt32, message?: string) {
  this.lessThan(y).assertEquals(true, message);
}

/**
 * @deprecated Use {@link greaterThan} instead.
 *
 * Checks if a {@link UInt32} is greater than another one.
 */
gt(y: UInt32) {
  return y.lessThan(this);
}

/**
 * Checks if a {@link UInt32} is greater than another one.
 */
greaterThan(y: UInt32) {
    return y.lessThan(this);
  }

  /**
   * @deprecated Use {@link assertGreaterThan} instead.
   *
   * Asserts that a {@link UInt32} is greater than another one.
   */
  assertGt(y: UInt32, message?: string) {
    y.assertLessThan(this, message);
  }

  /**
   * Asserts that a {@link UInt32} is greater than another one.
   */
  assertGreaterThan(y: UInt32, message?: string) {
    y.assertLessThan(this, message);
  }

  /**
   * @deprecated Use {@link greaterThanOrEqual} instead.
   *
   * Checks if a {@link UInt32} is greater than or equal to another one.
   */
  gte(y: UInt32) {
    return this.lessThan(y).not();
  }

  /**
   * Checks if a {@link UInt32} is greater than or equal to another one.
   */
  greaterThanOrEqual(y: UInt32) {
    return this.lessThan(y).not();
  }

  /**
   * @deprecated Use {@link assertGreaterThanOrEqual} instead.

   *
   * Asserts that a {@link UInt32} is greater than or equal to another one.
   */
  assertGte(y: UInt32, message?: string) {
    y.assertLessThanOrEqual(this, message);
  }

  /**
   * Asserts that a {@link UInt32} is greater than or equal to another one.
   */
  assertGreaterThanOrEqual(y: UInt32, message?: string) {
    y.assertLessThanOrEqual(this, message);
  }
}
```

```
class Sign extends CircuitValue {
 @prop value: Field; // +/- 1

 static get one() {
  return new Sign(Field(1));
 }
 static get minusOne() {
  return new Sign(Field(-1));
 }
 static check(x: Sign) {
  // x^2 === 1  <=>  x === 1 or x === -1
  x.value.square().assertEquals(Field(1));
 }
 static emptyValue(): Sign {
  return Sign.one;
 }
 static toInput(x: Sign): HashInput {
  return { packed: [[x.isPositive().toField(), 1]] };
 }
 static toJSON(x: Sign) {
  if (x.toString() === '1') return 'Positive';
  if (x.neg().toString() === '1') return 'Negative';
  throw Error(&nbsp;Invalid Sign: ${x}&nbsp;);
 }
 static fromJSON<T extends AnyConstructor>(
  x: 'Positive' | 'Negative'
 ): InstanceType<T> {
  return (x === 'Positive' ? new Sign(Field(1)) : new Sign(Field(-1))) as any;
 }
 neg() {
  return new Sign(this.value.neg());
 }
 mul(y: Sign) {
  return new Sign(this.value.mul(y.value));
 }
 isPositive() {
  return this.value.equals(Field(1));
 }
 toString() {
  return this.value.toString();
 }
}

type BalanceChange = Types.AccountUpdate['body']['balanceChange'];

/**
 * A 64 bit signed integer with values ranging from -18,446,744,073,709,551,615 to
18,446,744,073,709,551,615.
 */
class Int64 extends CircuitValue implements BalanceChange {
 // * in the range [-2^64+1, 2^64-1], unlike a normal int64
 // * under- and overflowing is disallowed, similar to UInt64, unlike a normal int64+

 @prop magnitude: UInt64; // absolute value
 @prop sgn: Sign; // +/- 1

 // Some thoughts regarding the representation as field elements:
 // toFields returns the in-circuit representation, so the main objective is to minimize the number of
constraints
 // that result from this representation. Therefore, I think the only candidate for an efficient 1-field
representation
 // is the one where the Int64 is the field: toFields = Int64 => [Int64.magnitude.mul(Int64.sign)]. Anything else
involving
 // bit packing would just lead to very inefficient circuit operations.
 //
 // So, is magnitude * sign ("1-field") a more efficient representation than (magnitude, sign) ("2-field")?
 // Several common operations like add, mul, etc, operate on 1-field so in 2-field they result in one additional
multiplication
 // constraint per operand. However, the check operation (constraining to 64 bits + a sign) which is called at
the introduction
 // of every witness, and also at the end of add, mul, etc, operates on 2-field. So here, the 1-field
representation needs
 // to add an additional magnitude * sign = Int64 multiplication constraint, which will typically cancel out most
of the gains
 // achieved by 1-field elsewhere.
 // There are some notable operations for which 2-field is definitely better:
 //
 // * div and mod (which do integer division with rounding on the magnitude)
 // * converting the Int64 to a Currency.Amount.Signed (for the zkapp balance), which has the exact same
(magnitude, sign) representation we use here.
 //
 // The second point is one of the main things an Int64 is used for, and was the original motivation to use 2
fields.
 // Overall, I think the existing implementation is the optimal one.

 constructor(magnitude: UInt64, sgn = Sign.one) {
   super(magnitude, sgn);
 }

 /**
  * Creates a new {@link Int64} from a {@link Field}.
  *
  * Does check if the {@link Field} is within range.
  */
 private static fromFieldUnchecked(x: Field) {
   let TWO64 = 1n << 64n;
   let xBigInt = x.toBigInt();
   let isValidPositive = xBigInt < TWO64; // covers {0,...,2^64 - 1}
   let isValidNegative = Field.ORDER - xBigInt < TWO64; // {-2^64 + 1,...,-1}
   if (!isValidPositive && !isValidNegative)
     throw Error(&nbsp;Int64: Expected a value between (-2^64, 2^64), got ${x}&nbsp;);
   let magnitude = Field(isValidPositive ? x.toString() : x.neg().toString());
   let sign = isValidPositive ? Sign.one : Sign.minusOne;
   return new Int64(new UInt64(magnitude), sign);
 }

 // this doesn't check ranges because we assume they're already checked on UInts
 /**
  * Creates a new {@link Int64} from a {@link Field}.
  *
  * **Does not** check if the {@link Field} is within range.
  */
 static fromUnsigned(x: UInt64 | UInt32) {
   return new Int64(x instanceof UInt32 ? x.toUInt64() : x);
 }

 // this checks the range if the argument is a constant
 /**
  * Creates a new {@link Int64}.
  *
  * Check the range if the argument is a constant.
  */
 static from(x: Int64 | UInt32 | UInt64 | Field | number | string | bigint) {
   if (x instanceof Int64) return x;
   if (x instanceof UInt64 || x instanceof UInt32) {
     return Int64.fromUnsigned(x);
   }
   return Int64.fromFieldUnchecked(Field(x));
 }
 /**
  * Turns the {@link Int64} into a string.
  */
 toString() {
   let abs = this.magnitude.toString();
   let sgn = this.isPositive().toBoolean() || abs === '0' ? '' : '-';
   return sgn + abs;
 }
 isConstant() {
   return this.magnitude.value.isConstant() && this.sgn.isConstant();
 }

 // --- circuit-compatible operations below ---
 // the assumption here is that all Int64 values that appear in a circuit are already checked as valid
 // this is because Provable.witness calls .check, which calls .check on each prop, i.e. UInt64 and Sign
 // so we only have to do additional checks if an operation on valid inputs can have an invalid outcome
(example: overflow)
 /**
  * Static method to create a {@link Int64} with value &nbsp;0&nbsp;.
  */
 static get zero() {
   return new Int64(UInt64.zero);
 }
 /**
  * Static method to create a {@link Int64} with value &nbsp;1&nbsp;.
  */
```

```
static get one() {
  return new Int64(UInt64.one);
}
/**
 * Static method to create a {@link Int64} with value &nbsp;-1&nbsp;.
 */
static get minusOne() {
  return new Int64(UInt64.one).neg();
}

/**
 * Returns the {@link Field} value.
 */
toField() {
  return this.magnitude.value.mul(this.sgn.value);
}
/**
 * Static method to create a {@link Int64} from a {@link Field}.
 */
static fromField(x: Field): Int64 {
  // constant case - just return unchecked value
  if (x.isConstant()) return Int64.fromFieldUnchecked(x);
  // variable case - create a new checked witness and prove consistency with original field
  let xInt = Provable.witness(Int64, () => Int64.fromFieldUnchecked(x));
  xInt.toField().assertEquals(x); // sign(x) * |x| === x
  return xInt;
}

/**
 * Negates the value.
 *
 * &nbsp;Int64.from(5).neg()&nbsp; will turn into &nbsp;Int64.from(-5)&nbsp;
 */
neg() {
  // doesn't need further check if &nbsp;this&nbsp; is valid
  return new Int64(this.magnitude, this.sgn.neg());
}
/**
 * Addition with overflow checking.
 */
add(y: Int64 | number | string | bigint | UInt64 | UInt32) {
  let y_ = Int64.from(y);
  return Int64.fromField(this.toField().add(y_.toField()));
}
/**
 * Subtraction with underflow checking.
 */
sub(y: Int64 | number | string | bigint | UInt64 | UInt32) {
  let y_ = Int64.from(y);
  return Int64.fromField(this.toField().sub(y_.toField()));
}
/**
```

```
 * Multiplication with overflow checking.
 */
mul(y: Int64 | number | string | bigint | UInt64 | UInt32) {
  let y_ = Int64.from(y);
  return Int64.fromField(this.toField().mul(y_.toField()));
}
/**
 * Integer division.
 *
 * &nbsp;x.div(y)&nbsp; returns the floor of &nbsp;x / y&nbsp;, that is, the greatest
 * &nbsp;z&nbsp; such that &nbsp;z * y <= x&nbsp;.
 *
 */
div(y: Int64 | number | string | bigint | UInt64 | UInt32) {
  let y_ = Int64.from(y);
  let { quotient } = this.magnitude.divMod(y_.magnitude);
  let sign = this.sgn.mul(y_.sgn);
  return new Int64(quotient, sign);
}
/**
 * Integer remainder.
 *
 * &nbsp;x.mod(y)&nbsp; returns the value &nbsp;z&nbsp; such that &nbsp;0 <= z < y&nbsp; and
 * &nbsp;x - z&nbsp; is divisble by &nbsp;y&nbsp;.
 */
mod(y: UInt64 | number | string | bigint | UInt32) {
  let y_ = UInt64.from(y);
  let rest = this.magnitude.divMod(y_).rest.value;
  rest = Provable.if(this.isPositive(), rest, y_.value.sub(rest));
  return new Int64(new UInt64(rest));
}

/**
 * Checks if two values are equal.
 */
equals(y: Int64 | number | string | bigint | UInt64 | UInt32) {
  let y_ = Int64.from(y);
  return this.toField().equals(y_.toField());
}
/**
 * Asserts that two values are equal.
 */
assertEquals(
  y: Int64 | number | string | bigint | UInt64 | UInt32,
  message?: string
) {
  let y_ = Int64.from(y);
  this.toField().assertEquals(y_.toField(), message);
}
/**
 * Checks if the value is postive.
 */
```

```
  isPositive() {
    return this.sgn.isPositive();
  }
}
```

</file>

<file>

## path: /src/lib/merkle-tree.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/merkle-tree.unit-test.ts

```
import { Bool, Field } from './core.js';
import { maybeSwap, maybeSwapBad } from './merkle_tree.js';
import { Random, test } from './testing/property.js';
import { expect } from 'expect';

test(Random.bool, Random.field, Random.field, (b, x, y) => {
  let [x0, y0] = maybeSwap(Bool(!!b), Field(x), Field(y));
  let [x1, y1] = maybeSwapBad(Bool(!!b), Field(x), Field(y));

  // both versions of &nbsp;maybeSwap&nbsp; should behave the same
  expect(x0).toEqual(x1);
  expect(y0).toEqual(y1);

  // if the boolean is true, it shouldn't swap the fields; otherwise, it should
  if (b) {
    expect(x0.toBigInt()).toEqual(x);
    expect(y0.toBigInt()).toEqual(y);
  } else {
    expect(x0.toBigInt()).toEqual(y);
    expect(y0.toBigInt()).toEqual(x);
  }
});
```

</file>

<file>

## path: /src/lib/merkle_map.test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/merkle_map.test.ts

```
import { isReady, shutdown, Field, MerkleMap } from 'o1js';

describe('Merkle Map', () => {
  beforeAll(async () => {
    await isReady;
  });
  afterAll(async () => {
    setTimeout(shutdown, 0);
  });

  it('set and get a value from a key', () => {
    const map = new MerkleMap();

    const key = Field.random();
    const value = Field.random();

    map.set(key, value);

    expect(map.get(key).equals(value).toBoolean());
  });

  it('check merkle map witness computes the correct root and key', () => {
    const map = new MerkleMap();

    const key = Field.random();
    const value = Field.random();

    map.set(key, value);

    const witness = map.getWitness(key);

    const emptyMap = new MerkleMap();

    const [emptyLeafWitnessRoot, witnessKey] = witness.computeRootAndKey(
      Field(0)
    );
    const [witnessRoot, _] = witness.computeRootAndKey(value);

    expect(
      emptyLeafWitnessRoot.equals(emptyMap.getRoot()).toBoolean() &&
        witnessKey.equals(key).toBoolean() &&
        witnessRoot.equals(map.getRoot()).toBoolean()
    );
  });
});
```

</file>

<file>

## path: /src/lib/merkle_map.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/merkle_map.ts

```
import { arrayProp, CircuitValue } from './circuit_value.js';
import { Field, Bool } from './core.js';
import { Poseidon } from './hash.js';
import { MerkleTree, MerkleWitness } from './merkle_tree.js';
import { Provable } from './provable.js';

const bits = 255;
const printDebugs = false;

export class MerkleMap {
  tree: InstanceType<typeof MerkleTree>;

  // ------------------------------------------------

  /**
   * Creates a new, empty Merkle Map.
   * @returns A new MerkleMap
   */
  constructor() {
    if (bits > 255) {
      throw Error('bits must be <= 255');
    }
    if (bits !== 255) {
      console.warn(
        'bits set to',
        bits + '. Should be set to 255 in production to avoid collisions'
      );
    }
    this.tree = new MerkleTree(bits + 1);
  }

  // ------------------------------------------------

  _keyToIndex(key: Field) {
    // the bit map is reversed to make reconstructing the key during proving more convenient
    let keyBits = key
      .toBits()
      .slice(0, bits)
      .reverse()
      .map((b) => b.toBoolean());

    let n = 0n;
    for (let i = 0; i < keyBits.length; i++) {
      const b = keyBits[i] ? 1 : 0;
      n += 2n ** BigInt(i) * BigInt(b);
    }

    return n;
  }
```

```
// ---------------------------------------------------------------

/**
 * Sets a key of the merkle map to a given value.
 * @param key The key to set in the map.
 * @param key The value to set.
 */
set(key: Field, value: Field) {
  const index = this._keyToIndex(key);
  this.tree.setLeaf(index, value);
}

// ---------------------------------------------------------------

/**
 * Returns a value given a key. Values are by default Field(0).
 * @param key The key to get the value from.
 * @returns The value stored at the key.
 */
get(key: Field) {
  const index = this._keyToIndex(key);
  return this.tree.getNode(0, index);
}

// ---------------------------------------------------------------

/**
 * Returns the root of the Merkle Map.
 * @returns The root of the Merkle Map.
 */
getRoot() {
  return this.tree.getRoot();
}

/**
  * Returns a circuit-compatible witness (also known as [Merkle Proof or Merkle Witness]
(https://computersciencewiki.org/index.php/Merkle_proof)) for the given key.
  * @param key The key to make a witness for.
  * @returns A MerkleMapWitness, which can be used to assert changes to the MerkleMap, and the
witness's key.
  */
getWitness(key: Field) {
  const index = this._keyToIndex(key);
  class MyMerkleWitness extends MerkleWitness(bits + 1) {}
  const witness = new MyMerkleWitness(this.tree.getWitness(index));

  if (printDebugs) {
    // witness bits and key bits should be the reverse of each other, so
    // we can calculate the key during recursively traversing the path
    console.log(
      'witness bits',
      witness.isLeft.map((l) => (l.toBoolean() ? '0' : '1')).join(', ')
    );
    console.log(
      'key bits',
      key
        .toBits()
        .slice(0, bits)
        .map((l) => (l.toBoolean() ? '1' : '0'))
        .join(', ')
    );
  }
  return new MerkleMapWitness(witness.isLeft, witness.path);
 }
}

// =======================================================

export class MerkleMapWitness extends CircuitValue {
 @arrayProp(Bool, bits) isLefts: Bool[];
 @arrayProp(Field, bits) siblings: Field[];

 constructor(isLefts: Bool[], siblings: Field[]) {
  super();
  this.isLefts = isLefts;
  this.siblings = siblings;
 }

 /**
  * computes the merkle tree root for a given value and the key for this witness
  * @param value The value to compute the root for.
  * @returns A tuple of the computed merkle root, and the key that is connected to the path updated by this
witness.
  */
 computeRootAndKey(value: Field) {
  let hash = value;

  const isLeft = this.isLefts;
  const siblings = this.siblings;

  let key = Field(0);

  for (let i = 0; i < bits; i++) {
   const left = Provable.if(isLeft[i], hash, siblings[i]);
   const right = Provable.if(isLeft[i], siblings[i], hash);
   hash = Poseidon.hash([left, right]);

   const bit = Provable.if(isLeft[i], Field(0), Field(1));

   key = key.mul(2).add(bit);
  }

  return [hash, key];
 }
```

```
}
```

</file>

<file>

## path: /src/lib/merkle_tree.test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/merkle_tree.test.ts

```
import {
  isReady,
  shutdown,
  Poseidon,
  Field,
  MerkleTree,
  MerkleWitness,
} from 'o1js';

describe('Merkle Tree', () => {
  beforeAll(async () => {
    await isReady;
  });
  afterAll(async () => {
    setTimeout(shutdown, 0);
  });

  it('root of empty tree of size 1', () => {
    const tree = new MerkleTree(1);
    expect(tree.getRoot().toString()).toEqual(Field(0).toString());
  });

  it('root is correct', () => {
    const tree = new MerkleTree(2);
    tree.setLeaf(0n, Field(1));
    tree.setLeaf(1n, Field(2));
    expect(tree.getRoot().toString()).toEqual(
      Poseidon.hash([Field(1), Field(2)]).toString()
    );
  });

  it('builds correct tree', () => {
    const tree = new MerkleTree(4);
    tree.setLeaf(0n, Field(1));
    tree.setLeaf(1n, Field(2));
    tree.setLeaf(2n, Field(3));
    expect(tree.validate(0n)).toBe(true);
    expect(tree.validate(1n)).toBe(true);
    expect(tree.validate(2n)).toBe(true);
    expect(tree.validate(3n)).toBe(true);
  });
```

```
 it('tree of height 128', () => {
   const tree = new MerkleTree(128);

   const index = 2n ** 64n;
   expect(tree.validate(index)).toBe(true);

   tree.setLeaf(index, Field(1));
   expect(tree.validate(index)).toBe(true);
 });

 it('tree of height 256', () => {
   const tree = new MerkleTree(256);

   const index = 2n ** 128n;
   expect(tree.validate(index)).toBe(true);

   tree.setLeaf(index, Field(1));
   expect(tree.validate(index)).toBe(true);
 });

 it('works with MerkleWitness', () => {
   // tree with height 3 (4 leaves)
   const HEIGHT = 3;
   let tree = new MerkleTree(HEIGHT);
   class MyMerkleWitness extends MerkleWitness(HEIGHT) {}

   // tree with the leaves [15, 16, 17, 18]
   tree.fill([15, 16, 17, 18].map(Field));

   // witness for the leaf '17', at index 2
   let witness = new MyMerkleWitness(tree.getWitness(2n));

   // calculate index
   expect(witness.calculateIndex().toString()).toEqual('2');

   // calculate root
   let root = witness.calculateRoot(Field(17));
   expect(tree.getRoot()).toEqual(root);

   root = witness.calculateRoot(Field(16));
   expect(tree.getRoot()).not.toEqual(root);

   // construct and check path manually
   let leftHalfHash = Poseidon.hash([Field(15), Field(16)]).toString();
   let expectedWitness = {
     path: ['18', leftHalfHash],
     isLeft: [true, false],
   };
   expect(witness.toJSON()).toEqual(expectedWitness);
 });
});
```

</file>

<file>

## path: /src/lib/merkle_tree.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/merkle_tree.ts

```
/**
 * This file contains all code related to the [Merkle Tree](https://en.wikipedia.org/wiki/Merkle_tree)
implementation available in o1js.
 */

import { CircuitValue, arrayProp } from './circuit_value.js';
import { Circuit } from './circuit.js';
import { Poseidon } from './hash.js';
import { Bool, Field } from './core.js';
import { Provable } from './provable.js';

// external API
export { Witness, MerkleTree, MerkleWitness, BaseMerkleWitness };

// internal API
export { maybeSwap, maybeSwapBad };

type Witness = { isLeft: boolean; sibling: Field }[];

/**
 * A [Merkle Tree](https://en.wikipedia.org/wiki/Merkle_tree) is a binary tree in which every leaf is the
cryptography hash of a piece of data,
 * and every node is the hash of the concatenation of its two child nodes.
 *
 * A Merkle Tree allows developers to easily and securely verify the integrity of large amounts of data.
 *
 * Take a look at our [documentation](https://docs.minaprotocol.com/en/zkapps) on how to use Merkle Trees
in combination with zkApps and zero knowledge programming!
 *
 * Levels are indexed from leaves (level 0) to root (level N - 1).
 */
class MerkleTree {
  private nodes: Record<number, Record<string, Field>> = {};
  private zeroes: Field[];

  /**
   * Creates a new, empty [Merkle Tree](https://en.wikipedia.org/wiki/Merkle_tree).
   * @param height The height of Merkle Tree.
   * @returns A new MerkleTree
   */
  constructor(public readonly height: number) {
    this.zeroes = new Array(height);
    this.zeroes[0] = Field(0);
    for (let i = 1; i < height; i += 1) {
      this.zeroes[i] = Poseidon.hash([this.zeroes[i - 1], this.zeroes[i - 1]]);
    }
  }

  /**
   * Returns a node which lives at a given index and level.
   * @param level Level of the node.
   * @param index Index of the node.
   * @returns The data of the node.
   */
  getNode(level: number, index: bigint): Field {
    return this.nodes[level]?.[index.toString()] ?? this.zeroes[level];
  }

  /**
   * Returns the root of the [Merkle Tree](https://en.wikipedia.org/wiki/Merkle_tree).
   * @returns The root of the Merkle Tree.
   */
  getRoot(): Field {
    return this.getNode(this.height - 1, 0n);
  }

  // TODO: this allows to set a node at an index larger than the size. OK?
  private setNode(level: number, index: bigint, value: Field) {
    (this.nodes[level] ??= {})[index.toString()] = value;
  }

  // TODO: if this is passed an index bigger than the max, it will set a couple of out-of-bounds nodes but not
affect the real Merkle root. OK?
  /**
   * Sets the value of a leaf node at a given index to a given value.
   * @param index Position of the leaf node.
   * @param leaf New value.
   */
  setLeaf(index: bigint, leaf: Field) {
    if (index >= this.leafCount) {
      throw new Error(
        &nbsp;index ${index} is out of range for ${this.leafCount} leaves.&nbsp;
      );
    }
    this.setNode(0, index, leaf);
    let currIndex = index;
    for (let level = 1; level < this.height; level++) {
      currIndex /= 2n;

      const left = this.getNode(level - 1, currIndex * 2n);
      const right = this.getNode(level - 1, currIndex * 2n + 1n);

      this.setNode(level, currIndex, Poseidon.hash([left, right]));
    }
```

```
}

/**
 * Returns the witness (also known as [Merkle Proof or Merkle Witness]
(https://computersciencewiki.org/index.php/Merkle_proof)) for the leaf at the given index.
 * @param index Position of the leaf node.
 * @returns The witness that belongs to the leaf.
 */
getWitness(index: bigint): Witness {
  if (index >= this.leafCount) {
    throw new Error(
      &nbsp;index ${index} is out of range for ${this.leafCount} leaves.&nbsp;
    );
  }
  const witness = [];
  for (let level = 0; level < this.height - 1; level++) {
    const isLeft = index % 2n === 0n;
    const sibling = this.getNode(level, isLeft ? index + 1n : index - 1n);
    witness.push({ isLeft, sibling });
    index /= 2n;
  }
  return witness;
}

// TODO: this will always return true if the merkle tree was constructed normally; seems to be only useful
for testing. remove?
/**
 * Checks if the witness that belongs to the leaf at the given index is a valid witness.
 * @param index Position of the leaf node.
 * @returns True if the witness for the leaf node is valid.
 */
validate(index: bigint): boolean {
  const path = this.getWitness(index);
  let hash = this.getNode(0, index);
  for (const node of path) {
    hash = Poseidon.hash(
      node.isLeft ? [hash, node.sibling] : [node.sibling, hash]
    );
  }

  return hash.toString() === this.getRoot().toString();
}

// TODO: should this take an optional offset? should it fail if the array is too long?
/**
 * Fills all leaves of the tree.
 * @param leaves Values to fill the leaves with.
 */
fill(leaves: Field[]) {
  leaves.forEach((value, index) => {
    this.setLeaf(BigInt(index), value);
  });
  }

  /**
   * Returns the amount of leaf nodes.
   * @returns Amount of leaf nodes.
   */
  get leafCount(): bigint {
    return 2n ** BigInt(this.height - 1);
  }
}

/**
 * The {@link BaseMerkleWitness} class defines a circuit-compatible base class for [Merkle Witness']
(https://computersciencewiki.org/index.php/Merkle_proof).
 */
class BaseMerkleWitness extends CircuitValue {
  static height: number;
  path: Field[];
  isLeft: Bool[];
  height(): number {
    return (this.constructor as any).height;
  }

  /**
   * Takes a {@link Witness} and turns it into a circuit-compatible Witness.
   * @param witness Witness.
   * @returns A circuit-compatible Witness.
   */
  constructor(witness: Witness) {
    super();
    let height = witness.length + 1;
    if (height !== this.height()) {
      throw Error(
        &nbsp;Length of witness ${height}-1 doesn't match static tree height ${this.height()}.&nbsp;
      );
    }
    this.path = witness.map((item) => item.sibling);
    this.isLeft = witness.map((item) => Bool(item.isLeft));
  }

  /**
   * Calculates a root depending on the leaf value.
   * @param leaf Value of the leaf node that belongs to this Witness.
   * @returns The calculated root.
   */
  calculateRoot(leaf: Field): Field {
    let hash = leaf;
    let n = this.height();

    for (let i = 1; i < n; ++i) {
      let isLeft = this.isLeft[i - 1];
      const [left, right] = maybeSwap(isLeft, hash, this.path[i - 1]);
      hash = Poseidon.hash([left, right]);
    }

    return hash;
  }

  /**
   * Calculates a root depending on the leaf value.
   * @deprecated This is a less efficient version of {@link calculateRoot} which was added for compatibility
with existing deployed contracts
   */
  calculateRootSlow(leaf: Field): Field {
    let hash = leaf;
    let n = this.height();

    for (let i = 1; i < n; ++i) {
      let isLeft = this.isLeft[i - 1];
      const [left, right] = maybeSwapBad(isLeft, hash, this.path[i - 1]);
      hash = Poseidon.hash([left, right]);
    }

    return hash;
  }

  /**
   * Calculates the index of the leaf node that belongs to this Witness.
   * @returns Index of the leaf.
   */
  calculateIndex(): Field {
    let powerOfTwo = Field(1);
    let index = Field(0);
    let n = this.height();

    for (let i = 1; i < n; ++i) {
      index = Provable.if(this.isLeft[i - 1], index, index.add(powerOfTwo));
      powerOfTwo = powerOfTwo.mul(2);
    }

    return index;
  }
}

/**
 * Returns a circuit-compatible Witness for a specific Tree height.
 * @param height Height of the Merkle Tree that this Witness belongs to.
 * @returns A circuit-compatible Merkle Witness.
 */
function MerkleWitness(height: number): typeof BaseMerkleWitness {
  class MerkleWitness_ extends BaseMerkleWitness {
    static height = height;
  }
  arrayProp(Field, height - 1)(MerkleWitness_.prototype, 'path');
  arrayProp(Bool, height - 1)(MerkleWitness_.prototype, 'isLeft');
  return MerkleWitness_;
}

function maybeSwapBad(b: Bool, x: Field, y: Field): [Field, Field] {
  const x_ = Provable.if(b, x, y); // y + b*(x - y)
  const y_ = Provable.if(b, y, x); // x + b*(y - x)
  return [x_, y_];
}

// more efficient version of &nbsp;maybeSwapBad&nbsp; which reuses an intermediate variable
function maybeSwap(b: Bool, x: Field, y: Field): [Field, Field] {
  let m = b.toField().mul(x.sub(y)); // b*(x - y)
  const x_ = y.add(m); // y + b*(x - y)
  const y_ = x.sub(m); // x - b*(x - y) = x + b*(y - x)
  return [x_, y_];
}
```

</file>

<file>

## path: /src/lib/mina/account.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/mina/account.ts

```
import { Types } from '../../bindings/mina-transaction/types.js';
import { Bool, Field } from '../core.js';
import { Permissions } from '../account_update.js';
import { UInt32, UInt64 } from '../int.js';
import { PublicKey } from '../signature.js';
import { TokenId, ReceiptChainHash } from '../base58-encodings.js';
import { genericLayoutFold } from '../../bindings/lib/from-layout.js';
import {
  customTypes,
  TypeMap,
} from '../../bindings/mina-transaction/gen/transaction.js';
import { jsLayout } from '../../bindings/mina-transaction/gen/js-layout.js';

export { FetchedAccount, Account, PartialAccount };
export { accountQuery, parseFetchedAccount, fillPartialAccount };

type AuthRequired = Types.Json.AuthRequired;
type Account = Types.Account;
const Account = Types.Account;

type PartialAccount = Omit<Partial<Account>, 'zkapp'> & {
  zkapp?: Partial<Account['zkapp']>;
};

// TODO auto-generate this type and the query
```

```
type FetchedAccount = {
  publicKey: string;
  token: string;
  nonce: string;
  balance: { total: string };
  tokenSymbol: string | null;
  receiptChainHash: string | null;
  timing: {
    initialMinimumBalance: string | null;
    cliffTime: string | null;
    cliffAmount: string | null;
    vestingPeriod: string | null;
    vestingIncrement: string | null;
  };
  permissions: {
    editState: AuthRequired;
    access: AuthRequired;
    send: AuthRequired;
    receive: AuthRequired;
    setDelegate: AuthRequired;
    setPermissions: AuthRequired;
    setVerificationKey: AuthRequired;
    setZkappUri: AuthRequired;
    editActionState: AuthRequired;
    setTokenSymbol: AuthRequired;
    incrementNonce: AuthRequired;
    setVotingFor: AuthRequired;
    setTiming: AuthRequired;
  } | null;
  delegateAccount: { publicKey: string } | null;
  votingFor: string | null;
  zkappState: string[] | null;
  verificationKey: { verificationKey: string; hash: string } | null;
  actionState: string[] | null;
  provedState: boolean | null;
  zkappUri: string | null;
};
const accountQuery = (publicKey: string, tokenId: string) => &nbsp;{
  account(publicKey: "${publicKey}", token: "${tokenId}") {
    publicKey
    token
    nonce
    balance { total }
    tokenSymbol
    receiptChainHash
    timing {
      initialMinimumBalance
      cliffTime
      cliffAmount
      vestingPeriod
      vestingIncrement
    }
    permissions {
      editState
      access
      send
      receive
      setDelegate
      setPermissions
      setVerificationKey
      setZkappUri
      editActionState
      setTokenSymbol
      incrementNonce
      setVotingFor
      setTiming
    }
    delegateAccount { publicKey }
    votingFor
    zkappState
    verificationKey {
      verificationKey
      hash
    }
    actionState
    provedState
    zkappUri
  }
}
&nbsp;;

// convert FetchedAccount (from graphql) to Account (internal representation both here and in Mina)
function parseFetchedAccount({
  publicKey,
  nonce,
  zkappState,
  balance,
  permissions,
  timing: {
    cliffAmount,
    cliffTime,
    initialMinimumBalance,
    vestingIncrement,
    vestingPeriod,
  },
  delegateAccount,
  receiptChainHash,
  actionState,
  token,
  tokenSymbol,
  verificationKey,
  provedState,
  zkappUri,
}: FetchedAccount): Account {
```

```
let hasZkapp =
  zkappState !== null ||
  verificationKey !== null ||
  actionState !== null ||
  zkappUri !== null ||
  provedState;
let partialAccount: PartialAccount = {
  publicKey: PublicKey.fromBase58(publicKey),
  tokenId: TokenId.fromBase58(token),
  tokenSymbol: tokenSymbol ?? undefined,
  balance: balance && UInt64.from(balance.total),
  nonce: UInt32.from(nonce),
  receiptChainHash:
    (receiptChainHash && ReceiptChainHash.fromBase58(receiptChainHash)) ||
    undefined,
  delegate:
    (delegateAccount && PublicKey.fromBase58(delegateAccount.publicKey)) ??
    undefined,
  votingFor: undefined, // TODO
  timing:
    (cliffAmount &&
      cliffTime &&
      initialMinimumBalance &&
      vestingIncrement &&
      vestingPeriod && {
        isTimed: Bool(true),
        cliffAmount: UInt64.from(cliffAmount),
        cliffTime: UInt32.from(cliffTime),
        initialMinimumBalance: UInt64.from(initialMinimumBalance),
        vestingIncrement: UInt64.from(vestingIncrement),
        vestingPeriod: UInt32.from(vestingPeriod),
      }) ||
    undefined,
  permissions:
    (permissions && Permissions.fromJSON(permissions)) ??
    Permissions.initial(),
  zkapp: hasZkapp
    ? {
        appState: (zkappState && zkappState.map(Field)) ?? undefined,
        verificationKey:
          (verificationKey && {
            data: verificationKey.verificationKey,
            hash: Field(verificationKey.hash),
          }) ??
          undefined,
        zkappVersion: undefined, // TODO
        actionState: (actionState && actionState.map(Field)) ?? undefined,
        lastActionSlot: undefined, // TODO
        provedState: provedState !== null ? Bool(provedState) : undefined,
        zkappUri: zkappUri !== null ? zkappUri : undefined,
      }
    : undefined,
  };
  return fillPartialAccount(partialAccount);
}

function fillPartialAccount(account: PartialAccount): Account {
  return genericLayoutFold(
    TypeMap,
    customTypes,
    {
      map(type, value) {
        // if value exists, use it; otherwise fall back to dummy value
        if (value !== undefined) return value;
        // fall back to dummy value
        if (type.emptyValue) return type.emptyValue();
        return type.fromFields(
          Array(type.sizeInFields()).fill(Field(0)),
          type.toAuxiliary()
        );
      },
      reduceArray(array) {
        return array;
      },
      reduceObject(_, record) {
        return record;
      },
      reduceFlaggedOption() {
        // doesn't occur for accounts
        throw Error('not relevant');
      },
      reduceOrUndefined(value) {
        // don't fill in value that's allowed to be undefined
        return value;
      },
    },
    jsLayout.Account as any,
    account as unknown
  );
}
```

</file>

<file>

## path: /src/lib/mina/constants.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/mina/constants.ts

```
/**
 * This file contains constants used in the Mina protocol.
 * Originally defined in the mina_compile_config file in the mina repo:
 * https://github.com/MinaProtocol/mina/blob/develop/src/lib/mina_compile_config/mina_compile_config.ml
 */

// Constants used to calculate cost of a transaction
export namespace TransactionCost {
  // Defined in
https://github.com/MinaProtocol/mina/blob/e8c743488cf0c8f0b7925b7a48a914ca73ed13a1/src/lib/mina_compile_config/mina_compile_config.ml#L67
  export const PROOF_COST = 10.26 as const;

  // Defined in
https://github.com/MinaProtocol/mina/blob/e8c743488cf0c8f0b7925b7a48a914ca73ed13a1/src/lib/mina_compile_config/mina_compile_config.ml#L69
  export const SIGNED_PAIR_COST = 10.08 as const;

  // Defined in
https://github.com/MinaProtocol/mina/blob/e39abf79b7fdf96717eb8a8ee88ec42ba1e2663d/src/lib/mina_compile_config/mina_compile_config.ml#L71
  export const SIGNED_SINGLE_COST = 9.14 as const;

  // Defined in
https://github.com/MinaProtocol/mina/blob/e39abf79b7fdf96717eb8a8ee88ec42ba1e2663d/src/lib/mina_compile_config/mina_compile_config.ml#L73
  export const COST_LIMIT = 69.45 as const;
}

// Constants to define the maximum number of events and actions in a transaction
export namespace TransactionLimits {
  // Defined in
https://github.com/MinaProtocol/mina/blob/e39abf79b7fdf96717eb8a8ee88ec42ba1e2663d/src/lib/mina_compile_config/mina_compile_config.ml#L75
  export const MAX_ACTION_ELEMENTS = 100 as const;
  // Defined in
https://github.com/MinaProtocol/mina/blob/e39abf79b7fdf96717eb8a8ee88ec42ba1e2663d/src/lib/mina_compile_config/mina_compile_config.ml#L77
  export const MAX_EVENT_ELEMENTS = 100 as const;
}
```

</file>

<file>

## path: /src/lib/mina/errors.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/mina/errors.ts

```
import { Types } from '../../bindings/mina-transaction/types.js';
import { TokenId } from '../account_update.js';
import { Int64 } from '../int.js';

export { invalidTransactionError };

const ErrorHandlers = {
  Invalid_fee_excess({
    transaction: { accountUpdates },
    isFeePayer,
    accountCreationFee,
  }: ErrorHandlerArgs) {
    // TODO: handle fee payer for Invalid_fee_excess?
    if (isFeePayer) return;

    let balances = accountUpdates.map(({ body }) => {
      if (body.tokenId.equals(TokenId.default).toBoolean()) {
        return Number(Int64.fromObject(body.balanceChange).toString()) * 1e-9;
      }
    });
    let sum = balances.reduce((a = 0, b = 0) => a + b) ?? 0;
    return &nbsp;Invalid fee excess.
This means that balance changes in your transaction do not sum up to the amount of fees needed.
Here's the list of balance changes:

${balances
  .map((balance, i) => {
    return &nbsp;Account update #${i + 1}) ${
      balance === undefined
        ? 'not a MINA account'
        : &nbsp;${balance.toFixed(2)} MINA&nbsp;
    }&nbsp;;
  })
  .join(&nbsp;\n&nbsp;)}

Total change: ${sum.toFixed(2)} MINA

If there are no new accounts created in your transaction, then this sum should be equal to 0.00 MINA.
If you are creating new accounts -- by updating accounts that didn't exist yet --
then keep in mind the ${(Number(accountCreationFee) * 1e-9).toFixed(
      2
    )} MINA account creation fee, and make sure that the sum equals
${(-Number(accountCreationFee) * 1e-9).toFixed(
  2
)} times the number of newly created accounts.&nbsp;;
  },
};

type ErrorHandlerArgs = {
  transaction: Types.ZkappCommand;
  accountUpdateIndex: number;
  isFeePayer: boolean;
  accountCreationFee: string | number;
};
```

```
function invalidTransactionError(
  transaction: Types.ZkappCommand,
  errors: string[][][],
  additionalContext: { accountCreationFee: string | number }
): string {
  let errorMessages = [];
  let rawErrors = JSON.stringify(errors);

  // handle errors for fee payer
  let errorsForFeePayer = errors[0];
  for (let [error] of errorsForFeePayer) {
    let message = ErrorHandlers[error as keyof typeof ErrorHandlers]?.({
      transaction,
      accountUpdateIndex: NaN,
      isFeePayer: true,
      ...additionalContext,
    });
    if (message) errorMessages.push(message);
  }

  // handle errors for each account update
  let n = transaction.accountUpdates.length;
  for (let i = 0; i < n; i++) {
    let errorsForUpdate = errors[i + 1];
    for (let [error] of errorsForUpdate) {
      let message = ErrorHandlers[error as keyof typeof ErrorHandlers]?.({
        transaction,
        accountUpdateIndex: i,
        isFeePayer: false,
        ...additionalContext,
      });
      if (message) errorMessages.push(message);
    }
  }

  if (errorMessages.length > 1) {
    return [
      'There were multiple errors when applying your transaction:',
      ...errorMessages.map((msg, i) => &nbsp;${i + 1}.) ${msg}&nbsp;),
      &nbsp;Raw list of errors: ${rawErrors}&nbsp;,
    ].join('\n\n');
  }
  if (errorMessages.length === 1) {
    return &nbsp;${errorMessages[0]}\n\nRaw list of errors: ${rawErrors}&nbsp;;
  }
  // fallback if we don't have a good error message yet
  return rawErrors;
}
```

</file>

<file>

## path: /src/lib/mina.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/mina.ts

```
import { Ledger } from '../snarky.js';
import { Field } from './core.js';
import { UInt32, UInt64 } from './int.js';
import { PrivateKey, PublicKey } from './signature.js';
import {
  addMissingProofs,
  addMissingSignatures,
  FeePayerUnsigned,
  ZkappCommand,
  AccountUpdate,
  ZkappPublicInput,
  TokenId,
  CallForest,
  Authorization,
  Actions,
  Events,
  dummySignature,
} from './account_update.js';
import * as Fetch from './fetch.js';
import { assertPreconditionInvariants, NetworkValue } from './precondition.js';
import { cloneCircuitValue, toConstant } from './circuit_value.js';
import { Empty, Proof, verify } from './proof_system.js';
import { Context } from './global-context.js';
import { SmartContract } from './zkapp.js';
import { invalidTransactionError } from './mina/errors.js';
import { Types, TypesBigint } from '../bindings/mina-transaction/types.js';
import { Account } from './mina/account.js';
import { TransactionCost, TransactionLimits } from './mina/constants.js';
import { Provable } from './provable.js';
import { prettifyStacktrace } from './errors.js';
import { Ml } from './ml/conversion.js';
import {
  transactionCommitments,
  verifyAccountUpdateSignature,
} from '../mina-signer/src/sign-zkapp-command.js';

export {
  createTransaction,
  BerkeleyQANet,
  Network,
  LocalBlockchain,
  currentTransaction,
  CurrentTransaction,
  Transaction,
  TransactionId,
  activeInstance,
  setActiveInstance,
  transaction,
  sender,
  currentSlot,
  getAccount,
  hasAccount,
  getBalance,
  getNetworkState,
  accountCreationFee,
  sendTransaction,
  fetchEvents,
  fetchActions,
  getActions,
  FeePayerSpec,
  ActionStates,
  faucet,
  waitForFunding,
  getProofsEnabled,
  // for internal testing only
  filterGroups,
};
interface TransactionId {
  isSuccess: boolean;
  wait(options?: { maxAttempts?: number; interval?: number }): Promise<void>;
  hash(): string | undefined;
}

type Transaction = {
  /**
   * Transaction structure used to describe a state transition on the Mina blockchain.
   */
  transaction: ZkappCommand;
  /**
   * Returns a JSON representation of the {@link Transaction}.
   */
  toJSON(): string;
  /**
   * Returns a pretty-printed JSON representation of the {@link Transaction}.
   */
  toPretty(): any;
  /**
   * Returns the GraphQL query for the Mina daemon.
   */
  toGraphqlQuery(): string;
  /**
   * Signs all {@link AccountUpdate}s included in the {@link Transaction} that require a signature.
   *
   * {@link AccountUpdate}s that require a signature can be specified with &nbsp;
{AccountUpdate|SmartContract}.requireSignature()&nbsp;.
   *
   * @param additionalKeys The list of keys that should be used to sign the {@link Transaction}
   */
  sign(additionalKeys?: PrivateKey[]): Transaction;
  /**
   * Generates proofs for the {@link Transaction}.
   *
   * This can take some time.
   */
  prove(): Promise<(Proof<ZkappPublicInput, Empty> | undefined)[]>;
  /**
   * Sends the {@link Transaction} to the network.
   */
  send(): Promise<TransactionId>;
};

const Transaction = {
  fromJSON(json: Types.Json.ZkappCommand): Transaction {
    let transaction = ZkappCommand.fromJSON(json);
    return newTransaction(transaction, activeInstance.proofsEnabled);
  },
};

type FetchMode = 'fetch' | 'cached' | 'test';
type CurrentTransaction = {
  sender?: PublicKey;
  accountUpdates: AccountUpdate[];
  fetchMode: FetchMode;
  isFinalRunOutsideCircuit: boolean;
  numberOfRuns: 0 | 1 | undefined;
};

let currentTransaction = Context.create<CurrentTransaction>();

/**
 * Allows you to specify information about the fee payer account and the transaction.
 */
type FeePayerSpec =
  | PublicKey
  | {
      sender: PublicKey;
      fee?: number | string | UInt64;
      memo?: string;
      nonce?: number;
    }
  | undefined;

type DeprecatedFeePayerSpec =
  | PublicKey
  | PrivateKey
  | ((
      | {
          feePayerKey: PrivateKey;
          sender?: PublicKey;
```

```
        }
      | {
          feePayerKey?: PrivateKey;
          sender: PublicKey;
        }
    ) & {
      fee?: number | string | UInt64;
      memo?: string;
      nonce?: number;
    })
  | undefined;

type ActionStates = {
  fromActionState?: Field;
  endActionState?: Field;
};

function reportGetAccountError(publicKey: string, tokenId: string) {
  if (tokenId === TokenId.toBase58(TokenId.default)) {
    return &nbsp;getAccount: Could not find account for public key ${publicKey}&nbsp;;
  } else {
    return &nbsp;getAccount: Could not find account for public key ${publicKey} with the tokenId
${tokenId}&nbsp;;
  }
}

function createTransaction(
  feePayer: DeprecatedFeePayerSpec,
  f: () => unknown,
  numberOfRuns: 0 | 1 | undefined,
  {
    fetchMode = 'cached' as FetchMode,
    isFinalRunOutsideCircuit = true,
    proofsEnabled = true,
  } = {}
): Transaction {
  if (currentTransaction.has()) {
    throw new Error('Cannot start new transaction within another transaction');
  }
  let feePayerSpec: {
    sender?: PublicKey;
    feePayerKey?: PrivateKey;
    fee?: number | string | UInt64;
    memo?: string;
    nonce?: number;
  };
  if (feePayer === undefined) {
    feePayerSpec = {};
  } else if (feePayer instanceof PrivateKey) {
    feePayerSpec = { feePayerKey: feePayer, sender: feePayer.toPublicKey() };
  } else if (feePayer instanceof PublicKey) {
    feePayerSpec = { sender: feePayer };
```

```
 } else {
   feePayerSpec = feePayer;
   if (feePayerSpec.sender === undefined)
     feePayerSpec.sender = feePayerSpec.feePayerKey?.toPublicKey();
 }
 let { feePayerKey, sender, fee, memo = '', nonce } = feePayerSpec;

 let transactionId = currentTransaction.enter({
   sender,
   accountUpdates: [],
   fetchMode,
   isFinalRunOutsideCircuit,
   numberOfRuns,
 });

 // run circuit
 // we have this while(true) loop because one of the smart contracts we're calling inside &nbsp;f&nbsp;
might be calling
 // SmartContract.analyzeMethods, which would be running its methods again inside
&nbsp;Provable.constraintSystem&nbsp;, which
 // would throw an error when nested inside &nbsp;Provable.runAndCheck&nbsp;. So if that happens, we
have to run &nbsp;analyzeMethods&nbsp; first
 // and retry &nbsp;Provable.runAndCheck(f)&nbsp;. Since at this point in the function, we don't know which
smart contracts are involved,
 // we created that hack with a &nbsp;bootstrap()&nbsp; function that analyzeMethods sticks on the error, to
call itself again.
 try {
   let err: any;
   while (true) {
     if (err !== undefined) err.bootstrap();
     try {
       if (fetchMode === 'test') {
         Provable.runUnchecked(() => {
           f();
           Provable.asProver(() => {
             let tx = currentTransaction.get();
             tx.accountUpdates = CallForest.map(tx.accountUpdates, (a) =>
               toConstant(AccountUpdate, a)
             );
           });
         });
       } else {
         f();
       }
       break;
     } catch (err_) {
       if ((err_ as any)?.bootstrap) err = err_;
       else throw err_;
     }
   }
 } catch (err) {
   currentTransaction.leave(transactionId);
```

```
    throw err;
  }
  let accountUpdates = currentTransaction.get().accountUpdates;
  // TODO: I'll be back
  // CallForest.addCallers(accountUpdates);
  accountUpdates = CallForest.toFlatList(accountUpdates);

  try {
    // check that on-chain values weren't used without setting a precondition
    for (let accountUpdate of accountUpdates) {
      assertPreconditionInvariants(accountUpdate);
    }
  } catch (err) {
    currentTransaction.leave(transactionId);
    throw err;
  }

  let feePayerAccountUpdate: FeePayerUnsigned;
  if (sender !== undefined) {
    // if senderKey is provided, fetch account to get nonce and mark to be signed
    let nonce_;
    let senderAccount = getAccount(sender, TokenId.default);

    if (nonce === undefined) {
      nonce_ = senderAccount.nonce;
    } else {
      nonce_ = UInt32.from(nonce);
      senderAccount.nonce = nonce_;
      Fetch.addCachedAccount(senderAccount);
    }
    feePayerAccountUpdate = AccountUpdate.defaultFeePayer(sender, nonce_);
    if (feePayerKey !== undefined)
      feePayerAccountUpdate.lazyAuthorization!.privateKey = feePayerKey;
    if (fee !== undefined) {
      feePayerAccountUpdate.body.fee =
        fee instanceof UInt64 ? fee : UInt64.from(String(fee));
    }
  } else {
    // otherwise use a dummy fee payer that has to be filled in later
    feePayerAccountUpdate = AccountUpdate.dummyFeePayer();
  }

  let transaction: ZkappCommand = {
    accountUpdates,
    feePayer: feePayerAccountUpdate,
    memo,
  };

  currentTransaction.leave(transactionId);
  return newTransaction(transaction, proofsEnabled);
}
```

```
function newTransaction(transaction: ZkappCommand, proofsEnabled?: boolean) {
  let self: Transaction = {
    transaction,
    sign(additionalKeys?: PrivateKey[]) {
      self.transaction = addMissingSignatures(self.transaction, additionalKeys);
      return self;
    },
    async prove() {
      let { zkappCommand, proofs } = await addMissingProofs(self.transaction, {
        proofsEnabled,
      });
      self.transaction = zkappCommand;
      return proofs;
    },
    toJSON() {
      let json = ZkappCommand.toJSON(self.transaction);
      return JSON.stringify(json);
    },
    toPretty() {
      return ZkappCommand.toPretty(self.transaction);
    },
    toGraphqlQuery() {
      return Fetch.sendZkappQuery(self.toJSON());
    },
    async send() {
      try {
        return await sendTransaction(self);
      } catch (error) {
        throw prettifyStacktrace(error);
      }
    },
  };
  return self;
}

interface Mina {
  transaction(
    sender: DeprecatedFeePayerSpec,
    f: () => void
  ): Promise<Transaction>;
  currentSlot(): UInt32;
  hasAccount(publicKey: PublicKey, tokenId?: Field): boolean;
  getAccount(publicKey: PublicKey, tokenId?: Field): Account;
  getNetworkState(): NetworkValue;
  getNetworkConstants(): {
    genesisTimestamp: UInt64;
    /**
     * Duration of 1 slot in millisecondw
     */
    slotTime: UInt64;
    accountCreationFee: UInt64;
  };
  accountCreationFee(): UInt64;
  sendTransaction(transaction: Transaction): Promise<TransactionId>;
  fetchEvents: (
    publicKey: PublicKey,
    tokenId?: Field,
    filterOptions?: Fetch.EventActionFilterOptions
  ) => ReturnType<typeof Fetch.fetchEvents>;
  fetchActions: (
    publicKey: PublicKey,
    actionStates?: ActionStates,
    tokenId?: Field
  ) => ReturnType<typeof Fetch.fetchActions>;
  getActions: (
    publicKey: PublicKey,
    actionStates?: ActionStates,
    tokenId?: Field
  ) => { hash: string; actions: string[][] }[];
  proofsEnabled: boolean;
}

const defaultAccountCreationFee = 1_000_000_000;

/**
 * A mock Mina blockchain running locally and useful for testing.
 */
function LocalBlockchain({
  accountCreationFee = defaultAccountCreationFee as string | number,
  proofsEnabled = true,
  enforceTransactionLimits = true,
} = {}) {
  const slotTime = 3 * 60 * 1000;
  const startTime = Date.now();
  const genesisTimestamp = UInt64.from(startTime);

  const ledger = Ledger.create();

  let networkState = defaultNetworkState();

  function addAccount(publicKey: PublicKey, balance: string) {
    ledger.addAccount(Ml.fromPublicKey(publicKey), balance);
  }

  let testAccounts: {
    publicKey: PublicKey;
    privateKey: PrivateKey;
  }[] = [];

  for (let i = 0; i < 10; ++i) {
    let MINA = 10n ** 9n;
    const largeValue = 1000n * MINA;
    const k = PrivateKey.random();
    const pk = k.toPublicKey();
    addAccount(pk, largeValue.toString());
    testAccounts.push({ privateKey: k, publicKey: pk });
  }

  const events: Record<string, any> = {};
  const actions: Record<
    string,
    Record<string, { actions: string[][]; hash: string }[]>
  > = {};

  return {
    proofsEnabled,
    accountCreationFee: () => UInt64.from(accountCreationFee),
    getNetworkConstants() {
      return {
        genesisTimestamp,
        accountCreationFee: UInt64.from(accountCreationFee),
        slotTime: UInt64.from(slotTime),
      };
    },
    currentSlot() {
      return UInt32.from(
        Math.ceil((new Date().valueOf() - startTime) / slotTime)
      );
    },
    hasAccount(publicKey: PublicKey, tokenId: Field = TokenId.default) {
      return !!ledger.getAccount(
        Ml.fromPublicKey(publicKey),
        Ml.constFromField(tokenId)
      );
    },
    getAccount(
      publicKey: PublicKey,
      tokenId: Field = TokenId.default
    ): Account {
      let accountJson = ledger.getAccount(
        Ml.fromPublicKey(publicKey),
        Ml.constFromField(tokenId)
      );
      if (accountJson === undefined) {
        throw new Error(
          reportGetAccountError(publicKey.toBase58(), TokenId.toBase58(tokenId))
        );
      }
      return Types.Account.fromJSON(accountJson);
    },
    getNetworkState() {
      return networkState;
    },
    async sendTransaction(txn: Transaction): Promise<TransactionId> {
      txn.sign();
    let zkappCommandJson = ZkappCommand.toJSON(txn.transaction);
    let commitments = transactionCommitments(
      TypesBigint.ZkappCommand.fromJSON(zkappCommandJson)
    );

    if (enforceTransactionLimits) verifyTransactionLimits(txn.transaction);

    for (const update of txn.transaction.accountUpdates) {
      let accountJson = ledger.getAccount(
        Ml.fromPublicKey(update.body.publicKey),
        Ml.constFromField(update.body.tokenId)
      );

      let authIsProof = !!update.authorization.proof;
      let kindIsProof = update.body.authorizationKind.isProved.toBoolean();
      // checks and edge case where a proof is expected, but the developer forgot to invoke await tx.prove()
      // this resulted in an assertion OCaml error, which didn't contain any useful information
      if (kindIsProof && !authIsProof) {
        throw Error(
          &nbsp;The actual authorization does not match the expected authorization kind. Did you forget to
  invoke \&nbsp;await tx.prove();\&nbsp;?&nbsp;
        );
      }

      if (accountJson) {
        let account = Account.fromJSON(accountJson);

        await verifyAccountUpdate(
          account,
          update,
          commitments,
          proofsEnabled
        );
      }
    }

    try {
      ledger.applyJsonTransaction(
        JSON.stringify(zkappCommandJson),
        String(accountCreationFee),
        JSON.stringify(networkState)
      );
    } catch (err: any) {
      try {
        // reverse errors so they match order of account updates
        // TODO: label updates, and try to give precise explanations about what went wrong
        let errors = JSON.parse(err.message);
        err.message = invalidTransactionError(txn.transaction, errors, {
          accountCreationFee,
        });
      } finally {
        throw err;
    }
  }

  // fetches all events from the transaction and stores them
  // events are identified and associated with a publicKey and tokenId
  txn.transaction.accountUpdates.forEach((p, i) => {
    let pJson = zkappCommandJson.accountUpdates[i];
    let addr = pJson.body.publicKey;
    let tokenId = pJson.body.tokenId;
    events[addr] ??= {};
    if (p.body.events.data.length > 0) {
      events[addr][tokenId] ??= [];
      let updatedEvents = p.body.events.data.map((data) => {
        return {
          data,
          transactionInfo: {
            transactionHash: '',
            transactionStatus: '',
            transactionMemo: '',
          },
        };
      });
      events[addr][tokenId].push({
        events: updatedEvents,
        blockHeight: networkState.blockchainLength,
        globalSlot: networkState.globalSlotSinceGenesis,
        // The following fields are fetched from the Mina network. For now, we mock these values out
        // since networkState does not contain these fields.
        blockHash: '',
        parentBlockHash: '',
        chainStatus: '',
      });
    }

    // actions/sequencing events

    // most recent action state
    let storedActions = actions[addr]?.[tokenId];
    let latestActionState_ =
      storedActions?.[storedActions.length - 1]?.hash;
    // if there exists no hash, this means we initialize our latest hash with the empty state
    let latestActionState =
      latestActionState_ !== undefined
        ? Field(latestActionState_)
        : Actions.emptyActionState();

    actions[addr] ??= {};
    if (p.body.actions.data.length > 0) {
      let newActionState = Actions.updateSequenceState(
        latestActionState,
        p.body.actions.hash
      );
```

```
        actions[addr][tokenId] ??= [];
        actions[addr][tokenId].push({
          actions: pJson.body.actions,
          hash: newActionState.toString(),
        });
      }
    });
    return {
      isSuccess: true,
      wait: async (_options?: {
        maxAttempts?: number;
        interval?: number;
      }) => {
        console.log(
          'Info: Waiting for inclusion in a block is not supported for LocalBlockchain.'
        );
      },
      hash: (): string => {
        const message =
          'Info: Txn Hash retrieving is not supported for LocalBlockchain.';
        console.log(message);
        return message;
      },
    };
  },
  async transaction(sender: DeprecatedFeePayerSpec, f: () => void) {
    // bad hack: run transaction just to see whether it creates proofs
    // if it doesn't, this is the last chance to run SmartContract.runOutsideCircuit, which is supposed to run
only once
    // TODO: this has obvious holes if multiple zkapps are involved, but not relevant currently because we
can't prove with multiple account updates
    // and hopefully with upcoming work by Matt we can just run everything in the prover, and nowhere else
    let tx = createTransaction(sender, f, 0, {
      isFinalRunOutsideCircuit: false,
      proofsEnabled,
      fetchMode: 'test',
    });
    let hasProofs = tx.transaction.accountUpdates.some(
      Authorization.hasLazyProof
    );
    return createTransaction(sender, f, 1, {
      isFinalRunOutsideCircuit: !hasProofs,
      proofsEnabled,
    });
  },
  applyJsonTransaction(json: string) {
    return ledger.applyJsonTransaction(
      json,
      String(accountCreationFee),
      JSON.stringify(networkState)
    );
  },
async fetchEvents(publicKey: PublicKey, tokenId: Field = TokenId.default) {
  return events?.[publicKey.toBase58()]?.[TokenId.toBase58(tokenId)] ?? [];
},
async fetchActions(
  publicKey: PublicKey,
  actionStates?: ActionStates,
  tokenId: Field = TokenId.default
) {
  return this.getActions(publicKey, actionStates, tokenId);
},
getActions(
  publicKey: PublicKey,
  actionStates?: ActionStates,
  tokenId: Field = TokenId.default
): { hash: string; actions: string[][] }[] {
  let currentActions =
    actions?.[publicKey.toBase58()]?.[TokenId.toBase58(tokenId)] ?? [];
  let { fromActionState, endActionState } = actionStates ?? {};

  let emptyState = Actions.emptyActionState();
  if (endActionState?.equals(emptyState).toBoolean()) return [];

  let start = fromActionState?.equals(emptyState).toBoolean()
    ? undefined
    : fromActionState?.toString();
  let end = endActionState?.toString();

  let startIndex = 0;
  if (start) {
    let i = currentActions.findIndex((e) => e.hash === start);
    if (i === -1) throw Error(&nbsp;getActions: fromActionState not found.&nbsp;);
    startIndex = i + 1;
  }
  let endIndex: number | undefined;
  if (end) {
    let i = currentActions.findIndex((e) => e.hash === end);
    if (i === -1) throw Error(&nbsp;getActions: endActionState not found.&nbsp;);
    endIndex = i + 1;
  }
  return currentActions.slice(startIndex, endIndex);
},
addAccount,
/**
 * An array of 10 test accounts that have been pre-filled with
 * 30000000000 units of currency.
 */
testAccounts,
setGlobalSlot(slot: UInt32 | number) {
  networkState.globalSlotSinceGenesis = UInt32.from(slot);
},
incrementGlobalSlot(increment: UInt32 | number) {
  networkState.globalSlotSinceGenesis =
      networkState.globalSlotSinceGenesis.add(increment);
    },
    setBlockchainLength(height: UInt32) {
      networkState.blockchainLength = height;
    },
    setTotalCurrency(currency: UInt64) {
      networkState.totalCurrency = currency;
    },
    setProofsEnabled(newProofsEnabled: boolean) {
      proofsEnabled = newProofsEnabled;
    },
  };
}
// assert type compatibility without preventing LocalBlockchain to return additional properties / methods
LocalBlockchain satisfies (...args: any) => Mina;

/**
 * Represents the Mina blockchain running on a real network
 */
function Network(graphqlEndpoint: string): Mina;
function Network(endpoints: {
  mina: string | string[];
  archive?: string | string[];
  lightnetAccountManager?: string;
}): Mina;
function Network(
  input:
    | {
        mina: string | string[];
        archive?: string | string[];
        lightnetAccountManager?: string;
      }
    | string
): Mina {
  let accountCreationFee = UInt64.from(defaultAccountCreationFee);
  let minaGraphqlEndpoint: string;
  let archiveEndpoint: string;
  let lightnetAccountManagerEndpoint: string;

  if (input && typeof input === 'string') {
    minaGraphqlEndpoint = input;
    Fetch.setGraphqlEndpoint(minaGraphqlEndpoint);
  } else if (input && typeof input === 'object') {
    if (!input.mina)
      throw new Error(
        "Network: malformed input. Please provide an object with 'mina' endpoint."
      );
    if (Array.isArray(input.mina) && input.mina.length !== 0) {
      minaGraphqlEndpoint = input.mina[0];
      Fetch.setGraphqlEndpoint(minaGraphqlEndpoint);
      Fetch.setMinaGraphqlFallbackEndpoints(input.mina.slice(1));
    } else if (typeof input.mina === 'string') {
```

```
    minaGraphqlEndpoint = input.mina;
    Fetch.setGraphqlEndpoint(minaGraphqlEndpoint);
  }

  if (input.archive !== undefined) {
    if (Array.isArray(input.archive) && input.archive.length !== 0) {
      archiveEndpoint = input.archive[0];
      Fetch.setArchiveGraphqlEndpoint(archiveEndpoint);
      Fetch.setArchiveGraphqlFallbackEndpoints(input.archive.slice(1));
    } else if (typeof input.archive === 'string') {
      archiveEndpoint = input.archive;
      Fetch.setArchiveGraphqlEndpoint(archiveEndpoint);
    }
  }

  if (
    input.lightnetAccountManager !== undefined &&
    typeof input.lightnetAccountManager === 'string'
  ) {
    lightnetAccountManagerEndpoint = input.lightnetAccountManager;
    Fetch.setLightnetAccountManagerEndpoint(lightnetAccountManagerEndpoint);
  }
} else {
  throw new Error(
    "Network: malformed input. Please provide a string or an object with 'mina' and 'archive' endpoints."
  );
}

// copied from mina/genesis_ledgers/berkeley.json
// TODO fetch from graphql instead of hardcoding
const genesisTimestampString = '2023-02-23T20:00:01Z';
const genesisTimestamp = UInt64.from(
  Date.parse(genesisTimestampString.slice(0, -1) + '+00:00')
);
// TODO also fetch from graphql
const slotTime = UInt64.from(3 * 60 * 1000);
return {
  accountCreationFee: () => accountCreationFee,
  getNetworkConstants() {
    return {
      genesisTimestamp,
      slotTime,
      accountCreationFee,
    };
  },
  currentSlot() {
    throw Error(
      'currentSlot() is not implemented yet for remote blockchains.'
    );
  },
  hasAccount(publicKey: PublicKey, tokenId: Field = TokenId.default) {
    if (
      !currentTransaction.has() ||
      currentTransaction.get().fetchMode === 'cached'
    ) {
      return !!Fetch.getCachedAccount(
        publicKey,
        tokenId,
        minaGraphqlEndpoint
      );
    }
    return false;
  },
  getAccount(publicKey: PublicKey, tokenId: Field = TokenId.default) {
    if (currentTransaction()?.fetchMode === 'test') {
      Fetch.markAccountToBeFetched(publicKey, tokenId, minaGraphqlEndpoint);
      let account = Fetch.getCachedAccount(
        publicKey,
        tokenId,
        minaGraphqlEndpoint
      );
      return account ?? dummyAccount(publicKey);
    }
    if (
      !currentTransaction.has() ||
      currentTransaction.get().fetchMode === 'cached'
    ) {
      let account = Fetch.getCachedAccount(
        publicKey,
        tokenId,
        minaGraphqlEndpoint
      );
      if (account !== undefined) return account;
    }
    throw Error(
      &nbsp;${reportGetAccountError(
        publicKey.toBase58(),
        TokenId.toBase58(tokenId)
      )}\nGraphql endpoint: ${minaGraphqlEndpoint}&nbsp;
    );
  },
  getNetworkState() {
    if (currentTransaction()?.fetchMode === 'test') {
      Fetch.markNetworkToBeFetched(minaGraphqlEndpoint);
      let network = Fetch.getCachedNetwork(minaGraphqlEndpoint);
      return network ?? defaultNetworkState();
    }
    if (
      !currentTransaction.has() ||
      currentTransaction.get().fetchMode === 'cached'
    ) {
      let network = Fetch.getCachedNetwork(minaGraphqlEndpoint);
      if (network !== undefined) return network;
    }
```

```
    throw Error(
      &nbsp;getNetworkState: Could not fetch network state from graphql endpoint
${minaGraphqlEndpoint}&nbsp;
    );
  },
  async sendTransaction(txn: Transaction) {
    txn.sign();

    verifyTransactionLimits(txn.transaction);

    let [response, error] = await Fetch.sendZkapp(txn.toJSON());
    let errors: any[] | undefined;
    if (response === undefined && error !== undefined) {
      console.log('Error: Failed to send transaction', error);
      errors = [error];
    } else if (response && response.errors && response.errors.length > 0) {
      console.log(
        'Error: Transaction returned with errors',
        JSON.stringify(response.errors, null, 2)
      );
      errors = response.errors;
    }

    let isSuccess = errors === undefined;
    let maxAttempts: number;
    let attempts = 0;
    let interval: number;

    return {
      isSuccess,
      data: response?.data,
      errors,
      async wait(options?: { maxAttempts?: number; interval?: number }) {
        if (!isSuccess) {
          console.warn(
            'Transaction.wait(): returning immediately because the transaction was not successful.'
          );
          return;
        }
        // default is 45 attempts * 20s each = 15min
        // the block time on berkeley is currently longer than the average 3-4min, so its better to target a
higher block time
        // fetching an update every 20s is more than enough with a current block time of 3min
        maxAttempts = options?.maxAttempts ?? 45;
        interval = options?.interval ?? 20000;

        const executePoll = async (
          resolve: () => void,
          reject: (err: Error) => void | Error
        ) => {
          let txId = response?.data?.sendZkapp?.zkapp?.hash;
          let res;
          try {
            res = await Fetch.checkZkappTransaction(txId);
          } catch (error) {
            isSuccess = false;
            return reject(error as Error);
          }
          attempts++;
          if (res.success) {
            isSuccess = true;
            return resolve();
          } else if (res.failureReason) {
            isSuccess = false;
            return reject(
              new Error(
                &nbsp;Transaction failed.\nTransactionId: ${txId}\nAttempts: ${attempts}\nfailureReason(s):
${res.failureReason}&nbsp;
              )
            );
          } else if (maxAttempts && attempts === maxAttempts) {
            isSuccess = false;
            return reject(
              new Error(
                &nbsp;Exceeded max attempts.\nTransactionId: ${txId}\nAttempts: ${attempts}\nLast received
status: ${res}&nbsp;
              )
            );
          } else {
            setTimeout(executePoll, interval, resolve, reject);
          }
        };

        return new Promise(executePoll);
      },
      hash() {
        return response?.data?.sendZkapp?.zkapp?.hash;
      },
    };
  },
  async transaction(sender: DeprecatedFeePayerSpec, f: () => void) {
    let tx = createTransaction(sender, f, 0, {
      fetchMode: 'test',
      isFinalRunOutsideCircuit: false,
    });
    await Fetch.fetchMissingData(minaGraphqlEndpoint, archiveEndpoint);
    let hasProofs = tx.transaction.accountUpdates.some(
      Authorization.hasLazyProof
    );
    return createTransaction(sender, f, 1, {
      fetchMode: 'cached',
      isFinalRunOutsideCircuit: !hasProofs,
    });
  },
  async fetchEvents(
    publicKey: PublicKey,
    tokenId: Field = TokenId.default,
    filterOptions: Fetch.EventActionFilterOptions = {}
  ) {
    let pubKey = publicKey.toBase58();
    let token = TokenId.toBase58(tokenId);

    return Fetch.fetchEvents(
      { publicKey: pubKey, tokenId: token },
      archiveEndpoint,
      filterOptions
    );
  },
  async fetchActions(
    publicKey: PublicKey,
    actionStates?: ActionStates,
    tokenId: Field = TokenId.default
  ) {
    let pubKey = publicKey.toBase58();
    let token = TokenId.toBase58(tokenId);
    let { fromActionState, endActionState } = actionStates ?? {};
    let fromActionStateBase58 = fromActionState
      ? fromActionState.toString()
      : undefined;
    let endActionStateBase58 = endActionState
      ? endActionState.toString()
      : undefined;

    return Fetch.fetchActions(
      {
        publicKey: pubKey,
        actionStates: {
          fromActionState: fromActionStateBase58,
          endActionState: endActionStateBase58,
        },
        tokenId: token,
      },
      archiveEndpoint
    );
  },
  getActions(
    publicKey: PublicKey,
    actionStates?: ActionStates,
    tokenId: Field = TokenId.default
  ) {
    if (currentTransaction()?.fetchMode === 'test') {
      Fetch.markActionsToBeFetched(
        publicKey,
        tokenId,
        archiveEndpoint,
        actionStates
      );
      let actions = Fetch.getCachedActions(publicKey, tokenId);
      return actions ?? [];
    }
    if (
      !currentTransaction.has() ||
      currentTransaction.get().fetchMode === 'cached'
    ) {
      let actions = Fetch.getCachedActions(publicKey, tokenId);
      if (actions !== undefined) return actions;
    }
    throw Error(
      &nbsp;getActions: Could not find actions for the public key ${publicKey}&nbsp;
    );
  },
  proofsEnabled: true,
 };
}

/**
 *
 * @deprecated This is deprecated in favor of {@link Mina.Network}, which is exactly the same function.
 * The name &nbsp;BerkeleyQANet&nbsp; was misleading because it suggested that this is specific to a
particular network.
 */
function BerkeleyQANet(graphqlEndpoint: string) {
 return Network(graphqlEndpoint);
}

let activeInstance: Mina = {
 accountCreationFee: () => UInt64.from(defaultAccountCreationFee),
 getNetworkConstants() {
  throw new Error('must call Mina.setActiveInstance first');
 },
 currentSlot: () => {
  throw new Error('must call Mina.setActiveInstance first');
 },
 hasAccount(publicKey: PublicKey, tokenId: Field = TokenId.default) {
  if (
   !currentTransaction.has() ||
   currentTransaction.get().fetchMode === 'cached'
  ) {
   return !!Fetch.getCachedAccount(
    publicKey,
    tokenId,
    Fetch.networkConfig.minaEndpoint
   );
  }
  return false;
 },
 getAccount(publicKey: PublicKey, tokenId: Field = TokenId.default) {
  if (currentTransaction()?.fetchMode === 'test') {
      Fetch.markAccountToBeFetched(
        publicKey,
        tokenId,
        Fetch.networkConfig.minaEndpoint
      );
      return dummyAccount(publicKey);
    }
    if (
      !currentTransaction.has() ||
      currentTransaction.get().fetchMode === 'cached'
    ) {
      let account = Fetch.getCachedAccount(
        publicKey,
        tokenId,
        Fetch.networkConfig.minaEndpoint
      );
      if (account === undefined)
        throw Error(
          &nbsp;${reportGetAccountError(
            publicKey.toBase58(),
            TokenId.toBase58(tokenId)
          )}\n\nEither call Mina.setActiveInstance first or explicitly add the account with
addCachedAccount&nbsp;
        );
      return account;
    }
    throw new Error('must call Mina.setActiveInstance first');
  },
  getNetworkState() {
    throw new Error('must call Mina.setActiveInstance first');
  },
  sendTransaction() {
    throw new Error('must call Mina.setActiveInstance first');
  },
  async transaction(sender: DeprecatedFeePayerSpec, f: () => void) {
    return createTransaction(sender, f, 0);
  },
  fetchEvents(_publicKey: PublicKey, _tokenId: Field = TokenId.default) {
    throw Error('must call Mina.setActiveInstance first');
  },
  fetchActions(
    _publicKey: PublicKey,
    _actionStates?: ActionStates,
    _tokenId: Field = TokenId.default
  ) {
    throw Error('must call Mina.setActiveInstance first');
  },
  getActions(
    _publicKey: PublicKey,
    _actionStates?: ActionStates,
    _tokenId: Field = TokenId.default
  ) {
    throw Error('must call Mina.setActiveInstance first');
  },
  proofsEnabled: true,
};

/**
 * Set the currently used Mina instance.
 */
function setActiveInstance(m: Mina) {
  activeInstance = m;
}

/**
 * Construct a smart contract transaction. Within the callback passed to this function,
 * you can call into the methods of smart contracts.
 *
 * &nbsp;&nbsp;&nbsp;
 * let tx = await Mina.transaction(sender, () => {
 *   myZkapp.update();
 *   someOtherZkapp.someOtherMethod();
 * });
 * &nbsp;&nbsp;&nbsp;
 *
 * @return A transaction that can subsequently be submitted to the chain.
 */
function transaction(sender: FeePayerSpec, f: () => void): Promise<Transaction>;
function transaction(f: () => void): Promise<Transaction>;
/**
 * @deprecated It's deprecated to pass in the fee payer's private key. Pass in the public key instead.
 * &nbsp;&nbsp;&nbsp;
 * // good
 * Mina.transaction(publicKey, ...);
 * Mina.transaction({ sender: publicKey }, ...);
 *
 * // deprecated
 * Mina.transaction(privateKey, ...);
 * Mina.transaction({ feePayerKey: privateKey }, ...);
 * &nbsp;&nbsp;&nbsp;
 */
function transaction(
  sender: DeprecatedFeePayerSpec,
  f: () => void
): Promise<Transaction>;
function transaction(
  senderOrF: DeprecatedFeePayerSpec | (() => void),
  fOrUndefined?: () => void
): Promise<Transaction> {
  let sender: DeprecatedFeePayerSpec;
  let f: () => void;
  try {
    if (fOrUndefined !== undefined) {
      sender = senderOrF as DeprecatedFeePayerSpec;
    f = fOrUndefined;
  } else {
    sender = undefined;
    f = senderOrF as () => void;
  }
  return activeInstance.transaction(sender, f);
 } catch (error) {
  throw prettifyStacktrace(error);
 }
}

/**
 * Returns the public key of the current transaction's sender account.
 *
 * Throws an error if not inside a transaction, or the sender wasn't passed in.
 */
function sender() {
 let tx = currentTransaction();
 if (tx === undefined)
  throw Error(
    &nbsp;The sender is not available outside a transaction. Make sure you only use it within
\&nbsp;Mina.transaction\&nbsp; blocks or smart contract methods.&nbsp;
  );
 let sender = currentTransaction()?.sender;
 if (sender === undefined)
  throw Error(
    &nbsp;The sender is not available, because the transaction block was created without the optional
\&nbsp;sender\&nbsp; argument.
Here's an example for how to pass in the sender and make it available:

Mina.transaction(sender, // <-- pass in sender's public key here
() => {
 // methods can use this.sender
});
&nbsp;
  );
 return sender;
}

/**
 * @return The current slot number, according to the active Mina instance.
 */
function currentSlot(): UInt32 {
 return activeInstance.currentSlot();
}

/**
 * @return The account data associated to the given public key.
 */
function getAccount(publicKey: PublicKey, tokenId?: Field): Account {
 return activeInstance.getAccount(publicKey, tokenId);
}
```

```
/**
 * Checks if an account exists within the ledger.
 */
function hasAccount(publicKey: PublicKey, tokenId?: Field): boolean {
  return activeInstance.hasAccount(publicKey, tokenId);
}

/**
 * @return Data associated with the current state of the Mina network.
 */
function getNetworkState() {
  return activeInstance.getNetworkState();
}

/**
 * @return The balance associated to the given public key.
 */
function getBalance(publicKey: PublicKey, tokenId?: Field) {
  return activeInstance.getAccount(publicKey, tokenId).balance;
}

/**
 * Returns the default account creation fee.
 */
function accountCreationFee() {
  return activeInstance.accountCreationFee();
}

async function sendTransaction(txn: Transaction) {
  return await activeInstance.sendTransaction(txn);
}

/**
 * @return A list of emitted events associated to the given public key.
 */
async function fetchEvents(
  publicKey: PublicKey,
  tokenId: Field,
  filterOptions: Fetch.EventActionFilterOptions = {}
) {
  return await activeInstance.fetchEvents(publicKey, tokenId, filterOptions);
}

/**
 * @return A list of emitted sequencing actions associated to the given public key.
 */
async function fetchActions(
  publicKey: PublicKey,
  actionStates?: ActionStates,
  tokenId?: Field
) {
  return await activeInstance.fetchActions(publicKey, actionStates, tokenId);
}

/**
 * @return A list of emitted sequencing actions associated to the given public key.
 */
function getActions(
  publicKey: PublicKey,
  actionStates?: ActionStates,
  tokenId?: Field
) {
  return activeInstance.getActions(publicKey, actionStates, tokenId);
}

function getProofsEnabled() {
  return activeInstance.proofsEnabled;
}

function dummyAccount(pubkey?: PublicKey): Account {
  let dummy = Types.Account.emptyValue();
  if (pubkey) dummy.publicKey = pubkey;
  return dummy;
}

function defaultNetworkState(): NetworkValue {
  let epochData: NetworkValue['stakingEpochData'] = {
    ledger: { hash: Field(0), totalCurrency: UInt64.zero },
    seed: Field(0),
    startCheckpoint: Field(0),
    lockCheckpoint: Field(0),
    epochLength: UInt32.zero,
  };
  return {
    snarkedLedgerHash: Field(0),
    blockchainLength: UInt32.zero,
    minWindowDensity: UInt32.zero,
    totalCurrency: UInt64.zero,
    globalSlotSinceGenesis: UInt32.zero,
    stakingEpochData: epochData,
    nextEpochData: cloneCircuitValue(epochData),
  };
}

async function verifyAccountUpdate(
  account: Account,
  accountUpdate: AccountUpdate,
  transactionCommitments: { commitment: bigint; fullCommitment: bigint },
  proofsEnabled: boolean
): Promise<void> {
  // check that that top-level updates have mayUseToken = No
  // (equivalent check exists in the Mina node)
  if (
    accountUpdate.body.callDepth === 0 &&
    !AccountUpdate.MayUseToken.isNo(accountUpdate).toBoolean()
  ) {
    throw Error(
      'Top-level account update can not use or pass on token permissions. Make sure that\n' +
        'accountUpdate.body.mayUseToken = AccountUpdate.MayUseToken.No;'
    );
  }

  let perm = account.permissions;

  // check if addMissingSignatures failed to include a signature
  // due to a missing private key
  if (accountUpdate.authorization === dummySignature()) {
    let pk = PublicKey.toBase58(accountUpdate.body.publicKey);
    throw Error(
      &nbsp;verifyAccountUpdate: Detected a missing signature for (${pk}), private key was missing.&nbsp;
    );
  }
  // we are essentially only checking if the update is empty or an actual update
  function includesChange<T extends {}>(
    val: T | string | null | (string | null)[]
  ): boolean {
    if (Array.isArray(val)) {
      return !val.every((v) => v === null);
    } else {
      return val !== null;
    }
  }

  function permissionForUpdate(key: string): Types.AuthRequired {
    switch (key) {
      case 'appState':
        return perm.editState;
      case 'delegate':
        return perm.setDelegate;
      case 'verificationKey':
        return perm.setVerificationKey;
      case 'permissions':
        return perm.setPermissions;
      case 'zkappUri':
        return perm.setZkappUri;
      case 'tokenSymbol':
        return perm.setTokenSymbol;
      case 'timing':
        return perm.setTiming;
      case 'votingFor':
        return perm.setVotingFor;
      case 'actions':
        return perm.editActionState;
      case 'incrementNonce':
        return perm.incrementNonce;
      case 'send':
        return perm.send;
      case 'receive':
        return perm.receive;
      default:
        throw Error(&nbsp;Invalid permission for field ${key}: does not exist.&nbsp;);
    }
  }

  let accountUpdateJson = accountUpdate.toJSON();
  const update = accountUpdateJson.body.update;

  let errorTrace = '';

  let isValidProof = false;
  let isValidSignature = false;

  // we don't check if proofs aren't enabled
  if (!proofsEnabled) isValidProof = true;

  if (accountUpdate.authorization.proof && proofsEnabled) {
    try {
      let publicInput = accountUpdate.toPublicInput();
      let publicInputFields = ZkappPublicInput.toFields(publicInput);

      const proof = SmartContract.Proof().fromJSON({
        maxProofsVerified: 2,
        proof: accountUpdate.authorization.proof!,
        publicInput: publicInputFields.map((f) => f.toString()),
        publicOutput: [],
      });

      let verificationKey = account.zkapp?.verificationKey?.data!;
      isValidProof = await verify(proof.toJSON(), verificationKey);
      if (!isValidProof) {
        throw Error(
          &nbsp;Invalid proof for account update\n${JSON.stringify(update)}&nbsp;
        );
      }
    } catch (error) {
      errorTrace += '\n\n' + (error as Error).message;
      isValidProof = false;
    }
  }

  if (accountUpdate.authorization.signature) {
    // checking permissions and authorization for each account update individually
    try {
      isValidSignature = verifyAccountUpdateSignature(
        TypesBigint.AccountUpdate.fromJSON(accountUpdateJson),
        transactionCommitments,
        'testnet'
      );
    } catch (error) {
      errorTrace += '\n\n' + (error as Error).message;
      isValidSignature = false;
    }
  }

  let verified = false;

  function checkPermission(p0: Types.AuthRequired, field: string) {
    let p = Types.AuthRequired.toJSON(p0);
    if (p === 'None') return;

    if (p === 'Impossible') {
      throw Error(
        &nbsp;Transaction verification failed: Cannot update field '${field}' because permission for this field is
  '${p}'&nbsp;
      );
    }

    if (p === 'Signature' || p === 'Either') {
      verified ||= isValidSignature;
    }

    if (p === 'Proof' || p === 'Either') {
      verified ||= isValidProof;
    }

    if (!verified) {
      throw Error(
        &nbsp;Transaction verification failed: Cannot update field '${field}' because permission for this field is
  '${p}', but the required authorization was not provided or is invalid.
        ${errorTrace !== '' ? 'Error trace: ' + errorTrace : ''}&nbsp;
      );
    }
  }

  // goes through the update field on a transaction
  Object.entries(update).forEach(([key, value]) => {
    if (includesChange(value)) {
      let p = permissionForUpdate(key);
      checkPermission(p, key);
    }
  });

  // checks the sequence events (which result in an updated sequence state)
  if (accountUpdate.body.actions.data.length > 0) {
    let p = permissionForUpdate('actions');
    checkPermission(p, 'actions');
  }

  if (accountUpdate.body.incrementNonce.toBoolean()) {
    let p = permissionForUpdate('incrementNonce');
    checkPermission(p, 'incrementNonce');
  }

  // this checks for an edge case where an account update can be approved using proofs but
  // a) the proof is invalid (bad verification key)
  // and b) there are no state changes initiate so no permissions will be checked
  // however, if the verification key changes, the proof should still be invalid
  if (errorTrace && !verified) {
    throw Error(
      &nbsp;One or more proofs were invalid and no other form of authorization was
provided.\n${errorTrace}&nbsp;
    );
  }
}

function verifyTransactionLimits({ accountUpdates }: ZkappCommand) {
  let eventElements = { events: 0, actions: 0 };

  let authKinds = accountUpdates.map((update) => {
    eventElements.events += countEventElements(update.body.events);
    eventElements.actions += countEventElements(update.body.actions);
    let { isSigned, isProved, verificationKeyHash } =
      update.body.authorizationKind;
    return {
      isSigned: isSigned.toBoolean(),
      isProved: isProved.toBoolean(),
      verificationKeyHash: verificationKeyHash.toString(),
    };
  });
  // insert entry for the fee payer
  authKinds.unshift({
    isSigned: true,
    isProved: false,
    verificationKeyHash: '',
  });
  let authTypes = filterGroups(authKinds);

  /*
  np := proof
  n2 := signedPair
  n1 := signedSingle

  formula used to calculate how expensive a zkapp transaction is

  10.26*np + 10.08*n2 + 9.14*n1 < 69.45
  */
  let totalTimeRequired =
    TransactionCost.PROOF_COST * authTypes.proof +
    TransactionCost.SIGNED_PAIR_COST * authTypes.signedPair +
    TransactionCost.SIGNED_SINGLE_COST * authTypes.signedSingle;
  let isWithinCostLimit = totalTimeRequired < TransactionCost.COST_LIMIT;

  let isWithinEventsLimit =
    eventElements.events <= TransactionLimits.MAX_EVENT_ELEMENTS;
  let isWithinActionsLimit =
    eventElements.actions <= TransactionLimits.MAX_ACTION_ELEMENTS;

  let error = '';

  if (!isWithinCostLimit) {
    // TODO: we should add a link to the docs explaining the reasoning behind it once we have such an
explainer
    error += &nbsp;Error: The transaction is too expensive, try reducing the number of AccountUpdates that
are attached to the transaction.
Each transaction needs to be processed by the snark workers on the network.
Certain layouts of AccountUpdates require more proving time than others, and therefore are too expensive.

${JSON.stringify(authTypes)}
\n\n&nbsp;;
  }

  if (!isWithinEventsLimit) {
    error += &nbsp;Error: The account updates in your transaction are trying to emit too much event data. The
maximum allowed number of field elements in events is ${TransactionLimits.MAX_EVENT_ELEMENTS},
but you tried to emit ${eventElements.events}.\n\n&nbsp;;
  }

  if (!isWithinActionsLimit) {
    error += &nbsp;Error: The account updates in your transaction are trying to emit too much action data.
The maximum allowed number of field elements in actions is
${TransactionLimits.MAX_ACTION_ELEMENTS}, but you tried to emit
${eventElements.actions}.\n\n&nbsp;;
  }

  if (error) throw Error('Error during transaction sending:\n\n' + error);
}

function countEventElements({ data }: Events) {
  return data.reduce((acc, ev) => acc + ev.length, 0);
}

type AuthorizationKind = { isProved: boolean; isSigned: boolean };

const isPair = (a: AuthorizationKind, b: AuthorizationKind) =>
  !a.isProved && !b.isProved;

function filterPairs(xs: AuthorizationKind[]): {
  xs: { isProved: boolean; isSigned: boolean }[];
  pairs: number;
} {
  if (xs.length <= 1) return { xs, pairs: 0 };
  if (isPair(xs[0], xs[1])) {
    let rec = filterPairs(xs.slice(2));
    return { xs: rec.xs, pairs: rec.pairs + 1 };
  } else {
    let rec = filterPairs(xs.slice(1));
    return { xs: [xs[0]].concat(rec.xs), pairs: rec.pairs };
  }
}

function filterGroups(xs: AuthorizationKind[]) {
  let pairs = filterPairs(xs);
  xs = pairs.xs;

  let singleCount = 0;
  let proofCount = 0;

  xs.forEach((t) => {
    if (t.isProved) proofCount++;
    else singleCount++;
  });

  return {
    signedPair: pairs.pairs,
    signedSingle: singleCount,
    proof: proofCount,
  };
}

async function waitForFunding(address: string): Promise<void> {
  let attempts = 0;
  let maxAttempts = 30;
  let interval = 30000;
  const executePoll = async (
    resolve: () => void,
    reject: (err: Error) => void | Error
  ) => {
    let { account } = await Fetch.fetchAccount({ publicKey: address });
    attempts++;
    if (account) {
      return resolve();
    } else if (maxAttempts && attempts === maxAttempts) {
      return reject(new Error(&nbsp;Exceeded max attempts&nbsp;));
    } else {
      setTimeout(executePoll, interval, resolve, reject);
    }
  };
  return new Promise(executePoll);
}

/**
 * Requests the [testnet faucet](https://faucet.minaprotocol.com/api/v1/faucet) to fund a public key.
 */
async function faucet(pub: PublicKey, network: string = 'berkeley-qanet') {
  let address = pub.toBase58();
  let response = await fetch('https://faucet.minaprotocol.com/api/v1/faucet', {
    method: 'POST',
    headers: { 'Content-Type': 'application/json' },
    body: JSON.stringify({
      network,
      address: address,
    }),
  });
  response = await response.json();
  if (response.status.toString() !== 'success') {
    throw new Error(
      &nbsp;Error funding account ${address}, got response status: ${response.status}, text:
${response.statusText}&nbsp;
    );
  }
  await waitForFunding(address);
}
```

</file>

<file>

## path: /src/lib/mina.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/mina.unit-test.ts

```
import { filterGroups } from './mina.js';
import { expect } from 'expect';
import { shutdown } from '../index.js';

let S = { isProved: false, isSigned: true };
let N = { isProved: false, isSigned: false };
let P = { isProved: true, isSigned: false };

expect(filterGroups([S, S, S, S, S, S])).toEqual({
  proof: 0,
  signedPair: 3,
  signedSingle: 0,
});

expect(filterGroups([N, N, N, N, N, N])).toEqual({
  proof: 0,
  signedPair: 3,
  signedSingle: 0,
});

expect(filterGroups([N, S, S, N, N, S])).toEqual({
  proof: 0,
  signedPair: 3,
  signedSingle: 0,
});

expect(filterGroups([S, P, S, S, S, S])).toEqual({
  proof: 1,
  signedPair: 2,
  signedSingle: 1,
});

expect(filterGroups([N, P, N, P, N, P])).toEqual({
  proof: 3,
  signedPair: 0,
  signedSingle: 3,
});

expect(filterGroups([N, P])).toEqual({
  proof: 1,
  signedPair: 0,
  signedSingle: 1,
});

expect(filterGroups([N, S])).toEqual({
  proof: 0,
  signedPair: 1,
  signedSingle: 0,
});

expect(filterGroups([P, P])).toEqual({
  proof: 2,
  signedPair: 0,
  signedSingle: 0,
});

expect(filterGroups([P, P, S, N, N])).toEqual({
  proof: 2,
  signedPair: 1,
  signedSingle: 1,
});

expect(filterGroups([P])).toEqual({
  proof: 1,
  signedPair: 0,
  signedSingle: 0,
});

expect(filterGroups([S])).toEqual({
  proof: 0,
  signedPair: 0,
  signedSingle: 1,
});

expect(filterGroups([N])).toEqual({
  proof: 0,
  signedPair: 0,
  signedSingle: 1,
});

expect(filterGroups([N, N])).toEqual({
  proof: 0,
  signedPair: 1,
  signedSingle: 0,
});

expect(filterGroups([N, S])).toEqual({
  proof: 0,
  signedPair: 1,
  signedSingle: 0,
});

expect(filterGroups([S, N])).toEqual({
  proof: 0,
  signedPair: 1,
  signedSingle: 0,
});

expect(filterGroups([S, S])).toEqual({
  proof: 0,
  signedPair: 1,
  signedSingle: 0,
});
shutdown();
```

</file>

<file>

## path: /src/lib/ml/base.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/ml/base.ts

```
import { TupleN } from '../util/types.js';

/**
 * This module contains basic methods for interacting with OCaml
 */
export {
  MlArray,
  MlPair,
  MlList,
  MlOption,
  MlBool,
  MlBytes,
  MlResult,
  MlUnit,
  MlString,
  MlTuple,
};

// ocaml types

type MlPair<X, Y> = [0, X, Y];
type MlArray<T> = [0, ...T[]];
type MlList<T> = [0, T, 0 | MlList<T>];
type MlOption<T> = 0 | [0, T];
type MlBool = 0 | 1;
type MlResult<T, E> = [0, T] | [1, E];
type MlUnit = 0;

/**
 * js_of_ocaml representation of a byte array,
 * see https://github.com/ocsigen/js_of_ocaml/blob/master/runtime/mlBytes.js
 */
type MlBytes = { t: number; c: string; l: number };
type MlString = MlBytes;

const MlArray = {
  to<T>(arr: T[]): MlArray<T> {
    return [0, ...arr];
  },
  from<T>([, ...arr]: MlArray<T>): T[] {
    return arr;
  },
  map<T, S>([, ...arr]: MlArray<T>, map: (t: T) => S): MlArray<S> {
    return [0, ...arr.map(map)];
  },
  mapTo<T, S>(arr: T[], map: (t: T) => S): MlArray<S> {
    return [0, ...arr.map(map)];
  },
  mapFrom<T, S>([, ...arr]: MlArray<T>, map: (t: T) => S): S[] {
    return arr.map(map);
  },
};

const MlPair = Object.assign(
  function MlTuple<X, Y>(x: X, y: Y): MlPair<X, Y> {
    return [0, x, y];
  },
  {
    from<X, Y>([, x, y]: MlPair<X, Y>): [X, Y] {
      return [x, y];
    },
    first<X>(t: MlPair<X, unknown>): X {
      return t[1];
    },
    second<Y>(t: MlPair<unknown, Y>): Y {
      return t[2];
    },
  }
);

const MlBool = Object.assign(
  function MlBool(b: boolean): MlBool {
    return b ? 1 : 0;
  },
  {
    from(b: MlBool) {
      return !!b;
    },
  }
);

const MlOption = Object.assign(
  function MlOption<T>(x?: T): MlOption<T> {
    return x === undefined ? 0 : [0, x];
  },
  {
    from<T>(option: MlOption<T>): T | undefined {
      return option === 0 ? undefined : option[1];
    },
    map<T, S>(option: MlOption<T>, map: (t: T) => S): MlOption<S> {
      if (option === 0) return 0;
      return [0, map(option[1])];
    },
    mapFrom<T, S>(option: MlOption<T>, map: (t: T) => S): S | undefined {
      if (option === 0) return undefined;
      return map(option[1]);
    },
    mapTo<T, S>(option: T | undefined, map: (t: T) => S): MlOption<S> {
      if (option === undefined) return 0;
      return [0, map(option)];
    },
    isNone(option: MlOption<unknown>): option is 0 {
      return option === 0;
    },
    isSome<T>(option: MlOption<T>): option is [0, T] {
      return option !== 0;
    },
  }
);

const MlResult = {
  ok<T, E>(t: T): MlResult<T, E> {
    return [0, t];
  },
  unitError<T>(): MlResult<T, 0> {
    return [1, 0];
  },
};
```

```
/**
 * tuple type that has the length as generic parameter
 */
type MlTuple<T, N extends number> = N extends N
  ? number extends N
    ? [0, ...T[]] // N is not typed as a constant => fall back to array
    : [0, ...TupleRec<T, N, []>]
  : never;

type TupleRec<T, N extends number, R extends unknown[]> = R['length'] extends N
  ? R
  : TupleRec<T, N, [T, ...R]>;

type Tuple<T> = [T, ...T[]] | [];

const MlTuple = {
  map<T extends Tuple<any>, B>(
    [, ...mlTuple]: [0, ...T],
    f: (a: T[number]) => B
  ): [0, ...{ [i in keyof T]: B }] {
    return [0, ...mlTuple.map(f)] as any;
  },

  mapFrom<T, N extends number, B>(
    [, ...mlTuple]: MlTuple<T, N>,
    f: (a: T) => B
  ): B[] {
    return mlTuple.map(f);
  },

  mapTo<T extends Tuple<any> | TupleN<any, any>, B>(
    tuple: T,
    f: (a: T[number]) => B
  ): [0, ...{ [i in keyof T]: B }] {
    return [0, ...tuple.map(f)] as any;
  },
};
```

</file>

<file>

## path: /src/lib/ml/consistency.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/ml/consistency.unit-test.ts

```
import { Ledger, Test } from '../../snarky.js';
import { Random, test } from '../testing/property.js';
import { Field, Bool } from '../core.js';
import { PrivateKey, PublicKey } from '../signature.js';
import { TokenId, dummySignature } from '../account_update.js';
import { Ml } from './conversion.js';
import { expect } from 'expect';
import { FieldConst } from '../field.js';
import { Provable } from '../provable.js';

// PrivateKey.toBase58, fromBase58

test(Random.privateKey, (s) => {
  // private key to/from bigint
  let sk = PrivateKey.fromBigInt(s);
  expect(sk.toBigInt()).toEqual(s);

  let skMl = Ml.fromPrivateKey(sk);

  // toBase58 - check consistency with ml
  let ml = Test.encoding.privateKeyToBase58(skMl);
  let js = sk.toBase58();
  expect(js).toEqual(ml);

  // fromBase58 - check consistency with where we started
  expect(PrivateKey.fromBase58(js)).toEqual(sk);
  expect(Test.encoding.privateKeyOfBase58(ml)).toEqual(skMl);
});

// PublicKey.toBase58, fromBase58

test(Random.publicKey, (pk0) => {
  // public key from bigint
  let pk = PublicKey.from({ x: Field(pk0.x), isOdd: Bool(!!pk0.isOdd) });
  let pkMl = Ml.fromPublicKey(pk);

  // toBase58 - check consistency with ml
  let ml = Test.encoding.publicKeyToBase58(pkMl);
  let js = pk.toBase58();
  expect(js).toEqual(ml);

  // fromBase58 - check consistency with where we started
  expect(PublicKey.fromBase58(js)).toEqual(pk);
  expect(Test.encoding.publicKeyOfBase58(ml)).toEqual(pkMl);
});

// dummy signature
let js = dummySignature();
let ml = Test.signature.dummySignature();
expect(js).toEqual(ml);

// token id to/from base58

test(Random.field, (x) => {
  let js = TokenId.toBase58(Field(x));
  let ml = Test.encoding.tokenIdToBase58(FieldConst.fromBigint(x));
  expect(js).toEqual(ml);

  expect(TokenId.fromBase58(js).toBigInt()).toEqual(x);
});

let defaultTokenId = 'wSHV2S4qX9jFsLjQo8r1BsMLH2ZRKsZx6EJd1sbozGPieEC4Jf';
expect(TokenId.fromBase58(defaultTokenId).toString()).toEqual('1');

// derive token id

let randomTokenId = Random.oneOf(Random.field, TokenId.default.toBigInt());

// - non provable

test(Random.publicKey, randomTokenId, (publicKey, field) => {
  let tokenOwner = PublicKey.from({
    x: Field(publicKey.x),
    isOdd: Bool(!!publicKey.isOdd),
  });
  let parentTokenId = Field(field);

  let js = TokenId.derive(tokenOwner, parentTokenId);
  let ml = Field(
    Test.tokenId.derive(
      Ml.fromPublicKey(tokenOwner),
      Ml.constFromField(parentTokenId)
    )
  );
  expect(js).toEqual(ml);
});

// - provable

test(Random.publicKey, randomTokenId, (publicKey, field) => {
  let tokenOwner = PublicKey.from({
    x: Field(publicKey.x),
    isOdd: Bool(!!publicKey.isOdd),
  });
  let parentTokenId = Field(field);

  Provable.runAndCheck(() => {
    tokenOwner = Provable.witness(PublicKey, () => tokenOwner);
    parentTokenId = Provable.witness(Field, () => parentTokenId);

    let js = TokenId.derive(tokenOwner, parentTokenId);
    let ml = Field(
      Test.tokenId.deriveChecked(
        Ml.fromPublicKeyVar(tokenOwner),
        Ml.varFromField(parentTokenId)
      )
    );
    expect(js.isConstant()).toEqual(false);
    expect(ml.isConstant()).toEqual(false);
    Provable.asProver(() => expect(js.toBigInt()).toEqual(ml.toBigInt()));
  });

  expect(js).toEqual(ml);
});
```

</file>

<file>

## path: /src/lib/ml/conversion.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/ml/conversion.ts

```
/**
 * this file contains conversion functions between JS and OCaml
 */

import type { MlPublicKey, MlPublicKeyVar } from '../../snarky.js';
import { HashInput } from '../circuit_value.js';
import { Bool, Field } from '../core.js';
import { FieldConst, FieldVar } from '../field.js';
import { Scalar, ScalarConst } from '../scalar.js';
import { PrivateKey, PublicKey } from '../signature.js';
import { MlPair, MlBool, MlArray } from './base.js';
import { MlFieldConstArray } from './fields.js';

export { Ml, MlHashInput };

const Ml = {
  constFromField,
  constToField,
  varFromField,
  varToField,

  fromScalar,
  toScalar,

  fromPrivateKey,
  toPrivateKey,

  fromPublicKey,
  toPublicKey,

  fromPublicKeyVar,
  toPublicKeyVar,
};

type MlHashInput = [
  flag: 0,
  field_elements: MlArray<FieldConst>,
  packed: MlArray<MlPair<FieldConst, number>>
];

const MlHashInput = {
  to({ fields = [], packed = [] }: HashInput): MlHashInput {
    return [
      0,
      MlFieldConstArray.to(fields),
      MlArray.to(
        packed.map(([field, size]) => [0, Ml.constFromField(field), size])
      ),
    ];
  },
  from([, fields, packed]: MlHashInput): HashInput {
    return {
      fields: MlFieldConstArray.from(fields),
      packed: MlArray.from(packed).map(
        ([, field, size]) => [Field(field), size] as [Field, number]
      ),
    };
  },
};

function constFromField(x: Field): FieldConst {
  return x.toConstant().value[1];
}
function constToField(x: FieldConst): Field {
  return Field(x);
}
function varFromField(x: Field): FieldVar {
  return x.value;
}
function varToField(x: FieldVar): Field {
  return Field(x);
}

function fromScalar(s: Scalar) {
  return s.toConstant().constantValue;
}
function toScalar(s: ScalarConst) {
  return Scalar.from(s);
}

function fromPrivateKey(sk: PrivateKey) {
  return fromScalar(sk.s);
}
function toPrivateKey(sk: ScalarConst) {
  return new PrivateKey(Scalar.from(sk));
}
```

```
function fromPublicKey(pk: PublicKey): MlPublicKey {
  return MlPair(pk.x.toConstant().value[1], MlBool(pk.isOdd.toBoolean()));
}
function toPublicKey([, x, isOdd]: MlPublicKey): PublicKey {
  return PublicKey.from({
    x: Field(x),
    isOdd: Bool(MlBool.from(isOdd)),
  });
}

function fromPublicKeyVar(pk: PublicKey): MlPublicKeyVar {
  return MlPair(pk.x.value, pk.isOdd.toField().value);
}
function toPublicKeyVar([, x, isOdd]: MlPublicKeyVar): PublicKey {
  return PublicKey.from({ x: Field(x), isOdd: Bool(isOdd) });
}
```

</file>

<file>

## path: /src/lib/ml/fields.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/ml/fields.ts

```
import { ConstantField, Field, FieldConst, FieldVar } from '../field.js';
import { MlArray } from './base.js';
export { MlFieldArray, MlFieldConstArray };

type MlFieldArray = MlArray<FieldVar>;
const MlFieldArray = {
  to(arr: Field[]): MlArray<FieldVar> {
    return MlArray.to(arr.map((x) => x.value));
  },
  from([, ...arr]: MlArray<FieldVar>) {
    return arr.map((x) => new Field(x));
  },
};

type MlFieldConstArray = MlArray<FieldConst>;
const MlFieldConstArray = {
  to(arr: Field[]): MlArray<FieldConst> {
    return MlArray.to(arr.map((x) => x.toConstant().value[1]));
  },
  from([, ...arr]: MlArray<FieldConst>): ConstantField[] {
    return arr.map((x) => new Field(x) as ConstantField);
  },
};
```

</file>

<file>

## path: /src/lib/nullifier.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/nullifier.ts

```
import type { Nullifier as JsonNullifier } from '../mina-signer/src/TSTypes.js';
import { Struct } from './circuit_value.js';
import { Field, Group, Scalar } from './core.js';
import { Poseidon } from './hash.js';
import { MerkleMapWitness } from './merkle_map.js';
import { PrivateKey, PublicKey, scaleShifted } from './signature.js';
import { Provable } from './provable.js';

export { Nullifier };

/**
 *
 * Nullifiers are used as a public commitment to a specific anonymous account,
 * to forbid actions like double spending, or allow a consistent identity between anonymous actions.
 *
 * RFC: https://github.com/o1-labs/o1js/issues/756
 *
 * Paper: https://eprint.iacr.org/2022/1255.pdf
 */
class Nullifier extends Struct({
  publicKey: Group,
  public: {
    nullifier: Group,
    s: Scalar,
  },
  private: {
    c: Field,
    g_r: Group,
    h_m_pk_r: Group,
  },
}) {
  static fromJSON(json: JsonNullifier): Nullifier {
    return super.fromJSON(json as any) as Nullifier;
  }

  /**
   * Verifies that the Nullifier belongs to a specific message. Throws an error if the Nullifier is incorrect.
   *
   * @example
   *
   * &nbsp;&nbsp;&nbsp;ts
   * let nullifierMessage = [voteId, ...otherData];
   * // throws an error if the nullifier is invalid or doesn't belong to this specific message
   * nullifier.verify(nullifierMessage);
   * &nbsp;&nbsp;&nbsp;
   */
  verify(message: Field[]) {
    let {
      publicKey,
      public: { nullifier, s },
      private: { c },
    } = this;

    // generator
    let G = Group.generator;

    // serialize public key into fields once
    let pk_fields = Group.toFields(publicKey);

    // x and y of hash(msg, pk), it doesn't return a Group because y is split into x0 and x1, both two roots of a
field element
    let {
      x,
      y: { x0 },
    } = Poseidon.hashToGroup([...message, ...pk_fields]);

    // check to prevent the prover from using the second square root and forging a non-unique nullifier
    x0.isEven().assertTrue();

    let h_m_pk = Group.fromFields([x, x0]);

    // shifted scalar see https://github.com/o1-
labs/o1js/blob/5333817a62890c43ac1b9cb345748984df271b62/src/lib/signature.ts#L220
    // pk^c
    let pk_c = scaleShifted(this.publicKey, Scalar.fromBits(c.toBits()));

    // g^r = g^s / pk^c
    let g_r = G.scale(s).sub(pk_c);

    // h(m, pk)^s
    let h_m_pk_s = Group.scale(h_m_pk, s);

    // h_m_pk_r =  h(m,pk)^s / nullifier^c
    let h_m_pk_s_div_nullifier_s = h_m_pk_s.sub(
      scaleShifted(nullifier, Scalar.fromBits(c.toBits()))
    );

    // this is supposed to match the entries generated on "the other side" of the nullifier (mina-signer, in an
wallet enclave)
    Poseidon.hash([
      ...Group.toFields(G),
      ...pk_fields,
      x,
      x0,
      ...Group.toFields(nullifier),
      ...Group.toFields(g_r),
      ...Group.toFields(h_m_pk_s_div_nullifier_s),
    ]).assertEquals(c, 'Nullifier does not match private input!');
  }

  /**
   * The key of the nullifier, which belongs to a unique message and a public key.
   * Used as an index in Merkle trees.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * // returns the key of the nullifier which can be used as index in a Merkle tree/map
   * let key = nullifier.key();
   * &nbsp;&nbsp;&nbsp;
   */
  key() {
    return Poseidon.hash(Group.toFields(this.public.nullifier));
  }

  /**
   * Returns the state of the Nullifier.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * // returns a Bool based on whether or not the nullifier has been used before
   * let isUnused = nullifier.isUnused();
   * &nbsp;&nbsp;&nbsp;
   */
  isUnused(witness: MerkleMapWitness, root: Field) {
    let [newRoot, key] = witness.computeRootAndKey(Field(0));
    key.assertEquals(this.key());
    let isUnused = newRoot.equals(root);

    let isUsed = witness.computeRootAndKey(Field(1))[0].equals(root);
    // prove that our Merkle witness is correct
    isUsed.or(isUnused).assertTrue();
    return isUnused; // if this is false, &nbsp;isUsed&nbsp; is true because of the check before
  }

  /**
   * Checks if the Nullifier has been used before.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * // asserts that the nullifier has not been used before, throws an error otherwise
   * nullifier.assertUnused();
   * &nbsp;&nbsp;&nbsp;
   */
  assertUnused(witness: MerkleMapWitness, root: Field) {
    let [impliedRoot, key] = witness.computeRootAndKey(Field(0));
    this.key().assertEquals(key);
    impliedRoot.assertEquals(root);
  }
```

```
/**
 * Sets the Nullifier, returns the new Merkle root.
 *
 * @example
 * &nbsp;&nbsp;&nbsp;ts
 * // calculates the new root of the Merkle tree in which the nullifier is set to used
 * let newRoot = nullifier.setUsed(witness);
 * &nbsp;&nbsp;&nbsp;
 */
setUsed(witness: MerkleMapWitness) {
  let [newRoot, key] = witness.computeRootAndKey(Field(1));
  key.assertEquals(this.key());
  return newRoot;
}

/**
 * Returns the {@link PublicKey} that is associated with this Nullifier.
 *
 * @example
 * &nbsp;&nbsp;&nbsp;ts
 * let pk = nullifier.getPublicKey();
 * &nbsp;&nbsp;&nbsp;
 */
getPublicKey() {
  return PublicKey.fromGroup(this.publicKey);
}

/**
 *
 * _Note_: This is *not* the recommended way to create a Nullifier in production. Please use mina-signer to
create Nullifiers.
 * Also, this function cannot be run within provable code to avoid unintended creations of Nullifiers - a
Nullifier should never be created inside proveable code (e.g. a smart contract) directly, but rather created
inside the users wallet (or other secure enclaves, so the private key never leaves that enclave).
 *
 * PLUME: An ECDSA Nullifier Scheme for Unique
 * Pseudonymity within Zero Knowledge Proofs
 * https://eprint.iacr.org/2022/1255.pdf chapter 3 page 14
 */
static createTestNullifier(message: Field[], sk: PrivateKey): JsonNullifier {
  if (Provable.inCheckedComputation()) {
    throw Error(
      'This function cannot not be run within provable code. If you want to create a Nullifier, run this method
outside provable code or use mina-signer to do so.'
    );
  }
  const Hash2 = Poseidon.hash;
  const Hash = Poseidon.hashToGroup;

  const pk = sk.toPublicKey().toGroup();

  const G = Group.generator;
```

```
    const r = Scalar.random();

    const gm = Hash([...message, ...Group.toFields(pk)]);

    const h_m_pk = Group({ x: gm.x, y: gm.y.x0 });

    const nullifier = h_m_pk.scale(sk.toBigInt());
    const h_m_pk_r = h_m_pk.scale(r.toBigInt());

    const g_r = G.scale(r.toBigInt());

    const c = Hash2([
      ...Group.toFields(G),
      ...Group.toFields(pk),
      ...Group.toFields(h_m_pk),
      ...Group.toFields(nullifier),
      ...Group.toFields(g_r),
      ...Group.toFields(h_m_pk_r),
    ]);

    // operations on scalars (r) should be in Fq, rather than Fp
    // while c is in Fp (due to Poseidon.hash), c needs to be handled as an element from Fq
    const s = r.add(sk.s.mul(Scalar.from(c.toBigInt())));

    return {
      publicKey: pk.toJSON(),
      private: {
        c: c.toString(),
        g_r: g_r.toJSON(),
        h_m_pk_r: h_m_pk_r.toJSON(),
      },
      public: {
        nullifier: nullifier.toJSON(),
        s: s.toJSON(),
      },
    };
  }
}
```

</file>

<file>

## path: /src/lib/nullifier.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/nullifier.unit-test.ts

```
import { createNullifier } from '../mina-signer/src/nullifier.js';
import { Field } from './core.js';
import { Nullifier } from './nullifier.js';
import { PrivateKey } from './signature.js';

let priv = PrivateKey.random();

let sk = BigInt(priv.s.toJSON());

let message = Array<Field>(5).fill(Field.random());

let jsonNullifier1 = createNullifier(
  message.map((f) => f.toBigInt()),
  sk
);

let nullifier1 = Nullifier.fromJSON(jsonNullifier1);
nullifier1.verify(message);

console.log('nullifier correctly deserializes, serializes and verifies');

// random sk that does not belong to a pk
let sk_faulty = BigInt(PrivateKey.random().s.toJSON());

let jsonNullifier2 = createNullifier(
  message.map((f) => f.toBigInt()),
  sk_faulty
);

// trying to manipulate the nullifier to take a real pk that it doesnt know the sk to
let pk = priv.toPublicKey().toGroup();
jsonNullifier2.publicKey = {
  x: pk.x.toBigInt(),
  y: pk.y.toBigInt(),
};

let nullifier2 = Nullifier.fromJSON(jsonNullifier2);
try {
  nullifier2.verify(message);
  console.log('incorrect nullifier was verified');
  console.log(JSON.stringify(nullifier2));
  process.exit(1);
} catch {
  console.log('invalid nullifier correctly throws an error (sk not known)');
}

let jsonNullifier3 = createNullifier(
  message.map((f) => f.toBigInt()),
  sk
);

// trying to manipulate the nullifier to take a different message
let nullifier3 = Nullifier.fromJSON(jsonNullifier3);
try {
  nullifier3.verify(Array<Field>(5).fill(Field.random()));
  console.log('incorrect nullifier was verified');
  console.log(JSON.stringify(nullifier3));
  process.exit(1);
} catch {
  console.log(
    'invalid nullifier correctly throws an error (manipulated message)'
  );
}
```

</file>

<file>

## path: /src/lib/precondition.test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/precondition.test.ts

```
import {
  shutdown,
  isReady,
  UInt64,
  UInt32,
  SmartContract,
  Mina,
  PrivateKey,
  AccountUpdate,
  method,
  PublicKey,
  Bool,
  Field,
} from 'o1js';

class MyContract extends SmartContract {
  @method shouldMakeCompileThrow() {
    this.network.blockchainLength.get();
  }
}

let zkappKey: PrivateKey;
let zkappAddress: PublicKey;
let zkapp: MyContract;
let feePayer: PublicKey;
let feePayerKey: PrivateKey;

beforeAll(async () => {
  // set up local blockchain, create zkapp keys, deploy the contract
  await isReady;
  let Local = Mina.LocalBlockchain({ proofsEnabled: false });
  Mina.setActiveInstance(Local);
  feePayerKey = Local.testAccounts[0].privateKey;
  feePayer = Local.testAccounts[0].publicKey;
  zkappKey = PrivateKey.random();
  zkappAddress = zkappKey.toPublicKey();
  zkapp = new MyContract(zkappAddress);

  let tx = await Mina.transaction(feePayer, () => {
    AccountUpdate.fundNewAccount(feePayer);
    zkapp.deploy();
  });
  tx.sign([feePayerKey, zkappKey]).send();
});
afterAll(() => setTimeout(shutdown, 0));

describe('preconditions', () => {
  it('get without constraint should throw', async () => {
    for (let precondition of implemented) {
      await expect(
        Mina.transaction(feePayer, () => {
          precondition().get();
          AccountUpdate.attachToTransaction(zkapp.self);
        })
      ).rejects.toThrow(/precondition/);
    }
  });

  it('get without constraint should throw during compile', async () => {
    await expect(() => MyContract.compile()).rejects.toThrow('precondition');
  });

  it('get + assertEquals should not throw', async () => {
    let nonce = zkapp.account.nonce.get();
    let tx = await Mina.transaction(feePayer, () => {
      zkapp.requireSignature();
      for (let precondition of implemented) {
        let p = precondition().get();
        precondition().assertEquals(p as any);
      }
      AccountUpdate.attachToTransaction(zkapp.self);
    });
    await tx.sign([feePayerKey, zkappKey]).send();
    // check that tx was applied, by checking nonce was incremented
    expect(zkapp.account.nonce.get()).toEqual(nonce.add(1));
  });

  it('get + assertEquals should throw for unimplemented fields', async () => {
    for (let precondition of unimplemented) {
      await expect(
        Mina.transaction(feePayer, () => {
          let p = precondition();
          p.assertEquals(p.get() as any);
          AccountUpdate.attachToTransaction(zkapp.self);
        })
      ).rejects.toThrow(/not implemented/);
    }
  });

  it('get + assertBetween should not throw', async () => {
    let nonce = zkapp.account.nonce.get();
    let tx = await Mina.transaction(feePayer, () => {
      for (let precondition of implementedWithRange) {
        let p: any = precondition().get();
        precondition().assertBetween(p.constructor.zero, p);
      }
      zkapp.requireSignature();
      AccountUpdate.attachToTransaction(zkapp.self);
    });
    await tx.sign([feePayerKey, zkappKey]).send();
    // check that tx was applied, by checking nonce was incremented
    expect(zkapp.account.nonce.get()).toEqual(nonce.add(1));
  });

  it('satisfied currentSlot.assertBetween should not throw', async () => {
    let nonce = zkapp.account.nonce.get();
    let tx = await Mina.transaction(feePayer, () => {
      zkapp.currentSlot.assertBetween(
        UInt32.from(0),
        UInt32.from(UInt32.MAXINT())
      );
      zkapp.requireSignature();
      AccountUpdate.attachToTransaction(zkapp.self);
    });
    await tx.sign([feePayerKey, zkappKey]).send();
    expect(zkapp.account.nonce.get()).toEqual(nonce.add(1));
  });

  it('get + assertNothing should not throw', async () => {
    let nonce = zkapp.account.nonce.get();
    let tx = await Mina.transaction(feePayer, () => {
      for (let precondition of implemented) {
        precondition().get();
        precondition().assertNothing();
      }
      zkapp.requireSignature();
      AccountUpdate.attachToTransaction(zkapp.self);
    });
    await tx.sign([feePayerKey, zkappKey]).send();
    // check that tx was applied, by checking nonce was incremented
    expect(zkapp.account.nonce.get()).toEqual(nonce.add(1));
  });

  it('get + manual precondition should not throw', async () => {
    // we only test this for a couple of preconditions
    let nonce = zkapp.account.nonce.get();
    let tx = await Mina.transaction(feePayer, () => {
      zkapp.account.balance.get();
      zkapp.self.body.preconditions.account.balance.isSome = Bool(true);
      zkapp.self.body.preconditions.account.balance.value.upper =
        UInt64.from(10e9);

      zkapp.network.blockchainLength.get();
      zkapp.self.body.preconditions.network.blockchainLength.isSome =
        Bool(true);
      zkapp.self.body.preconditions.network.blockchainLength.value.upper =
        UInt32.from(1000);

      zkapp.network.totalCurrency.get();
      zkapp.self.body.preconditions.network.totalCurrency.isSome = Bool(true);
      zkapp.self.body.preconditions.network.totalCurrency.value.upper =
        UInt64.from(1e9 * 1e9);
      zkapp.requireSignature();
      AccountUpdate.attachToTransaction(zkapp.self);
    });
    await tx.sign([feePayerKey, zkappKey]).send();
    // check that tx was applied, by checking nonce was incremented
    expect(zkapp.account.nonce.get()).toEqual(nonce.add(1));
  });

  it('unsatisfied assertEquals should be rejected (numbers)', async () => {
    for (let precondition of implementedNumber) {
      await expect(async () => {
        let tx = await Mina.transaction(feePayer, () => {
          let p = precondition().get();
          precondition().assertEquals(p.add(1) as any);
          AccountUpdate.attachToTransaction(zkapp.self);
        });
        await tx.sign([feePayerKey]).send();
      }).rejects.toThrow(/unsatisfied/);
    }
  });

  it('unsatisfied assertEquals should be rejected (booleans)', async () => {
    for (let precondition of implementedBool) {
      let tx = await Mina.transaction(feePayer, () => {
        let p = precondition().get();
        precondition().assertEquals(p.not());
        AccountUpdate.attachToTransaction(zkapp.self);
      });
      await expect(tx.sign([feePayerKey]).send()).rejects.toThrow(
        /unsatisfied/
      );
    }
  });

  it('unsatisfied assertEquals should be rejected (public key)', async () => {
    let publicKey = PublicKey.from({ x: Field(-1), isOdd: Bool(false) });
    let tx = await Mina.transaction(feePayer, () => {
      zkapp.account.delegate.assertEquals(publicKey);
      AccountUpdate.attachToTransaction(zkapp.self);
    });
    await expect(tx.sign([feePayerKey]).send()).rejects.toThrow(/unsatisfied/);
  });

  it('unsatisfied assertBetween should be rejected', async () => {
    for (let precondition of implementedWithRange) {
      let tx = await Mina.transaction(feePayer, () => {
        let p: any = precondition().get();
        precondition().assertBetween(p.add(20), p.add(30));
        AccountUpdate.attachToTransaction(zkapp.self);
      });
      await expect(tx.sign([feePayerKey]).send()).rejects.toThrow(
        /unsatisfied/
      );
    }
  });

  it('unsatisfied currentSlot.assertBetween should be rejected', async () => {
    let tx = await Mina.transaction(feePayer, () => {
      zkapp.currentSlot.assertBetween(UInt32.from(20), UInt32.from(30));
      AccountUpdate.attachToTransaction(zkapp.self);
    });
    await expect(tx.sign([feePayerKey]).send()).rejects.toThrow(/unsatisfied/);
  });

  // TODO: is this a gotcha that should be addressed?
  // the test below fails, so it seems that nonce is applied successfully with a WRONG precondition..
  // however, this is just because &nbsp;zkapp.sign()&nbsp; overwrites the nonce precondition with one that
 is satisfied
  it.skip('unsatisfied nonce precondition should be rejected', async () => {
    let tx = await Mina.transaction(feePayer, () => {
      zkapp.account.nonce.assertEquals(UInt32.from(1e8));
      zkapp.requireSignature();
      AccountUpdate.attachToTransaction(zkapp.self);
    });
    expect(() => tx.sign([zkappKey, feePayerKey]).send()).toThrow();
  });
 });

 let implementedNumber = [
  () => zkapp.account.balance,
  () => zkapp.account.nonce,
  () => zkapp.account.receiptChainHash,
  () => zkapp.network.blockchainLength,
  () => zkapp.network.globalSlotSinceGenesis,
  () => zkapp.network.timestamp,
  () => zkapp.network.minWindowDensity,
  () => zkapp.network.totalCurrency,
  () => zkapp.network.stakingEpochData.epochLength,
  () => zkapp.network.stakingEpochData.ledger.totalCurrency,
  () => zkapp.network.nextEpochData.epochLength,
  () => zkapp.network.nextEpochData.ledger.totalCurrency,
  () => zkapp.network.snarkedLedgerHash,
  () => zkapp.network.stakingEpochData.lockCheckpoint,
  () => zkapp.network.stakingEpochData.startCheckpoint,
  // () => zkapp.network.stakingEpochData.seed,
  () => zkapp.network.stakingEpochData.ledger.hash,
  () => zkapp.network.nextEpochData.lockCheckpoint,
  () => zkapp.network.nextEpochData.startCheckpoint,
  // () => zkapp.network.nextEpochData.seed,
  () => zkapp.network.nextEpochData.ledger.hash,
];
let implementedBool = [
  () => zkapp.account.isNew,
  () => zkapp.account.provedState,
];
let implemented = [
  ...implementedNumber,
  ...implementedBool,
  () => zkapp.account.delegate,
];
let implementedWithRange = [
  () => zkapp.account.balance,
  () => zkapp.account.nonce,
  () => zkapp.network.blockchainLength,
  () => zkapp.network.globalSlotSinceGenesis,
  () => zkapp.network.timestamp,
  () => zkapp.network.minWindowDensity,
  () => zkapp.network.totalCurrency,
  () => zkapp.network.stakingEpochData.epochLength,
  () => zkapp.network.stakingEpochData.ledger.totalCurrency,
  () => zkapp.network.nextEpochData.epochLength,
  () => zkapp.network.nextEpochData.ledger.totalCurrency,
];
let implementedWithRangeOnly = [() => zkapp.currentSlot];
let unimplemented = [
  () => zkapp.network.stakingEpochData.seed,
  () => zkapp.network.nextEpochData.seed,
];
```

</file>

<file>

## path: /src/lib/precondition.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/precondition.ts

```
import { Bool, Field } from './core.js';
import { circuitValueEquals } from './circuit_value.js';
import { Provable } from './provable.js';
import * as Mina from './mina.js';
import { Actions, AccountUpdate, Preconditions } from './account_update.js';
import { Int64, UInt32, UInt64 } from './int.js';
import { Layout } from '../bindings/mina-transaction/gen/transaction.js';
import { jsLayout } from '../bindings/mina-transaction/gen/js-layout.js';
import { emptyReceiptChainHash, TokenSymbol } from './hash.js';
import { PublicKey } from './signature.js';
import { ZkappUri } from '../bindings/mina-transaction/transaction-leaves.js';

export {
  preconditions,
  Account,
  Network,
  CurrentSlot,
  assertPreconditionInvariants,
  cleanPreconditionsCache,
  AccountValue,
  NetworkValue,
  getAccountPreconditions,
};

function preconditions(accountUpdate: AccountUpdate, isSelf: boolean) {
  initializePreconditions(accountUpdate, isSelf);
  return {
    account: Account(accountUpdate),
    network: Network(accountUpdate),
    currentSlot: CurrentSlot(accountUpdate),
  };
}

// note: please keep the two precondition implementations separate
// so we can add customized fields easily

function Network(accountUpdate: AccountUpdate): Network {
  let layout =
    jsLayout.AccountUpdate.entries.body.entries.preconditions.entries.network;
  let context = getPreconditionContextExn(accountUpdate);
  let network: RawNetwork = preconditionClass(
    layout as Layout,
    'network',
    accountUpdate,
    context
  );
  let timestamp = {
    get() {
      let slot = network.globalSlotSinceGenesis.get();
      return globalSlotToTimestamp(slot);
    },
    getAndAssertEquals() {
      let slot = network.globalSlotSinceGenesis.getAndAssertEquals();
      return globalSlotToTimestamp(slot);
    },
    assertEquals(value: UInt64) {
      let { genesisTimestamp, slotTime } =
        Mina.activeInstance.getNetworkConstants();
      let slot = timestampToGlobalSlot(
        value,
        &nbsp;Timestamp precondition unsatisfied: the timestamp can only equal numbers of the form
${genesisTimestamp} + k*${slotTime},\n&nbsp; +
          &nbsp;i.e., the genesis timestamp plus an integer number of slots.&nbsp;
      );
      return network.globalSlotSinceGenesis.assertEquals(slot);
    },
    assertBetween(lower: UInt64, upper: UInt64) {
      let [slotLower, slotUpper] = timestampToGlobalSlotRange(lower, upper);
      return network.globalSlotSinceGenesis.assertBetween(slotLower, slotUpper);
    },
    assertNothing() {
      return network.globalSlotSinceGenesis.assertNothing();
    },
  };
  return { ...network, timestamp };
}

function Account(accountUpdate: AccountUpdate): Account {
  let layout =
    jsLayout.AccountUpdate.entries.body.entries.preconditions.entries.account;
  let context = getPreconditionContextExn(accountUpdate);
  let identity = (x: any) => x;
  let update: Update = {
    delegate: {
      ...preconditionSubclass(
        accountUpdate,
        'account.delegate',
        PublicKey,
        context
      ),
      ...updateSubclass(accountUpdate, 'delegate', identity),
    },
    verificationKey: updateSubclass(accountUpdate, 'verificationKey', identity),
    permissions: updateSubclass(accountUpdate, 'permissions', identity),
    zkappUri: updateSubclass(accountUpdate, 'zkappUri', ZkappUri.fromJSON),
    tokenSymbol: updateSubclass(accountUpdate, 'tokenSymbol', TokenSymbol.from),
    timing: updateSubclass(accountUpdate, 'timing', identity),
    votingFor: updateSubclass(accountUpdate, 'votingFor', identity),
  };
  return {
    ...preconditionClass(layout as Layout, 'account', accountUpdate, context),
    ...update,
  };
}

function updateSubclass<K extends keyof Update>(
  accountUpdate: AccountUpdate,
  key: K,
  transform: (value: UpdateValue[K]) => UpdateValueOriginal[K]
) {
  return {
    set(value: UpdateValue[K]) {
      accountUpdate.body.update[key].isSome = Bool(true);
      accountUpdate.body.update[key].value = transform(value);
    },
  };
}

function CurrentSlot(accountUpdate: AccountUpdate): CurrentSlot {
  let context = getPreconditionContextExn(accountUpdate);
  return {
    assertBetween(lower: UInt32, upper: UInt32) {
      context.constrained.add('validWhile');
      let property: RangeCondition<UInt32> =
        accountUpdate.body.preconditions.validWhile;
      property.isSome = Bool(true);
      property.value.lower = lower;
      property.value.upper = upper;
    },
  };
}

let unimplementedPreconditions: LongKey[] = [
  // unimplemented because its not checked in the protocol
  'network.stakingEpochData.seed',
  'network.nextEpochData.seed',
];

type BaseType = 'UInt64' | 'UInt32' | 'Field' | 'Bool' | 'PublicKey';
let baseMap = { UInt64, UInt32, Field, Bool, PublicKey };

function preconditionClass(
  layout: Layout,
  baseKey: any,
  accountUpdate: AccountUpdate,
  context: PreconditionContext
): any {
  if (layout.type === 'option') {
    // range condition
    if (layout.optionType === 'closedInterval') {
      let lower = layout.inner.entries.lower.type as BaseType;
      let baseType = baseMap[lower];
      return preconditionSubClassWithRange(
        accountUpdate,
        baseKey,
        baseType as any,
        context
      );
    }
    // value condition
    else if (layout.optionType === 'flaggedOption') {
      let baseType = baseMap[layout.inner.type as BaseType];
      return preconditionSubclass(
        accountUpdate,
        baseKey,
        baseType as any,
        context
      );
    }
  } else if (layout.type === 'array') {
    return {}; // not applicable yet, TODO if we implement state
  } else if (layout.type === 'object') {
    // for each field, create a recursive object
    return Object.fromEntries(
      layout.keys.map((key) => {
        let value = layout.entries[key];
        return [
          key,
          preconditionClass(value, &nbsp;${baseKey}.${key}&nbsp;, accountUpdate, context),
        ];
      })
    );
  } else throw Error('bug');
}

function preconditionSubClassWithRange<
  K extends LongKey,
  U extends FlatPreconditionValue[K]
>(
  accountUpdate: AccountUpdate,
  longKey: K,
  fieldType: Provable<U>,
  context: PreconditionContext
) {
  return {
    ...preconditionSubclass(accountUpdate, longKey, fieldType as any, context),
    assertBetween(lower: any, upper: any) {
      context.constrained.add(longKey);
      let property: RangeCondition<any> = getPath(
        accountUpdate.body.preconditions,
        longKey
      );
      property.isSome = Bool(true);
      property.value.lower = lower;
      property.value.upper = upper;
    },
  };
}

function preconditionSubclass<
  K extends LongKey,
  U extends FlatPreconditionValue[K]
>(
  accountUpdate: AccountUpdate,
  longKey: K,
  fieldType: Provable<U>,
  context: PreconditionContext
) {
  if (fieldType === undefined) {
    throw Error(&nbsp;this.${longKey}: fieldType undefined&nbsp;);
  }
  let obj = {
    get() {
      if (unimplementedPreconditions.includes(longKey)) {
        let self = context.isSelf ? 'this' : 'accountUpdate';
        throw Error(&nbsp;${self}.${longKey}.get() is not implemented yet.&nbsp;);
      }
      let { read, vars } = context;
      read.add(longKey);
      return (vars[longKey] ??= getVariable(
        accountUpdate,
        longKey,
        fieldType
      )) as U;
    },
    getAndAssertEquals() {
      let value = obj.get();
      obj.assertEquals(value);
      return value;
    },
    assertEquals(value: U) {
      context.constrained.add(longKey);
      let property = getPath(
        accountUpdate.body.preconditions,
        longKey
      ) as AnyCondition<U>;
      if ('isSome' in property) {
        property.isSome = Bool(true);
        if ('lower' in property.value && 'upper' in property.value) {
          property.value.lower = value;
          property.value.upper = value;
        } else {
          property.value = value;
        }
      } else {
        setPath(accountUpdate.body.preconditions, longKey, value);
      }
    },
    assertNothing() {
      context.constrained.add(longKey);
    },
  };
  return obj;
```

```
}

function getVariable<K extends LongKey, U extends FlatPreconditionValue[K]>(
  accountUpdate: AccountUpdate,
  longKey: K,
  fieldType: Provable<U>
): U {
  return Provable.witness(fieldType, () => {
    let [accountOrNetwork, ...rest] = longKey.split('.');
    let key = rest.join('.');
    let value: U;
    if (accountOrNetwork === 'account') {
      let account = getAccountPreconditions(accountUpdate.body);
      value = account[key as keyof AccountValue] as U;
    } else if (accountOrNetwork === 'network') {
      let networkState = Mina.getNetworkState();
      value = getPath(networkState, key);
    } else if (accountOrNetwork === 'validWhile') {
      let networkState = Mina.getNetworkState();
      value = networkState.globalSlotSinceGenesis as U;
    } else {
      throw Error('impossible');
    }
    return value;
  });
}

function globalSlotToTimestamp(slot: UInt32) {
  let { genesisTimestamp, slotTime } =
    Mina.activeInstance.getNetworkConstants();
  return UInt64.from(slot).mul(slotTime).add(genesisTimestamp);
}
function timestampToGlobalSlot(timestamp: UInt64, message: string) {
  let { genesisTimestamp, slotTime } =
    Mina.activeInstance.getNetworkConstants();
  let { quotient: slot, rest } = timestamp
    .sub(genesisTimestamp)
    .divMod(slotTime);
  rest.value.assertEquals(Field(0), message);
  return slot.toUInt32();
}

function timestampToGlobalSlotRange(
  tsLower: UInt64,
  tsUpper: UInt64
): [lower: UInt32, upper: UInt32] {
  // we need &nbsp;slotLower <= current slot <= slotUpper&nbsp; to imply &nbsp;tsLower <= current
timestamp <= tsUpper&nbsp;
  // so we have to make the range smaller -- round up &nbsp;tsLower&nbsp; and round down
&nbsp;tsUpper&nbsp;
  // also, we should clamp to the UInt32 max range [0, 2**32-1]
  let { genesisTimestamp, slotTime } =
    Mina.activeInstance.getNetworkConstants();
  let tsLowerInt = Int64.from(tsLower)
    .sub(genesisTimestamp)
    .add(slotTime)
    .sub(1);
  let lowerCapped = Provable.if<UInt64>(
    tsLowerInt.isPositive(),
    UInt64,
    tsLowerInt.magnitude,
    UInt64.from(0)
  );
  let slotLower = lowerCapped.div(slotTime).toUInt32Clamped();
  // unsafe &nbsp;sub&nbsp; means the error in case tsUpper underflows slot 0 is ugly, but should not be
relevant in practice
  let slotUpper = tsUpper.sub(genesisTimestamp).div(slotTime).toUInt32Clamped();
  return [slotLower, slotUpper];
}

function getAccountPreconditions(body: {
  publicKey: PublicKey;
  tokenId?: Field;
}): AccountValue {
  let { publicKey, tokenId } = body;
  let hasAccount = Mina.hasAccount(publicKey, tokenId);
  if (!hasAccount) {
    return {
      balance: UInt64.zero,
      nonce: UInt32.zero,
      receiptChainHash: emptyReceiptChainHash(),
      actionState: Actions.emptyActionState(),
      delegate: publicKey,
      provedState: Bool(false),
      isNew: Bool(true),
    };
  }
  let account = Mina.getAccount(publicKey, tokenId);
  return {
    balance: account.balance,
    nonce: account.nonce,
    receiptChainHash: account.receiptChainHash,
    actionState: account.zkapp?.actionState?.[0] ?? Actions.emptyActionState(),
    delegate: account.delegate ?? account.publicKey,
    provedState: account.zkapp?.provedState ?? Bool(false),
    isNew: Bool(false),
  };
}

// per account update context for checking invariants on precondition construction
type PreconditionContext = {
  isSelf: boolean;
  vars: Partial<FlatPreconditionValue>;
  read: Set<LongKey>;
  constrained: Set<LongKey>;
};

function initializePreconditions(
  accountUpdate: AccountUpdate,
  isSelf: boolean
) {
  preconditionContexts.set(accountUpdate, {
    read: new Set(),
    constrained: new Set(),
    vars: {},
    isSelf,
  });
}

function cleanPreconditionsCache(accountUpdate: AccountUpdate) {
  let context = preconditionContexts.get(accountUpdate);
  if (context !== undefined) context.vars = {};
}

function assertPreconditionInvariants(accountUpdate: AccountUpdate) {
  let context = getPreconditionContextExn(accountUpdate);
  let self = context.isSelf ? 'this' : 'accountUpdate';
  let dummyPreconditions = Preconditions.ignoreAll();
  for (let preconditionPath of context.read) {
    // check if every precondition that was read was also contrained
    if (context.constrained.has(preconditionPath)) continue;

    // check if the precondition was modified manually, which is also a valid way of avoiding an error
    let precondition = getPath(
      accountUpdate.body.preconditions,
      preconditionPath
    );
    let dummy = getPath(dummyPreconditions, preconditionPath);
    if (!circuitValueEquals(precondition, dummy)) continue;

    // we accessed a precondition field but not constrained it explicitly - throw an error
    let hasAssertBetween = isRangeCondition(precondition);
    let shortPath = preconditionPath.split('.').pop();
    let errorMessage = &nbsp;You used \&nbsp;${self}.${preconditionPath}.get()\&nbsp; without adding a
precondition that links it to the actual ${shortPath}.
Consider adding this line to your code:
${self}.${preconditionPath}.assertEquals(${self}.${preconditionPath}.get());${
      hasAssertBetween
        ? &nbsp;
You can also add more flexible preconditions with
\&nbsp;${self}.${preconditionPath}.assertBetween(...)\&nbsp;.&nbsp;
        : ''
    }&nbsp;;
    throw Error(errorMessage);
  }
}
```

```
function getPreconditionContextExn(accountUpdate: AccountUpdate) {
  let c = preconditionContexts.get(accountUpdate);
  if (c === undefined) throw Error('bug: precondition context not found');
  return c;
}

const preconditionContexts = new WeakMap<AccountUpdate, PreconditionContext>();

// exported types

type NetworkPrecondition = Preconditions['network'];
type NetworkValue = PreconditionBaseTypes<NetworkPrecondition>;
type RawNetwork = PreconditionClassType<NetworkPrecondition>;
type Network = RawNetwork & {
  timestamp: PreconditionSubclassRangeType<UInt64>;
};

// TODO: should we add account.state?
// then can just use circuitArray(Field, 8) as the type
type AccountPrecondition = Omit<Preconditions['account'], 'state'>;
type AccountValue = PreconditionBaseTypes<AccountPrecondition>;
type Account = PreconditionClassType<AccountPrecondition> & Update;

type CurrentSlotPrecondition = Preconditions['validWhile'];
type CurrentSlot = {
  assertBetween(lower: UInt32, upper: UInt32): void;
};

type PreconditionBaseTypes<T> = {
  [K in keyof T]: T[K] extends RangeCondition<infer U>
    ? U
    : T[K] extends FlaggedOptionCondition<infer U>
    ? U
    : T[K] extends Field
    ? Field
    : PreconditionBaseTypes<T[K]>;
};

type PreconditionSubclassType<U> = {
  get(): U;
  getAndAssertEquals(): U;
  assertEquals(value: U): void;
  assertNothing(): void;
};
type PreconditionSubclassRangeType<U> = PreconditionSubclassType<U> & {
  assertBetween(lower: U, upper: U): void;
};

type PreconditionClassType<T> = {
  [K in keyof T]: T[K] extends RangeCondition<infer U>
    ? PreconditionSubclassRangeType<U>
    : T[K] extends FlaggedOptionCondition<infer U>
    ? PreconditionSubclassType<U>
    : T[K] extends Field
    ? PreconditionSubclassType<Field>
    : PreconditionClassType<T[K]>;
};

// update

type Update_ = Omit<AccountUpdate['body']['update'], 'appState'>;
type Update = {
  [K in keyof Update_]: { set(value: UpdateValue[K]): void };
};
type UpdateValueOriginal = {
  [K in keyof Update_]: Update_[K]['value'];
};
type UpdateValue = {
  [K in keyof Update_]: K extends 'zkappUri' | 'tokenSymbol'
    ? string
    : Update_[K]['value'];
};

// TS magic for computing flattened precondition types

type JoinEntries<K, P> = K extends string
  ? P extends [string, unknown, unknown]
    ? [&nbsp;${K}${P[0] extends '' ? '' : '.'}${P[0]}&nbsp;, P[1], P[2]]
    : never
  : never;

type PreconditionFlatEntry<T> = T extends RangeCondition<infer V>
  ? ['', T, V]
  : T extends FlaggedOptionCondition<infer U>
  ? ['', T, U]
  : { [K in keyof T]: JoinEntries<K, PreconditionFlatEntry<T[K]>> }[keyof T];

type FlatPreconditionValue = {
  [S in PreconditionFlatEntry<NetworkPrecondition> as &nbsp;network.${S[0]}&nbsp;]: S[2];
} & {
  [S in PreconditionFlatEntry<AccountPrecondition> as &nbsp;account.${S[0]}&nbsp;]: S[2];
} & { validWhile: PreconditionFlatEntry<CurrentSlotPrecondition>[2] };

type LongKey = keyof FlatPreconditionValue;

// types for the two kinds of conditions
type RangeCondition<T> = { isSome: Bool; value: { lower: T; upper: T } };
type FlaggedOptionCondition<T> = { isSome: Bool; value: T };
type AnyCondition<T> = RangeCondition<T> | FlaggedOptionCondition<T>;

function isRangeCondition<T extends object>(
  condition: AnyCondition<T>
): condition is RangeCondition<T> {
  return 'isSome' in condition && 'lower' in condition.value;
}

// helper. getPath({a: {b: 'x'}}, 'a.b') === 'x'
// TODO: would be awesome to type this
function getPath(obj: any, path: string) {
  let pathArray = path.split('.').reverse();
  while (pathArray.length > 0) {
    let key = pathArray.pop();
    obj = obj[key as any];
  }
  return obj;
}
function setPath(obj: any, path: string, value: any) {
  let pathArray = path.split('.');
  let key = pathArray.pop()!;
  getPath(obj, pathArray.join('.'))[key] = value;
}
```

</file>

<file>

## path: /src/lib/primitives.test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/primitives.test.ts

```
import { isReady, shutdown, Field, Bool, Provable } from 'o1js';
describe('bool', () => {
  beforeAll(async () => {
    await isReady;
    return;
  });

  afterAll(async () => {
    setTimeout(async () => {
      await shutdown();
    }, 0);
  });

  describe('inside circuit', () => {
    describe('toField', () => {
      it('should return a Field', async () => {
        expect(true).toEqual(true);
      });
      it('should convert false to Field element 0', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const xFalse = Provable.witness(Bool, () => new Bool(false));

            xFalse.toField().assertEquals(new Field(0));
          });
        }).not.toThrow();
      });
      it('should throw when false toString is compared to Field element other than 0 ', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const xFalse = Provable.witness(Bool, () => new Bool(false));
            xFalse.toField().assertEquals(new Field(1));
          });
        }).toThrow();
      });

      it('should convert true to Field element 1', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const xTrue = Provable.witness(Bool, () => new Bool(true));
            xTrue.toField().assertEquals(new Field(1));
          });
        }).not.toThrow();
      });

      it('should throw when true toField is compared to Field element other than 1 ', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const xTrue = Provable.witness(Bool, () => new Bool(true));
            xTrue.toField().assertEquals(new Field(0));
          });
        }).toThrow();
      });
    });

    describe('toFields', () => {
      it('should return an array of Fields', () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const x = Provable.witness(Bool, () => new Bool(false));
            const fieldArr = x.toFields();
            const isArr = Array.isArray(fieldArr);
            expect(isArr).toBe(true);
            fieldArr[0].assertEquals(new Field(0));
          });
        }).not.toThrow();
      });
    });
    describe('and', () => {
      it('true "and" true should return true', async () => {
        expect(() => {
          Provable.runAndCheck(() => {
            const xTrue = Provable.witness(Bool, () => new Bool(true));
            const yTrue = Provable.witness(Bool, () => new Bool(true));

            xTrue.and(yTrue).assertEquals(new Bool(true));
        });
      }).not.toThrow();
    });

    it('should throw if true "and" true is compared to false', async () => {
      expect(() => {
        Provable.runAndCheck(() => {
          const xTrue = Provable.witness(Bool, () => new Bool(true));
          const yTrue = Provable.witness(Bool, () => new Bool(true));

          xTrue.and(yTrue).assertEquals(new Bool(false));
        });
      }).toThrow();
    });

    it('false "and" false should return false', async () => {
      expect(() => {
        Provable.runAndCheck(() => {
          const xFalse = Provable.witness(Bool, () => new Bool(false));
          const yFalse = Provable.witness(Bool, () => new Bool(false));

          xFalse.and(yFalse).assertEquals(new Bool(false));
        });
      }).not.toThrow();
    });

    it('should throw if false "and" false is compared to true', async () => {
      expect(() => {
        Provable.runAndCheck(() => {
          const xFalse = Provable.witness(Bool, () => new Bool(false));
          const yFalse = Provable.witness(Bool, () => new Bool(false));

          xFalse.and(yFalse).assertEquals(new Bool(true));
        });
      }).toThrow();
    });

    it('false "and" true should return false', async () => {
      expect(() => {
        Provable.runAndCheck(() => {
          const xFalse = Provable.witness(Bool, () => new Bool(false));
          const yTrue = Provable.witness(Bool, () => new Bool(true));

          xFalse.and(yTrue).assertEquals(new Bool(false));
        });
      }).not.toThrow();
    });

    it('should throw if false "and" true is compared to true', async () => {
      expect(() => {
        Provable.runAndCheck(() => {
          const xFalse = Provable.witness(Bool, () => new Bool(false));
```

```
        const yTrue = Provable.witness(Bool, () => new Bool(true));

        xFalse.and(yTrue).assertEquals(new Bool(true));
      });
    }).toThrow();
  });
});

describe('not', () => {
  it('should return true', async () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const xTrue = Provable.witness(Bool, () => new Bool(true));
        xTrue.toField().assertEquals(new Field(1));
      });
    }).not.toThrow();
  });
  it('should return a new bool that is the negation of the input', async () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const xTrue = Provable.witness(Bool, () => new Bool(true));
        const yFalse = Provable.witness(Bool, () => new Bool(false));
        xTrue.not().assertEquals(new Bool(false));
        yFalse.not().assertEquals(new Bool(true));
      });
    }).not.toThrow();
  });

  it('should throw if input.not() is compared to input', async () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const xTrue = Provable.witness(Bool, () => new Bool(true));
        xTrue.not().assertEquals(xTrue);
      });
    }).toThrow();
  });
});

describe('or', () => {
  it('true "or" true should return true', async () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const xTrue = Provable.witness(Bool, () => new Bool(true));
        const yTrue = Provable.witness(Bool, () => new Bool(true));

        xTrue.or(yTrue).assertEquals(new Bool(true));
      });
    }).not.toThrow();
  });

  it('should throw if true "or" true is compared to false', async () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const xTrue = Provable.witness(Bool, () => new Bool(true));
        const yTrue = Provable.witness(Bool, () => new Bool(true));

        xTrue.or(yTrue).assertEquals(new Bool(false));
      });
    }).toThrow();
  });

  it('false "or" false should return false', async () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const xFalse = Provable.witness(Bool, () => new Bool(false));
        const yFalse = Provable.witness(Bool, () => new Bool(false));

        xFalse.or(yFalse).assertEquals(new Bool(false));
      });
    }).not.toThrow();
  });

  it('should throw if false "or" false is compared to true', async () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const xFalse = Provable.witness(Bool, () => new Bool(false));
        const yFalse = Provable.witness(Bool, () => new Bool(false));

        xFalse.or(yFalse).assertEquals(new Bool(true));
      });
    }).toThrow();
  });

  it('false "or" true should return true', async () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const xFalse = Provable.witness(Bool, () => new Bool(false));
        const yTrue = Provable.witness(Bool, () => new Bool(true));

        xFalse.or(yTrue).assertEquals(new Bool(true));
      });
    }).not.toThrow();
  });
});

describe('assertEquals', () => {
  it('should not throw on true "assertEqual" true', async () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(Bool, () => new Bool(true));

        x.assertEquals(x);
      });
    }).not.toThrow();
    });

    it('should throw on true "assertEquals" false', async () => {
      expect(() => {
        Provable.runAndCheck(() => {
          const x = Provable.witness(Bool, () => new Bool(true));
          const y = Provable.witness(Bool, () => new Bool(false));

          x.assertEquals(y);
        });
      }).toThrow();
    });
  });
  describe('equals', () => {
    it('should not throw on true "equals" true', async () => {
      expect(() => {
        Provable.runAndCheck(() => {
          const x = Provable.witness(Bool, () => new Bool(true));

          x.equals(x).assertEquals(true);
        });
      }).not.toThrow();
    });
    it('should throw on true "equals" false', async () => {
      expect(() => {
        Provable.runAndCheck(() => {
          const x = Provable.witness(Bool, () => new Bool(true));
          const y = Provable.witness(Bool, () => new Bool(false));
          x.equals(y).assertEquals(true);
        });
      }).toThrow();
    });
  });
});
describe('outside circuit', () => {
  describe('toField', () => {
    it('should convert false to Field element 0', () => {
      expect(new Bool(false).toField()).toEqual(new Field(0));
    });
    it('should throw when false toField is compared to Field element other than 0 ', () => {
      expect(() => {
        expect(new Bool(false).toField()).toEqual(new Field(1));
      }).toThrow();
    });

    it('should convert true to Field element 1', () => {
      expect(new Bool(true).toField()).toEqual(new Field(1));
    });

    it('should throw when true toField is compared to Field element other than 1 ', () => {
      expect(() => {
        expect(new Bool(true).toField()).toEqual(new Field(0));
    }).toThrow();
  });
});

describe('toFields', () => {
  it('should return an array of Fields', () => {
    const x = new Bool(false);
    const fieldArr = x.toFields();
    const isArr = Array.isArray(fieldArr);
    expect(isArr).toBe(true);
    expect(fieldArr[0]).toEqual(new Field(0));
  });
});
describe('and', () => {
  it('true "and" true should return true', async () => {
    const xTrue = new Bool(true);
    const yTrue = new Bool(true);
    expect(xTrue.and(yTrue)).toEqual(new Bool(true));
  });

  it('should throw if true "and" true is compared to false', async () => {
    expect(() => {
      const xTrue = new Bool(true);
      const yTrue = new Bool(true);
      expect(xTrue.and(yTrue)).toEqual(new Bool(false));
    }).toThrow();
  });

  it('false "and" false should return false', async () => {
    const xFalse = new Bool(false);
    const yFalse = new Bool(false);
    expect(xFalse.and(yFalse)).toEqual(new Bool(false));
  });

  it('should throw if false "and" false is compared to true', async () => {
    expect(() => {
      const xFalse = new Bool(false);
      const yFalse = new Bool(false);
      expect(xFalse.and(yFalse)).toEqual(new Bool(true));
    }).toThrow();
  });

  it('false "and" true should return false', async () => {
    const xFalse = new Bool(false);
    const yTrue = new Bool(true);
    expect(xFalse.and(yTrue)).toEqual(new Bool(false));
  });

  it('should throw if false "and" true is compared to true', async () => {
    expect(() => {
      const xFalse = new Bool(false);
      const yFalse = new Bool(false);
      expect(xFalse.and(yFalse)).toEqual(new Bool(true));
    }).toThrow();
  });
});
describe('not', () => {
  it('should return a new bool that is the negation of the input', async () => {
    const xTrue = new Bool(true);
    const yFalse = new Bool(false);
    expect(xTrue.not()).toEqual(new Bool(false));
    expect(yFalse.not()).toEqual(new Bool(true));
  });

  it('should throw if input.not() is compared to input', async () => {
    expect(() => {
      const xTrue = new Bool(true);

      xTrue.not().assertEquals(xTrue);
    }).toThrow();
  });
});

describe('or', () => {
  it('true "or" true should return true', async () => {
    const xTrue = new Bool(true);
    const yTrue = new Bool(true);

    xTrue.or(yTrue).assertEquals(new Bool(true));
  });

  it('should throw if true "or" true is compared to false', async () => {
    expect(() => {
      const xTrue = new Bool(true);
      const yTrue = new Bool(true);

      expect(xTrue.or(yTrue)).toEqual(new Bool(false));
    }).toThrow();
  });

  it('false "or" false should return false', async () => {
    const xFalse = new Bool(false);
    const yFalse = new Bool(false);

    expect(xFalse.or(yFalse)).toEqual(new Bool(false));
  });

  it('should throw if false "or" false is compared to true', async () => {
    expect(() => {
      const xFalse = new Bool(false);
      const yFalse = new Bool(false);

      expect(xFalse.or(yFalse)).toEqual(new Bool(true));
    }).toThrow();
    });

    it('false "or" true should return true', async () => {
      const xFalse = new Bool(false);
      const yTrue = new Bool(true);
      xFalse.or(yTrue).assertEquals(new Bool(true));
    });
  });
  describe('toBoolean', () => {
    it('should return a true javascript boolean', () => {
      const xTrue = new Bool(true);
      expect(xTrue.toBoolean()).toEqual(true);
    });
    it('should return a false javascript boolean', () => {
      const xFalse = new Bool(false);
      expect(xFalse.toBoolean()).toEqual(false);
    });
  });
 });
});
```

</file>

<file>

## path: /src/lib/primitives.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/primitives.unit-test.ts

```
import { Circuit, circuitMain } from './circuit.js';
import { UInt64, UInt32 } from './int.js';
import { expect } from 'expect';
import { Provable } from './provable.js';

class Primitives extends Circuit {
  @circuitMain
  static main() {
    // division
    let x64 = Provable.witness(UInt64, () => UInt64.from(10));
    x64.div(2).assertEquals(UInt64.from(5));
    let x32 = Provable.witness(UInt32, () => UInt32.from(15));
    x32.div(4).assertEquals(UInt32.from(3));
  }
}

let keypair = await Primitives.generateKeypair();
let proof = await Primitives.prove([], [], keypair);
let ok = await Primitives.verify([], keypair.verificationKey(), proof);

expect(ok).toEqual(true);

console.log('primitive operations in the circuit are working!      ');
```

</file>

<file>

## path: /src/lib/proof-system/cache.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/proof-system/cache.ts

```
import {
  writeFileSync,
  readFileSync,
  mkdirSync,
  resolve,
  cacheDir,
} from '../util/fs.js';
import { jsEnvironment } from '../../bindings/crypto/bindings/env.js';

// external API
export { Cache, CacheHeader };

// internal API
export { readCache, writeCache, withVersion, cacheHeaderVersion };

/**
 * Interface for storing and retrieving values, for caching.
 * &nbsp;read()&nbsp; and &nbsp;write()&nbsp; can just throw errors on failure.
 *
 * The data that will be passed to the cache for writing is exhaustively described by the {@link
CacheHeader} type.
 * It represents one of the following:
 * - The SRS. This is a deterministic lists of curve points (one per curve) that needs to be generated just
once,
 *   to be used for polynomial commitments.
 * - Lagrange basis commitments. Similar to the SRS, this will be created once for every power-of-2 circuit
size.
 * - Prover and verifier keys for every compiled circuit.
 *
 * Per smart contract or ZkProgram, several different keys are created:
 * - a step prover key (&nbsp;step-pk&nbsp;) and verification key (&nbsp;step-vk&nbsp;) _for every
method_.
 * - a wrap prover key (&nbsp;wrap-pk&nbsp;) and verification key (&nbsp;wrap-vk&nbsp;) for the entire
contract.
 */
type Cache = {
  /**
   * Read a value from the cache.
   *
   * @param header A small header to identify what is read from the cache.
   */
  read(header: CacheHeader): Uint8Array | undefined;

  /**
   * Write a value to the cache.
   *
   * @param header A small header to identify what is written to the cache. This will be used by
&nbsp;read()&nbsp; to retrieve the data.
   * @param value The value to write to the cache, as a byte array.
   */
  write(header: CacheHeader, value: Uint8Array): void;

  /**
   * Indicates whether the cache is writable.
   */
  canWrite: boolean;

  /**
   * If &nbsp;debug&nbsp; is toggled, &nbsp;read()&nbsp; and &nbsp;write()&nbsp; errors are logged to the
console.
   *
   * By default, cache errors are silent, because they don't necessarily represent an error condition,
   * but could just be a cache miss, or file system permissions incompatible with writing data.
   */
  debug?: boolean;
};

const cacheHeaderVersion = 1;

type CommonHeader = {
  /**
   * Header version to avoid parsing incompatible headers.
   */
  version: number;
  /**
   * An identifier that is persistent even as versions of the data change. Safe to use as a file path.
   */
  persistentId: string;
  /**
   * A unique identifier for the data to be read. Safe to use as a file path.
   */
  uniqueId: string;
  /**
   * Specifies whether the data to be read is a utf8-encoded string or raw binary data. This was added
   * because node's &nbsp;fs.readFileSync&nbsp; returns garbage when reading string files without
specifying the encoding.
   */
  dataType: 'string' | 'bytes';
};

type StepKeyHeader<Kind> = {
  kind: Kind;
  programName: string;
  methodName: string;
  methodIndex: number;
  hash: string;
};
type WrapKeyHeader<Kind> = { kind: Kind; programName: string; hash: string };
type PlainHeader<Kind> = { kind: Kind };

/**
 * A header that is passed to the caching layer, to support rich caching strategies.
 *
 * Both &nbsp;uniqueId&nbsp; and &nbsp;programId&nbsp; can safely be used as a file path.
 */
type CacheHeader = (
  | StepKeyHeader<'step-pk'>
  | StepKeyHeader<'step-vk'>
  | WrapKeyHeader<'wrap-pk'>
  | WrapKeyHeader<'wrap-vk'>
  | PlainHeader<'srs'>
  | PlainHeader<'lagrange-basis'>
) &
  CommonHeader;

function withVersion(
  header: Omit<CacheHeader, 'version'>,
  version = cacheHeaderVersion
): CacheHeader {
  let uniqueId = &nbsp;${header.uniqueId}-${version}&nbsp;;
  return { ...header, version, uniqueId } as CacheHeader;
}
```

```
// default methods to interact with a cache

function readCache(cache: Cache, header: CacheHeader): Uint8Array | undefined;
function readCache<T>(
  cache: Cache,
  header: CacheHeader,
  transform: (x: Uint8Array) => T
): T | undefined;
function readCache<T>(
  cache: Cache,
  header: CacheHeader,
  transform?: (x: Uint8Array) => T
): T | undefined {
  try {
    let result = cache.read(header);
    if (result === undefined) {
      if (cache.debug) console.trace('cache miss');
      return undefined;
    }
    if (transform === undefined) return result as any as T;
    return transform(result);
  } catch (e) {
    if (cache.debug) console.log('Failed to read cache', e);
    return undefined;
  }
}

function writeCache(cache: Cache, header: CacheHeader, value: Uint8Array) {
  if (!cache.canWrite) return false;
  try {
    cache.write(header, value);
    return true;
  } catch (e) {
    if (cache.debug) console.log('Failed to write cache', e);
    return false;
  }
}

const None: Cache = {
  read() {
    throw Error('not available');
  },
  write() {
    throw Error('not available');
  },
  canWrite: false,
};

const FileSystem = (cacheDirectory: string, debug?: boolean): Cache => ({
  read({ persistentId, uniqueId, dataType }) {
    if (jsEnvironment !== 'node') throw Error('file system not available');
    // read current uniqueId, return data if it matches
    let currentId = readFileSync(
      resolve(cacheDirectory, &nbsp;${persistentId}.header&nbsp;),
      'utf8'
    );
    if (currentId !== uniqueId) return undefined;

    if (dataType === 'string') {
      let string = readFileSync(resolve(cacheDirectory, persistentId), 'utf8');
      return new TextEncoder().encode(string);
    } else {
      let buffer = readFileSync(resolve(cacheDirectory, persistentId));
      return new Uint8Array(buffer.buffer);
    }
  },
  write({ persistentId, uniqueId, dataType }, data) {
    if (jsEnvironment !== 'node') throw Error('file system not available');
    mkdirSync(cacheDirectory, { recursive: true });
    writeFileSync(resolve(cacheDirectory, &nbsp;${persistentId}.header&nbsp;), uniqueId, {
      encoding: 'utf8',
    });
    writeFileSync(resolve(cacheDirectory, persistentId), data, {
      encoding: dataType === 'string' ? 'utf8' : undefined,
    });
  },
  canWrite: jsEnvironment === 'node',
  debug,
});

const FileSystemDefault = FileSystem(cacheDir('o1js'));

const Cache = {
  /**
   * Store data on the file system, in a directory of your choice.
   *
   * Data will be stored in two files per cache entry: a data file and a &nbsp;.header&nbsp; file.
   * The header file just contains a unique string which is used to determine whether we can use the cached
data.
   *
   * Note: this {@link Cache} only caches data in Node.js.
   */
  FileSystem,
  /**
   * Store data on the file system, in a standard cache directory depending on the OS.
   *
   * Data will be stored in two files per cache entry: a data file and a &nbsp;.header&nbsp; file.
   * The header file just contains a unique string which is used to determine whether we can use the cached
data.
   *
   * Note: this {@link Cache} only caches data in Node.js.
   */
  FileSystemDefault,
  /**
   * Don't store anything.
   */
  None,
};
```

</file>

<file>

## path: /src/lib/proof-system/prover-keys.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/proof-system/prover-keys.ts

```
/**
 * This file provides helpers to
 * - encode and decode all 4 kinds of snark keys to/from bytes
 * - create a header which is passed to the &nbsp;Cache&nbsp; so that it can figure out where and if to read from cache
 *
 * The inputs are &nbsp;SnarkKeyHeader&nbsp; and &nbsp;SnarkKey&nbsp;, which are OCaml tagged enums defined in pickles_bindings.ml
 */
import {
  WasmPastaFpPlonkIndex,
  WasmPastaFqPlonkIndex,
} from '../../bindings/compiled/node_bindings/plonk_wasm.cjs';
import { Pickles, getWasm } from '../../snarky.js';
import { VerifierIndex } from '../../bindings/crypto/bindings/kimchi-types.js';
import { getRustConversion } from '../../bindings/crypto/bindings.js';
import { MlString } from '../ml/base.js';
import { CacheHeader, cacheHeaderVersion } from './cache.js';
import type { MethodInterface } from '../proof_system.js';

export {
  parseHeader,
  encodeProverKey,
  decodeProverKey,
  SnarkKeyHeader,
  SnarkKey,
};
export type { MlWrapVerificationKey };

// there are 4 types of snark keys in Pickles which we all handle at once
enum KeyType {
  StepProvingKey,
  StepVerificationKey,
  WrapProvingKey,
  WrapVerificationKey,
}
```

```
type SnarkKeyHeader =
  | [KeyType.StepProvingKey, MlStepProvingKeyHeader]
  | [KeyType.StepVerificationKey, MlStepVerificationKeyHeader]
  | [KeyType.WrapProvingKey, MlWrapProvingKeyHeader]
  | [KeyType.WrapVerificationKey, MlWrapVerificationKeyHeader];

type SnarkKey =
  | [KeyType.StepProvingKey, MlBackendKeyPair<WasmPastaFpPlonkIndex>]
  | [KeyType.StepVerificationKey, VerifierIndex]
  | [KeyType.WrapProvingKey, MlBackendKeyPair<WasmPastaFqPlonkIndex>]
  | [KeyType.WrapVerificationKey, MlWrapVerificationKey];

/**
 * Create &nbsp;CacheHeader&nbsp; from a &nbsp;SnarkKeyHeader&nbsp; plus some context available to
&nbsp;compile()&nbsp;
 */
function parseHeader(
  programName: string,
  methods: MethodInterface[],
  header: SnarkKeyHeader
): CacheHeader {
  let hash = Pickles.util.fromMlString(header[1][2][8]);
  switch (header[0]) {
    case KeyType.StepProvingKey:
    case KeyType.StepVerificationKey: {
      let kind = snarkKeyStringKind[header[0]];
      let methodIndex = header[1][3];
      let methodName = methods[methodIndex].methodName;
      let persistentId = sanitize(&nbsp;${kind}-${programName}-${methodName}&nbsp;);
      let uniqueId = sanitize(
        &nbsp;${kind}-${programName}-${methodIndex}-${methodName}-${hash}&nbsp;
      );
      return {
        version: cacheHeaderVersion,
        uniqueId,
        kind,
        persistentId,
        programName,
        methodName,
        methodIndex,
        hash,
        dataType: snarkKeySerializationType[header[0]],
      };
    }
    case KeyType.WrapProvingKey:
    case KeyType.WrapVerificationKey: {
      let kind = snarkKeyStringKind[header[0]];
      let dataType = snarkKeySerializationType[header[0]];
      let persistentId = sanitize(&nbsp;${kind}-${programName}&nbsp;);
      let uniqueId = sanitize(&nbsp;${kind}-${programName}-${hash}&nbsp;);
      return {
        version: cacheHeaderVersion,
        uniqueId,
        kind,
        persistentId,
        programName,
        hash,
        dataType,
      };
    }
  }
}

/**
 * Encode a snark key to bytes
 */
function encodeProverKey(value: SnarkKey): Uint8Array {
  let wasm = getWasm();
  switch (value[0]) {
    case KeyType.StepProvingKey: {
      let index = value[1][1];
      let encoded = wasm.caml_pasta_fp_plonk_index_encode(index);
      return encoded;
    }
    case KeyType.StepVerificationKey: {
      let vkMl = value[1];
      const rustConversion = getRustConversion(getWasm());
      let vkWasm = rustConversion.fp.verifierIndexToRust(vkMl);
      let string = wasm.caml_pasta_fp_plonk_verifier_index_serialize(vkWasm);
      return new TextEncoder().encode(string);
    }
    case KeyType.WrapProvingKey: {
      let index = value[1][1];
      let encoded = wasm.caml_pasta_fq_plonk_index_encode(index);
      return encoded;
    }
    case KeyType.WrapVerificationKey: {
      let vk = value[1];
      let string = Pickles.encodeVerificationKey(vk);
      return new TextEncoder().encode(string);
    }
    default:
      value satisfies never;
      throw Error('todo');
  }
}

/**
 * Decode bytes to a snark key with the help of its header
 */
function decodeProverKey(header: SnarkKeyHeader, bytes: Uint8Array): SnarkKey {
  let wasm = getWasm();
  switch (header[0]) {
    case KeyType.StepProvingKey: {
```

```
    let srs = Pickles.loadSrsFp();
    let index = wasm.caml_pasta_fp_plonk_index_decode(bytes, srs);
    let cs = header[1][4];
    return [KeyType.StepProvingKey, [0, index, cs]];
  }
  case KeyType.StepVerificationKey: {
    let srs = Pickles.loadSrsFp();
    let string = new TextDecoder().decode(bytes);
    let vkWasm = wasm.caml_pasta_fp_plonk_verifier_index_deserialize(
      srs,
      string
    );
    const rustConversion = getRustConversion(getWasm());
    let vkMl = rustConversion.fp.verifierIndexFromRust(vkWasm);
    return [KeyType.StepVerificationKey, vkMl];
  }
  case KeyType.WrapProvingKey: {
    let srs = Pickles.loadSrsFq();
    let index = wasm.caml_pasta_fq_plonk_index_decode(bytes, srs);
    let cs = header[1][3];
    return [KeyType.WrapProvingKey, [0, index, cs]];
  }
  case KeyType.WrapVerificationKey: {
    let string = new TextDecoder().decode(bytes);
    let vk = Pickles.decodeVerificationKey(string);
    return [KeyType.WrapVerificationKey, vk];
  }
  default:
    header satisfies never;
    throw Error('todo');
 }
}

/**
 * Sanitize a string so that it can be used as a file name
 */
function sanitize(string: string): string {
 return string.toLowerCase().replace(/[^a-z0-9_-]/g, '_');
}

const snarkKeyStringKind = {
 [KeyType.StepProvingKey]: 'step-pk',
 [KeyType.StepVerificationKey]: 'step-vk',
 [KeyType.WrapProvingKey]: 'wrap-pk',
 [KeyType.WrapVerificationKey]: 'wrap-vk',
} as const;

const snarkKeySerializationType = {
 [KeyType.StepProvingKey]: 'bytes',
 [KeyType.StepVerificationKey]: 'string',
 [KeyType.WrapProvingKey]: 'bytes',
 [KeyType.WrapVerificationKey]: 'string',
} as const;

// pickles types

// Plonk_constraint_system.Make()().t

class MlConstraintSystem {
  // opaque type
}

// Dlog_plonk_based_keypair.Make().t

type MlBackendKeyPair<WasmIndex> = [
  _: 0,
  index: WasmIndex,
  cs: MlConstraintSystem
];

// Snarky_keys_header.t

type MlSnarkKeysHeader = [
  _: 0,
  headerVersion: number,
  kind: [_: 0, type: MlString, identifier: MlString],
  constraintConstants: unknown,
  commits: unknown,
  length: number,
  commitDate: MlString,
  constraintSystemHash: MlString,
  identifyingHash: MlString
];

// Pickles.Cache.{Step,Wrap}.Key.Proving.t

type MlStepProvingKeyHeader = [
  _: 0,
  typeEqual: number,
  snarkKeysHeader: MlSnarkKeysHeader,
  index: number,
  constraintSystem: MlConstraintSystem
];

type MlStepVerificationKeyHeader = [
  _: 0,
  typeEqual: number,
  snarkKeysHeader: MlSnarkKeysHeader,
  index: number,
  digest: unknown
];

type MlWrapProvingKeyHeader = [
  _: 0,
  typeEqual: number,
  snarkKeysHeader: MlSnarkKeysHeader,
  constraintSystem: MlConstraintSystem
];

type MlWrapVerificationKeyHeader = [
  _: 0,
  typeEqual: number,
  snarkKeysHeader: MlSnarkKeysHeader,
  digest: unknown
];

// Pickles.Verification_key.t

class MlWrapVerificationKey {
  // opaque type
}
```

</file>

<file>

## path: /src/lib/proof_system.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/proof_system.ts

```
import {
  EmptyNull,
  EmptyUndefined,
  EmptyVoid,
} from '../bindings/lib/generic.js';
import { withThreadPool } from '../bindings/js/wrapper.js';
import {
  ProvablePure,
  Pickles,
  FeatureFlags,
  MlFeatureFlags,
  Gate,
  GateType,
} from '../snarky.js';
import { Field, Bool } from './core.js';
import {
  FlexibleProvable,
  FlexibleProvablePure,
  InferProvable,
  ProvablePureExtended,
  provablePure,
  toConstant,
} from './circuit_value.js';
import { Provable } from './provable.js';
import { assert, prettifyStacktracePromise } from './errors.js';
import { snarkContext } from './provable-context.js';
import { hashConstant } from './hash.js';
import { MlArray, MlBool, MlResult, MlPair, MlUnit } from './ml/base.js';
import { MlFieldArray, MlFieldConstArray } from './ml/fields.js';
import { FieldConst, FieldVar } from './field.js';
import { Cache, readCache, writeCache } from './proof-system/cache.js';
import {
  decodeProverKey,
  encodeProverKey,
  parseHeader,
} from './proof-system/prover-keys.js';
import { setSrsCache, unsetSrsCache } from '../bindings/crypto/bindings/srs.js';

// public API
export {
  Proof,
  SelfProof,
  JsonProof,
  ZkProgram,
  ExperimentalZkProgram,
  verify,
  Empty,
  Undefined,
  Void,
};

// internal API
export {
  CompiledTag,
  sortMethodArguments,
  getPreviousProofsForProver,
  MethodInterface,
  GenericArgument,
  picklesRuleFromFunction,
  compileProgram,
  analyzeMethod,
  emptyValue,
  emptyWitness,
  synthesizeMethodArguments,
  methodArgumentsToConstant,
  methodArgumentTypesAndValues,
  isAsFields,
  Prover,
  dummyBase64Proof,
};

type Undefined = undefined;
const Undefined: ProvablePureExtended<undefined, null> =
  EmptyUndefined<Field>();
type Empty = Undefined;
const Empty = Undefined;
type Void = undefined;
```

```
const Void: ProvablePureExtended<void, null> = EmptyVoid<Field>();

class Proof<Input, Output> {
  static publicInputType: FlexibleProvablePure<any> = undefined as any;
  static publicOutputType: FlexibleProvablePure<any> = undefined as any;
  static tag: () => { name: string } = () => {
    throw Error(
      &nbsp;You cannot use the \&nbsp;Proof\&nbsp; class directly. Instead, define a subclass:\n&nbsp; +
        &nbsp;class MyProof extends Proof<PublicInput, PublicOutput> { ... }&nbsp;
    );
  };
  publicInput: Input;
  publicOutput: Output;
  proof: Pickles.Proof;
  maxProofsVerified: 0 | 1 | 2;
  shouldVerify = Bool(false);

  verify() {
    this.shouldVerify = Bool(true);
  }
  verifyIf(condition: Bool) {
    this.shouldVerify = condition;
  }
  toJSON(): JsonProof {
    let type = getStatementType(this.constructor as any);
    return {
      publicInput: type.input.toFields(this.publicInput).map(String),
      publicOutput: type.output.toFields(this.publicOutput).map(String),
      maxProofsVerified: this.maxProofsVerified,
      proof: Pickles.proofToBase64([this.maxProofsVerified, this.proof]),
    };
  }
  static fromJSON<S extends Subclass<typeof Proof>>(
    this: S,
    {
      maxProofsVerified,
      proof: proofString,
      publicInput: publicInputJson,
      publicOutput: publicOutputJson,
    }: JsonProof
  ): Proof<
    InferProvable<S['publicInputType']>,
    InferProvable<S['publicOutputType']>
  > {
    let [, proof] = Pickles.proofOfBase64(proofString, maxProofsVerified);
    let type = getStatementType(this);
    let publicInput = type.input.fromFields(publicInputJson.map(Field));
    let publicOutput = type.output.fromFields(publicOutputJson.map(Field));
    return new this({
      publicInput,
      publicOutput,
      proof,
      maxProofsVerified,
    }) as any;
  }

  constructor({
    proof,
    publicInput,
    publicOutput,
    maxProofsVerified,
  }: {
    proof: Pickles.Proof;
    publicInput: Input;
    publicOutput: Output;
    maxProofsVerified: 0 | 1 | 2;
  }) {
    this.publicInput = publicInput;
    this.publicOutput = publicOutput;
    this.proof = proof; // TODO optionally convert from string?
    this.maxProofsVerified = maxProofsVerified;
  }

  /**
   * Dummy proof. This can be useful for ZkPrograms that handle the base case in the same
   * method as the inductive case, using a pattern like this:
   *
   * &nbsp;&nbsp;&nbsp;ts
   * method(proof: SelfProof<I, O>, isRecursive: Bool) {
   *   proof.verifyIf(isRecursive);
   *   // ...
   * }
   * &nbsp;&nbsp;&nbsp;
   *
   * To use such a method in the base case, you need a dummy proof:
   *
   * &nbsp;&nbsp;&nbsp;ts
   * let dummy = await MyProof.dummy(publicInput, publicOutput, 1);
   * await myProgram.myMethod(dummy, Bool(false));
   * &nbsp;&nbsp;&nbsp;
   *
   * **Note**: The types of &nbsp;publicInput&nbsp; and &nbsp;publicOutput&nbsp;, as well as the
  &nbsp;maxProofsVerified&nbsp; parameter,
   * must match your ZkProgram. &nbsp;maxProofsVerified&nbsp; is the maximum number of proofs that
  any of your methods take as arguments.
   */
  static async dummy<Input, OutPut>(
    publicInput: Input,
    publicOutput: OutPut,
    maxProofsVerified: 0 | 1 | 2,
    domainLog2: number = 14
  ): Promise<Proof<Input, OutPut>> {
    let dummyRaw = await dummyProof(maxProofsVerified, domainLog2);
    return new this({
     publicInput,
     publicOutput,
     proof: dummyRaw,
     maxProofsVerified,
    });
  }
  }

  async function verify(
  proof: Proof<any, any> | JsonProof,
  verificationKey: string
  ) {
  let picklesProof: Pickles.Proof;
  let statement: Pickles.Statement<FieldConst>;
  if (typeof proof.proof === 'string') {
    // json proof
    [, picklesProof] = Pickles.proofOfBase64(
     proof.proof,
     proof.maxProofsVerified
    );
    let input = MlFieldConstArray.to(
     (proof as JsonProof).publicInput.map(Field)
    );
    let output = MlFieldConstArray.to(
     (proof as JsonProof).publicOutput.map(Field)
    );
    statement = MlPair(input, output);
  } else {
    // proof class
    picklesProof = proof.proof;
    let type = getStatementType(proof.constructor as any);
    let input = toFieldConsts(type.input, proof.publicInput);
    let output = toFieldConsts(type.output, proof.publicOutput);
    statement = MlPair(input, output);
  }
  return prettifyStacktracePromise(
    withThreadPool(() =>
     Pickles.verify(statement, picklesProof, verificationKey)
    )
  );
  }

  type JsonProof = {
  publicInput: string[];
  publicOutput: string[];
  maxProofsVerified: 0 | 1 | 2;
  proof: string;
  };
  type CompiledTag = unknown;

  let compiledTags = new WeakMap<any, CompiledTag>();
  let CompiledTag = {
  get(tag: any): CompiledTag | undefined {
    return compiledTags.get(tag);
  },
  store(tag: any, compiledTag: CompiledTag) {
    compiledTags.set(tag, compiledTag);
  },
};

function ZkProgram<
  StatementType extends {
    publicInput?: FlexibleProvablePure<any>;
    publicOutput?: FlexibleProvablePure<any>;
  },
  Types extends {
    // TODO: how to prevent a method called &nbsp;compile&nbsp; from type-checking?
    [I in string]: Tuple<PrivateInput>;
  }
>(
  config: StatementType & {
    name: string;
    methods: {
      [I in keyof Types]: Method<
        InferProvableOrUndefined<Get<StatementType, 'publicInput'>>,
        InferProvableOrVoid<Get<StatementType, 'publicOutput'>>,
        Types[I]
      >;
    };
    overrideWrapDomain?: 0 | 1 | 2;
  }
): {
  name: string;
  compile: (options?: { cache: Cache }) => Promise<{ verificationKey: string }>;
  verify: (
    proof: Proof<
      InferProvableOrUndefined<Get<StatementType, 'publicInput'>>,
      InferProvableOrVoid<Get<StatementType, 'publicOutput'>>
    >
  ) => Promise<boolean>;
  digest: () => string;
  analyzeMethods: () => ReturnType<typeof analyzeMethod>[];
  publicInputType: ProvableOrUndefined<Get<StatementType, 'publicInput'>>;
  publicOutputType: ProvableOrVoid<Get<StatementType, 'publicOutput'>>;
} & {
  [I in keyof Types]: Prover<
    InferProvableOrUndefined<Get<StatementType, 'publicInput'>>,
    InferProvableOrVoid<Get<StatementType, 'publicOutput'>>,
    Types[I]
  >;
} {
  let methods = config.methods;
  let publicInputType: ProvablePure<any> = config.publicInput! ?? Undefined;
  let publicOutputType: ProvablePure<any> = config.publicOutput! ?? Void;
  let selfTag = { name: config.name };
  type PublicInput = InferProvableOrUndefined<
    Get<StatementType, 'publicInput'>
  >;
  type PublicOutput = InferProvableOrVoid<Get<StatementType, 'publicOutput'>>;

  class SelfProof extends Proof<PublicInput, PublicOutput> {
    static publicInputType = publicInputType;
    static publicOutputType = publicOutputType;
    static tag = () => selfTag;
  }

  let keys: (keyof Types & string)[] = Object.keys(methods).sort(); // need to have methods in (any) fixed
order
  let methodIntfs = keys.map((key) =>
    sortMethodArguments('program', key, methods[key].privateInputs, SelfProof)
  );
  let methodFunctions = keys.map((key) => methods[key].method);
  let maxProofsVerified = getMaxProofsVerified(methodIntfs);

  function analyzeMethods() {
    return methodIntfs.map((methodEntry, i) =>
      analyzeMethod(publicInputType, methodEntry, methodFunctions[i])
    );
  }

  let compileOutput:
    | {
        provers: Pickles.Prover[];
        verify: (
          statement: Pickles.Statement<FieldConst>,
          proof: Pickles.Proof
        ) => Promise<boolean>;
      }
    | undefined;

  async function compile({ cache = Cache.FileSystemDefault } = {}) {
    let methodsMeta = analyzeMethods();
    let gates = methodsMeta.map((m) => m.gates);
    let { provers, verify, verificationKey } = await compileProgram({
      publicInputType,
      publicOutputType,
      methodIntfs,
      methods: methodFunctions,
      gates,
      proofSystemTag: selfTag,
      cache,
      overrideWrapDomain: config.overrideWrapDomain,
    });
    compileOutput = { provers, verify };
    return { verificationKey: verificationKey.data };
```

```
}

function toProver<K extends keyof Types & string>(
  key: K,
  i: number
): [K, Prover<PublicInput, PublicOutput, Types[K]>] {
  async function prove_(
    publicInput: PublicInput,
    ...args: TupleToInstances<Types[typeof key]>
  ): Promise<Proof<PublicInput, PublicOutput>> {
    let picklesProver = compileOutput?.provers?.[i];
    if (picklesProver === undefined) {
      throw Error(
        &nbsp;Cannot prove execution of program.${key}(), no prover found. &nbsp; +
          &nbsp;Try calling \&nbsp;await program.compile()\&nbsp; first, this will cache provers in the
background.&nbsp;
      );
    }
    let publicInputFields = toFieldConsts(publicInputType, publicInput);
    let previousProofs = MlArray.to(
      getPreviousProofsForProver(args, methodIntfs[i])
    );

    let id = snarkContext.enter({ witnesses: args, inProver: true });
    let result: UnwrapPromise<ReturnType<typeof picklesProver>>;
    try {
      result = await picklesProver(publicInputFields, previousProofs);
    } finally {
      snarkContext.leave(id);
    }
    let [publicOutputFields, proof] = MlPair.from(result);
    let publicOutput = fromFieldConsts(publicOutputType, publicOutputFields);
    class ProgramProof extends Proof<PublicInput, PublicOutput> {
      static publicInputType = publicInputType;
      static publicOutputType = publicOutputType;
      static tag = () => selfTag;
    }
    return new ProgramProof({
      publicInput,
      publicOutput,
      proof,
      maxProofsVerified,
    });
  }
  let prove: Prover<PublicInput, PublicOutput, Types[K]>;
  if (
    (publicInputType as any) === Undefined ||
    (publicInputType as any) === Void
  ) {
    prove = ((...args: TupleToInstances<Types[typeof key]>) =>
      (prove_ as any)(undefined, ...args)) as any;
  } else {
    prove = prove_ as any;
  }
  return [key, prove];
 }
 let provers = Object.fromEntries(keys.map(toProver)) as {
  [I in keyof Types]: Prover<PublicInput, PublicOutput, Types[I]>;
 };

 function verify(proof: Proof<PublicInput, PublicOutput>) {
  if (compileOutput?.verify === undefined) {
    throw Error(
      &nbsp;Cannot verify proof, verification key not found. Try calling \&nbsp;await
program.compile()\&nbsp; first.&nbsp;
    );
  }
  let statement = MlPair(
    toFieldConsts(publicInputType, proof.publicInput),
    toFieldConsts(publicOutputType, proof.publicOutput)
  );
  return compileOutput.verify(statement, proof.proof);
 }

 function digest() {
  let methodData = methodIntfs.map((methodEntry, i) =>
    analyzeMethod(publicInputType, methodEntry, methodFunctions[i])
  );
  let hash = hashConstant(
    Object.values(methodData).map((d) => Field(BigInt('0x' + d.digest)))
  );
  return hash.toBigInt().toString(16);
 }

 return Object.assign(
  selfTag,
  {
    compile,
    verify,
    digest,
    publicInputType: publicInputType as ProvableOrUndefined<
      Get<StatementType, 'publicInput'>
    >,
    publicOutputType: publicOutputType as ProvableOrVoid<
      Get<StatementType, 'publicOutput'>
    >,
    analyzeMethods,
  },
  provers
 );
}

let i = 0;
```

```
class SelfProof<PublicInput, PublicOutput> extends Proof<
  PublicInput,
  PublicOutput
> {}

function sortMethodArguments(
  programName: string,
  methodName: string,
  privateInputs: unknown[],
  selfProof: Subclass<typeof Proof>
): MethodInterface {
  let witnessArgs: Provable<unknown>[] = [];
  let proofArgs: Subclass<typeof Proof>[] = [];
  let allArgs: { type: 'proof' | 'witness' | 'generic'; index: number }[] = [];
  let genericArgs: Subclass<typeof GenericArgument>[] = [];
  for (let i = 0; i < privateInputs.length; i++) {
    let privateInput = privateInputs[i];
    if (isProof(privateInput)) {
      if (privateInput === Proof) {
        throw Error(
          &nbsp;You cannot use the \&nbsp;Proof\&nbsp; class directly. Instead, define a subclass:\n&nbsp; +
            &nbsp;class MyProof extends Proof<PublicInput, PublicOutput> { ... }&nbsp;
        );
      }
      allArgs.push({ type: 'proof', index: proofArgs.length });
      if (privateInput === SelfProof) {
        proofArgs.push(selfProof);
      } else {
        proofArgs.push(privateInput);
      }
    } else if (isAsFields(privateInput)) {
      allArgs.push({ type: 'witness', index: witnessArgs.length });
      witnessArgs.push(privateInput);
    } else if (isGeneric(privateInput)) {
      allArgs.push({ type: 'generic', index: genericArgs.length });
      genericArgs.push(privateInput);
    } else {
      throw Error(
        &nbsp;Argument ${
          i + 1
        } of method ${methodName} is not a provable type: ${privateInput}&nbsp;
      );
    }
  }
  if (proofArgs.length > 2) {
    throw Error(
      &nbsp;${programName}.${methodName}() has more than two proof arguments, which is not
supported.\n&nbsp; +
        &nbsp;Suggestion: You can merge more than two proofs by merging two at a time in a binary
tree.&nbsp;
    );
  }
  return {
    methodName,
    witnessArgs,
    proofArgs,
    allArgs,
    genericArgs,
  };
}

function isAsFields(
  type: unknown
): type is Provable<unknown> & ObjectConstructor {
  return (
    (typeof type === 'function' || typeof type === 'object') &&
    type !== null &&
    ['toFields', 'fromFields', 'sizeInFields', 'toAuxiliary'].every(
      (s) => s in type
    )
  );
}
function isProof(type: unknown): type is typeof Proof {
  // the second case covers subclasses
  return (
    type === Proof ||
    (typeof type === 'function' && type.prototype instanceof Proof)
  );
}

class GenericArgument {
  isEmpty: boolean;
  constructor(isEmpty = false) {
    this.isEmpty = isEmpty;
  }
}
let emptyGeneric = () => new GenericArgument(true);

function isGeneric(type: unknown): type is typeof GenericArgument {
  // the second case covers subclasses
  return (
    type === GenericArgument ||
    (typeof type === 'function' && type.prototype instanceof GenericArgument)
  );
}

function getPreviousProofsForProver(
  methodArgs: any[],
  { allArgs }: MethodInterface
) {
  let previousProofs: Pickles.Proof[] = [];
  for (let i = 0; i < allArgs.length; i++) {
    let arg = allArgs[i];
    if (arg.type === 'proof') {
      previousProofs[arg.index] = (methodArgs[i] as Proof<any, any>).proof;
    }
  }
  return previousProofs;
}

type MethodInterface = {
  methodName: string;
  // TODO: unify types of arguments
  // "circuit types" should be flexible enough to encompass proofs and callback arguments
  witnessArgs: Provable<unknown>[];
  proofArgs: Subclass<typeof Proof>[];
  genericArgs: Subclass<typeof GenericArgument>[];
  allArgs: { type: 'witness' | 'proof' | 'generic'; index: number }[];
  returnType?: Provable<any>;
};

// reasonable default choice for &nbsp;overrideWrapDomain&nbsp;
const maxProofsToWrapDomain = { 0: 0, 1: 1, 2: 1 } as const;

async function compileProgram({
  publicInputType,
  publicOutputType,
  methodIntfs,
  methods,
  gates,
  proofSystemTag,
  cache,
  overrideWrapDomain,
}: {
  publicInputType: ProvablePure<any>;
  publicOutputType: ProvablePure<any>;
  methodIntfs: MethodInterface[];
  methods: ((...args: any) => void)[];
  gates: Gate[][];
  proofSystemTag: { name: string };
  cache: Cache;
  overrideWrapDomain?: 0 | 1 | 2;
}) {
  let rules = methodIntfs.map((methodEntry, i) =>
    picklesRuleFromFunction(
      publicInputType,
      publicOutputType,
      methods[i],
      proofSystemTag,
      methodEntry,
      gates[i]
    )
  );
  let maxProofs = getMaxProofsVerified(methodIntfs);
  overrideWrapDomain ??= maxProofsToWrapDomain[maxProofs];
```

```
let picklesCache: Pickles.Cache = [
  0,
  function read_(mlHeader) {
    let header = parseHeader(proofSystemTag.name, methodIntfs, mlHeader);
    let result = readCache(cache, header, (bytes) =>
      decodeProverKey(mlHeader, bytes)
    );
    if (result === undefined) return MlResult.unitError();
    return MlResult.ok(result);
  },
  function write_(mlHeader, value) {
    if (!cache.canWrite) return MlResult.unitError();

    let header = parseHeader(proofSystemTag.name, methodIntfs, mlHeader);
    let didWrite = writeCache(cache, header, encodeProverKey(value));

    if (!didWrite) return MlResult.unitError();
    return MlResult.ok(undefined);
  },
  MlBool(cache.canWrite),
];

let { verificationKey, provers, verify, tag } =
  await prettifyStacktracePromise(
    withThreadPool(async () => {
      let result: ReturnType<typeof Pickles.compile>;
      let id = snarkContext.enter({ inCompile: true });
      setSrsCache(cache);
      try {
        result = Pickles.compile(MlArray.to(rules), {
          publicInputSize: publicInputType.sizeInFields(),
          publicOutputSize: publicOutputType.sizeInFields(),
          storable: picklesCache,
          overrideWrapDomain,
        });
      } finally {
        snarkContext.leave(id);
        unsetSrsCache();
      }
      let { getVerificationKey, provers, verify, tag } = result;
      CompiledTag.store(proofSystemTag, tag);
      let [, data, hash] = getVerificationKey();
      let verificationKey = { data, hash: Field(hash) };
      return { verificationKey, provers: MlArray.from(provers), verify, tag };
    })
  );
// wrap provers
let wrappedProvers = provers.map(
  (prover): Pickles.Prover =>
    async function picklesProver(
      publicInput: MlFieldConstArray,
      previousProofs: MlArray<Pickles.Proof>
    ) {
      return prettifyStacktracePromise(
        withThreadPool(() => prover(publicInput, previousProofs))
      );
    }
  );
  // wrap verify
  let wrappedVerify = async function picklesVerify(
    statement: Pickles.Statement<FieldConst>,
    proof: Pickles.Proof
  ) {
    return prettifyStacktracePromise(
      withThreadPool(() => verify(statement, proof))
    );
  };
  return {
    verificationKey,
    provers: wrappedProvers,
    verify: wrappedVerify,
    tag,
  };
}

function analyzeMethod<T>(
  publicInputType: ProvablePure<any>,
  methodIntf: MethodInterface,
  method: (...args: any) => T
) {
  return Provable.constraintSystem(() => {
    let args = synthesizeMethodArguments(methodIntf, true);
    let publicInput = emptyWitness(publicInputType);
    if (publicInputType === Undefined || publicInputType === Void)
      return method(...args);
    return method(publicInput, ...args);
  });
}

function picklesRuleFromFunction(
  publicInputType: ProvablePure<unknown>,
  publicOutputType: ProvablePure<unknown>,
  func: (...args: unknown[]) => any,
  proofSystemTag: { name: string },
  { methodName, witnessArgs, proofArgs, allArgs }: MethodInterface,
  gates: Gate[]
): Pickles.Rule {
  function main(publicInput: MlFieldArray): ReturnType<Pickles.Rule['main']> {
    let { witnesses: argsWithoutPublicInput, inProver } = snarkContext.get();
    assert(!(inProver && argsWithoutPublicInput === undefined));
    let finalArgs = [];
    let proofs: Proof<any, any>[] = [];
    let previousStatements: Pickles.Statement<FieldVar>[] = [];
    for (let i = 0; i < allArgs.length; i++) {
      let arg = allArgs[i];
      if (arg.type === 'witness') {
        let type = witnessArgs[arg.index];
        finalArgs[i] = Provable.witness(type, () => {
          return argsWithoutPublicInput?.[i] ?? emptyValue(type);
        });
      } else if (arg.type === 'proof') {
        let Proof = proofArgs[arg.index];
        let type = getStatementType(Proof);
        let proof_ = (argsWithoutPublicInput?.[i] as Proof<any, any>) ?? {
          proof: undefined,
          publicInput: emptyValue(type.input),
          publicOutput: emptyValue(type.output),
        };
        let { proof, publicInput, publicOutput } = proof_;
        publicInput = Provable.witness(type.input, () => publicInput);
        publicOutput = Provable.witness(type.output, () => publicOutput);
        let proofInstance = new Proof({ publicInput, publicOutput, proof });
        finalArgs[i] = proofInstance;
        proofs.push(proofInstance);
        let input = toFieldVars(type.input, publicInput);
        let output = toFieldVars(type.output, publicOutput);
        previousStatements.push(MlPair(input, output));
      } else if (arg.type === 'generic') {
        finalArgs[i] = argsWithoutPublicInput?.[i] ?? emptyGeneric();
      }
    }
    let result: any;
    if (publicInputType === Undefined || publicInputType === Void) {
      result = func(...finalArgs);
    } else {
      let input = fromFieldVars(publicInputType, publicInput);
      result = func(input, ...finalArgs);
    }
    // if the public output is empty, we don't evaluate &nbsp;toFields(result)&nbsp; to allow the function to
return something else in that case
    let hasPublicOutput = publicOutputType.sizeInFields() !== 0;
    let publicOutput = hasPublicOutput ? publicOutputType.toFields(result) : [];
    return {
      publicOutput: MlFieldArray.to(publicOutput),
      previousStatements: MlArray.to(previousStatements),
      shouldVerify: MlArray.to(
        proofs.map((proof) => proof.shouldVerify.toField().value)
      ),
    };
  }

  if (proofArgs.length > 2) {
    throw Error(
      &nbsp;${proofSystemTag.name}.${methodName}() has more than two proof arguments, which is not
supported.\n&nbsp; +
        &nbsp;Suggestion: You can merge more than two proofs by merging two at a time in a binary
```

```
tree.&nbsp;
    );
  }
  let proofsToVerify = proofArgs.map((Proof) => {
    let tag = Proof.tag();
    if (tag === proofSystemTag) return { isSelf: true as const };
    else {
      let compiledTag = CompiledTag.get(tag);
      if (compiledTag === undefined) {
        throw Error(
          &nbsp;${proofSystemTag.name}.compile() depends on ${tag.name}, but we cannot find compilation
output for ${tag.name}.\n&nbsp; +
            &nbsp;Try to run ${tag.name}.compile() first.&nbsp;
        );
      }
      return { isSelf: false, tag: compiledTag };
    }
  });

  let featureFlags = computeFeatureFlags(gates);

  return {
    identifier: methodName,
    main,
    featureFlags,
    proofsToVerify: MlArray.to(proofsToVerify),
  };
}

function synthesizeMethodArguments(
  { allArgs, proofArgs, witnessArgs }: MethodInterface,
  asVariables = false
) {
  let args = [];
  let empty = asVariables ? emptyWitness : emptyValue;
  for (let arg of allArgs) {
    if (arg.type === 'witness') {
      args.push(empty(witnessArgs[arg.index]));
    } else if (arg.type === 'proof') {
      let Proof = proofArgs[arg.index];
      let type = getStatementType(Proof);
      let publicInput = empty(type.input);
      let publicOutput = empty(type.output);
      args.push(new Proof({ publicInput, publicOutput, proof: undefined }));
    } else if (arg.type === 'generic') {
      args.push(emptyGeneric());
    }
  }
  return args;
}

function methodArgumentsToConstant(
  { allArgs, proofArgs, witnessArgs }: MethodInterface,
  args: any[]
) {
  let constArgs = [];
  for (let i = 0; i < allArgs.length; i++) {
    let arg = args[i];
    let { type, index } = allArgs[i];
    if (type === 'witness') {
      constArgs.push(toConstant(witnessArgs[index], arg));
    } else if (type === 'proof') {
      let Proof = proofArgs[index];
      let type = getStatementType(Proof);
      let publicInput = toConstant(type.input, arg.publicInput);
      let publicOutput = toConstant(type.output, arg.publicOutput);
      constArgs.push(
        new Proof({ publicInput, publicOutput, proof: arg.proof })
      );
    } else if (type === 'generic') {
      constArgs.push(arg);
    }
  }
  return constArgs;
}

let Generic = EmptyNull<Field>();

type TypeAndValue<T> = { type: Provable<T>; value: T };

function methodArgumentTypesAndValues(
  { allArgs, proofArgs, witnessArgs }: MethodInterface,
  args: unknown[]
) {
  let typesAndValues: TypeAndValue<any>[] = [];
  for (let i = 0; i < allArgs.length; i++) {
    let arg = args[i];
    let { type, index } = allArgs[i];
    if (type === 'witness') {
      typesAndValues.push({ type: witnessArgs[index], value: arg });
    } else if (type === 'proof') {
      let Proof = proofArgs[index];
      let proof = arg as Proof<any, any>;
      let types = getStatementType(Proof);
      // TODO this is cumbersome, would be nicer to have a single Provable for the statement stored on Proof
      let type = provablePure({ input: types.input, output: types.output });
      let value = { input: proof.publicInput, output: proof.publicOutput };
      typesAndValues.push({ type, value });
    } else if (type === 'generic') {
      typesAndValues.push({ type: Generic, value: arg });
    }
  }
  return typesAndValues;
}
```

```
function emptyValue<T>(type: FlexibleProvable<T>): T;
function emptyValue<T>(type: Provable<T>) {
  return type.fromFields(
    Array(type.sizeInFields()).fill(Field(0)),
    type.toAuxiliary()
  );
}

function emptyWitness<T>(type: FlexibleProvable<T>): T;
function emptyWitness<T>(type: Provable<T>) {
  return Provable.witness(type, () => emptyValue(type));
}

function getStatementType<
  T,
  O,
  P extends Subclass<typeof Proof> = typeof Proof
>(Proof: P): { input: ProvablePure<T>; output: ProvablePure<O> } {
  if (
    Proof.publicInputType === undefined ||
    Proof.publicOutputType === undefined
  ) {
    throw Error(
      &nbsp;You cannot use the \&nbsp;Proof\&nbsp; class directly. Instead, define a subclass:\n&nbsp; +
        &nbsp;class MyProof extends Proof<PublicInput, PublicOutput> { ... }&nbsp;
    );
  }
  return {
    input: Proof.publicInputType as any,
    output: Proof.publicOutputType as any,
  };
}

function getMaxProofsVerified(methodIntfs: MethodInterface[]) {
  return methodIntfs.reduce(
    (acc, { proofArgs }) => Math.max(acc, proofArgs.length),
    0
  ) as any as 0 | 1 | 2;
}

function fromFieldVars<T>(type: ProvablePure<T>, fields: MlFieldArray) {
  return type.fromFields(MlFieldArray.from(fields));
}
function toFieldVars<T>(type: ProvablePure<T>, value: T) {
  return MlFieldArray.to(type.toFields(value));
}

function fromFieldConsts<T>(type: ProvablePure<T>, fields: MlFieldConstArray) {
  return type.fromFields(MlFieldConstArray.from(fields));
}
function toFieldConsts<T>(type: ProvablePure<T>, value: T) {
  return MlFieldConstArray.to(type.toFields(value));
}

ZkProgram.Proof = function <
  PublicInputType extends FlexibleProvablePure<any>,
  PublicOutputType extends FlexibleProvablePure<any>
>(program: {
  name: string;
  publicInputType: PublicInputType;
  publicOutputType: PublicOutputType;
}) {
  type PublicInput = InferProvable<PublicInputType>;
  type PublicOutput = InferProvable<PublicOutputType>;
  return class ZkProgramProof extends Proof<PublicInput, PublicOutput> {
    static publicInputType = program.publicInputType;
    static publicOutputType = program.publicOutputType;
    static tag = () => program;
  };
};

function dummyProof(maxProofsVerified: 0 | 1 | 2, domainLog2: number) {
  return withThreadPool(
    async () => Pickles.dummyProof(maxProofsVerified, domainLog2)[1]
  );
}

async function dummyBase64Proof() {
  let proof = await dummyProof(2, 15);
  return Pickles.proofToBase64([2, proof]);
}

// what feature flags to set to enable certain gate types

const gateToFlag: Partial<Record<GateType, keyof FeatureFlags>> = {
  RangeCheck0: 'rangeCheck0',
  RangeCheck1: 'rangeCheck1',
  ForeignFieldAdd: 'foreignFieldAdd',
  ForeignFieldMul: 'foreignFieldMul',
  Xor16: 'xor',
  Rot64: 'rot',
  Lookup: 'lookup',
};

function computeFeatureFlags(gates: Gate[]): MlFeatureFlags {
  let flags: FeatureFlags = {
    rangeCheck0: false,
    rangeCheck1: false,
    foreignFieldAdd: false,
    foreignFieldMul: false,
    xor: false,
    rot: false,
    lookup: false,
    runtimeTables: false,
  };
  for (let gate of gates) {
    let flag = gateToFlag[gate.type];
    if (flag !== undefined) flags[flag] = true;
  }
  return [
    0,
    MlBool(flags.rangeCheck0),
    MlBool(flags.rangeCheck1),
    MlBool(flags.foreignFieldAdd),
    MlBool(flags.foreignFieldMul),
    MlBool(flags.xor),
    MlBool(flags.rot),
    MlBool(flags.lookup),
    MlBool(flags.runtimeTables),
  ];
}

// helpers for circuit context

function Prover<ProverData>() {
  return {
    async run<Result>(
      witnesses: unknown[],
      proverData: ProverData,
      callback: () => Promise<Result>
    ) {
      let id = snarkContext.enter({ witnesses, proverData, inProver: true });
      try {
        return await callback();
      } finally {
        snarkContext.leave(id);
      }
    },
    getData(): ProverData {
      return snarkContext.get().proverData;
    },
  };
}

// helper types

type Infer<T> = T extends Subclass<typeof Proof>
  ? InstanceType<T>
  : InferProvable<T>;

type Tuple<T> = [T, ...T[]] | [];
type TupleToInstances<T> = {
  [I in keyof T]: Infer<T[I]>;
} & any[];
```

```
type Subclass<Class extends new (...args: any) => any> = (new (
  ...args: any
) => InstanceType<Class>) & {
  [K in keyof Class]: Class[K];
} & { prototype: InstanceType<Class> };

type PrivateInput = Provable<any> | Subclass<typeof Proof>;

type Method<
  PublicInput,
  PublicOutput,
  Args extends Tuple<PrivateInput>
> = PublicInput extends undefined
  ? {
      privateInputs: Args;
      method(...args: TupleToInstances<Args>): PublicOutput;
    }
  : {
      privateInputs: Args;
      method(
        publicInput: PublicInput,
        ...args: TupleToInstances<Args>
      ): PublicOutput;
    };

type Prover<
  PublicInput,
  PublicOutput,
  Args extends Tuple<PrivateInput>
> = PublicInput extends undefined
  ? (
      ...args: TupleToInstances<Args>
    ) => Promise<Proof<PublicInput, PublicOutput>>
  : (
      publicInput: PublicInput,
      ...args: TupleToInstances<Args>
    ) => Promise<Proof<PublicInput, PublicOutput>>;

type ProvableOrUndefined<A> = A extends undefined ? typeof Undefined : A;
type ProvableOrVoid<A> = A extends undefined ? typeof Void : A;

type InferProvableOrUndefined<A> = A extends undefined
  ? undefined
  : InferProvable<A>;
type InferProvableOrVoid<A> = A extends undefined ? void : InferProvable<A>;

type UnwrapPromise<P> = P extends Promise<infer T> ? T : never;

/**
 * helper to get property type from an object, in place of &nbsp;T[Key]&nbsp;
 *
 * assume &nbsp;T extends { Key?: Something }&nbsp;.
 * if we use &nbsp;Get<T, Key>&nbsp; instead of &nbsp;T[Key]&nbsp;, we allow &nbsp;T&nbsp; to be
inferred _without_ the &nbsp;Key&nbsp; key,
 * and thus retain the precise type of &nbsp;T&nbsp; during inference
 */
type Get<T, Key extends string> = T extends { [K in Key]: infer Value }
  ? Value
  : undefined;

// deprecated experimental API

function ExperimentalZkProgram<
  StatementType extends {
    publicInput?: FlexibleProvablePure<any>;
    publicOutput?: FlexibleProvablePure<any>;
  },
  Types extends {
    [I in string]: Tuple<PrivateInput>;
  }
>(
  config: StatementType & {
    name?: string;
    methods: {
      [I in keyof Types]: Method<
        InferProvableOrUndefined<Get<StatementType, 'publicInput'>>,
        InferProvableOrVoid<Get<StatementType, 'publicOutput'>>,
        Types[I]
      >;
    };
    overrideWrapDomain?: 0 | 1 | 2;
  }
) {
  let config_ = { ...config, name: config.name ?? &nbsp;Program${i++}&nbsp; };
  return ZkProgram<StatementType, Types>(config_);
}
```

</file>

<file>

## path: /src/lib/proof_system.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/proof_system.unit-test.ts

```
import { Field } from './core.js';
import { Struct } from './circuit_value.js';
import { UInt64 } from './int.js';
import { ZkProgram } from './proof_system.js';
import { expect } from 'expect';

const EmptyProgram = ZkProgram({
  name: 'empty',
  publicInput: Field,
  methods: {
    run: {
      privateInputs: [],
      method: (publicInput: Field) => {},
    },
  },
});

const emptyMethodsMetadata = EmptyProgram.analyzeMethods();
emptyMethodsMetadata.forEach((methodMetadata) => {
  expect(methodMetadata).toEqual({
    rows: 0,
    digest: '4f5ddea76d29cfcfd8c595f14e31f21b',
    result: undefined,
    gates: [],
    publicInputSize: 0,
  });
});

class CounterPublicInput extends Struct({
  current: UInt64,
  updated: UInt64,
}) {}
const CounterProgram = ZkProgram({
  name: 'counter',
  publicInput: CounterPublicInput,
  methods: {
    increment: {
      privateInputs: [UInt64],
      method: (
        { current, updated }: CounterPublicInput,
        incrementBy: UInt64
      ) => {
        const newCount = current.add(incrementBy);
        newCount.assertEquals(updated);
      },
    },
  },
});

const incrementMethodMetadata = CounterProgram.analyzeMethods()[0];
expect(incrementMethodMetadata).toEqual(expect.objectContaining({ rows: 18 }));
```

</file>

<file>

## path: /src/lib/provable-context.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/provable-context.ts

```
import { Context } from './global-context.js';
import { Gate, JsonGate, Snarky } from '../snarky.js';
import { parseHexString } from '../bindings/crypto/bigint-helpers.js';
import { prettifyStacktrace } from './errors.js';

// internal API
export {
  snarkContext,
  SnarkContext,
  asProver,
  runAndCheck,
  runUnchecked,
  constraintSystem,
  inProver,
  inAnalyze,
  inCheckedComputation,
  inCompile,
  inCompileMode,
  gatesFromJson,
};

// global circuit-related context

type SnarkContext = {
  witnesses?: unknown[];
  proverData?: any;
  inProver?: boolean;
  inCompile?: boolean;
  inCheckedComputation?: boolean;
  inAnalyze?: boolean;
  inRunAndCheck?: boolean;
  inWitnessBlock?: boolean;
};
let snarkContext = Context.create<SnarkContext>({ default: {} });

// helpers to read circuit context

function inProver() {
  return !!snarkContext.get().inProver;
}
function inCheckedComputation() {
  let ctx = snarkContext.get();
  return !!ctx.inCompile || !!ctx.inProver || !!ctx.inCheckedComputation;
}
function inCompile() {
  return !!snarkContext.get().inCompile;
}
function inAnalyze() {
  return !!snarkContext.get().inAnalyze;
}
```

```
function inCompileMode() {
  let ctx = snarkContext.get();
  return !!ctx.inCompile || !!ctx.inAnalyze;
}

// runners for provable code

function asProver(f: () => void) {
  if (inCheckedComputation()) {
    Snarky.run.asProver(f);
  } else {
    f();
  }
}

function runAndCheck(f: () => void) {
  let id = snarkContext.enter({ inCheckedComputation: true });
  try {
    Snarky.run.runAndCheck(f);
  } catch (error) {
    throw prettifyStacktrace(error);
  } finally {
    snarkContext.leave(id);
  }
}

function runUnchecked(f: () => void) {
  let id = snarkContext.enter({ inCheckedComputation: true });
  try {
    Snarky.run.runUnchecked(f);
  } catch (error) {
    throw prettifyStacktrace(error);
  } finally {
    snarkContext.leave(id);
  }
}

function constraintSystem<T>(f: () => T) {
  let id = snarkContext.enter({ inAnalyze: true, inCheckedComputation: true });
  try {
    let result: T;
    let { rows, digest, json } = Snarky.run.constraintSystem(() => {
      result = f();
    });
    let { gates, publicInputSize } = gatesFromJson(json);
    return { rows, digest, result: result! as T, gates, publicInputSize };
  } catch (error) {
    throw prettifyStacktrace(error);
  } finally {
    snarkContext.leave(id);
  }
}
```

```
// helpers

function gatesFromJson(cs: { gates: JsonGate[]; public_input_size: number }) {
  let gates: Gate[] = cs.gates.map(({ typ, wires, coeffs: hexCoeffs }) => {
    let coeffs = hexCoeffs.map(hex => parseHexString(hex).toString());
    return { type: typ, wires, coeffs };
  });
  return { publicInputSize: cs.public_input_size, gates };
}
```

</file>

<file>

## path: /src/lib/provable.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/provable.ts

```
/**
 * {@link Provable} is
 * - a namespace with tools for writing provable code
 * - the main interface for types that can be used in provable code
 */
import { Field, Bool } from './core.js';
import { Provable as Provable_, Snarky } from '../snarky.js';
import type { FlexibleProvable, ProvableExtended } from './circuit_value.js';
import { Context } from './global-context.js';
import {
  HashInput,
  InferJson,
  InferProvable,
  InferredProvable,
} from '../bindings/lib/provable-snarky.js';
import { isField } from './field.js';
import {
  inCheckedComputation,
  inProver,
  snarkContext,
  asProver,
  runAndCheck,
  runUnchecked,
  constraintSystem,
} from './provable-context.js';
import { isBool } from './bool.js';

// external API
export { Provable };

// internal API
export {
  memoizationContext,
  MemoizationContext,
  memoizeWitness,
  getBlindingValue,
};

// TODO move type declaration here
/**
 * &nbsp;Provable<T>&nbsp; is the general circuit type interface. It describes how a type &nbsp;T&nbsp; is
made up of field elements and auxiliary (non-field element) data.
 *
 * You will find this as the required input type in a few places in o1js. One convenient way to create a
&nbsp;Provable<T>&nbsp; is using &nbsp;Struct&nbsp;.
 */
type Provable<T> = Provable_<T>;

const Provable = {
  /**
   * Create a new witness. A witness, or variable, is a value that is provided as input
   * by the prover. This provides a flexible way to introduce values from outside into the circuit.
   * However, note that nothing about how the value was created is part of the proof -
&nbsp;Provable.witness&nbsp;
   * behaves exactly like user input. So, make sure that after receiving the witness you make any assertions
   * that you want to associate with it.
   * @example
   * Example for re-implementing &nbsp;Field.inv&nbsp; with the help of &nbsp;witness&nbsp;:
   * &nbsp;&nbsp;&nbsp;ts
   * let invX = Provable.witness(Field, () => {
   *   // compute the inverse of &nbsp;x&nbsp; outside the circuit, however you like!
   *   return Field.inv(x));
   * }
   * // prove that &nbsp;invX&nbsp; is really the inverse of &nbsp;x&nbsp;:
   * invX.mul(x).assertEquals(1);
   * &nbsp;&nbsp;&nbsp;
   */
  witness,
  /**
   * Proof-compatible if-statement.
   * This behaves like a ternary conditional statement in JS.
   *
   * **Warning**: Since &nbsp;Provable.if()&nbsp; is a normal JS function call, both the if and the else branch
   * are evaluated before calling it. Therefore, you can't use this function
   * to guard against execution of one of the branches. It only allows you to pick one of two values.
   *
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const condition = Bool(true);
   * const result = Provable.if(condition, Field(1), Field(2)); // returns Field(1)
   * &nbsp;&nbsp;&nbsp;
   */
  if: if_,
  /**
   * Generalization of {@link Provable.if} for choosing between more than two different cases.
   * It takes a "mask", which is an array of &nbsp;Bool&nbsp;s that contains only one &nbsp;true&nbsp;
element, a type/constructor, and an array of values of that type.
   * The result is that value which corresponds to the true element of the mask.
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * let x = Provable.switch([Bool(false), Bool(true)], Field, [Field(1), Field(2)]);
   * x.assertEquals(2);
   * &nbsp;&nbsp;&nbsp;
   */
  switch: switch_,
  /**
   * Asserts that two values are equal.
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * class MyStruct extends Struct({ a: Field, b: Bool }) {};
   * const a: MyStruct = { a: Field(0), b: Bool(false) };
   * const b: MyStruct = { a: Field(1), b: Bool(true) };
   * Provable.assertEqual(MyStruct, a, b);
   * &nbsp;&nbsp;&nbsp;
   */
  assertEqual,
  /**
   * Checks if two elements are equal.
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * class MyStruct extends Struct({ a: Field, b: Bool }) {};
   * const a: MyStruct = { a: Field(0), b: Bool(false) };
   * const b: MyStruct = { a: Field(1), b: Bool(true) };
   * const isEqual = Provable.equal(MyStruct, a, b);
   * &nbsp;&nbsp;&nbsp;
   */
  equal,
  /**
   * Creates a {@link Provable} for a generic array.
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const ProvableArray = Provable.Array(Field, 5);
   * &nbsp;&nbsp;&nbsp;
   */
  Array: provableArray,
  /**
   * Interface to log elements within a circuit. Similar to &nbsp;console.log()&nbsp;.
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const element = Field(42);
   * Provable.log(element);
   * &nbsp;&nbsp;&nbsp;
   */
  log,
  /**
   * Runs code as a prover.
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * Provable.asProver(() => {
   *   // Your prover code here
   * });
   * &nbsp;&nbsp;&nbsp;
   */
  asProver,
  /**
   * Runs provable code quickly, without creating a proof, but still checking whether constraints are satisfied.
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * Provable.runAndCheck(() => {
   *   // Your code to check here
   * });
   * &nbsp;&nbsp;&nbsp;
   */
  runAndCheck,
  /**
   * Runs provable code quickly, without creating a proof, and not checking whether constraints are satisfied.
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * Provable.runUnchecked(() => {
   *   // Your code to run here
   * });
   * &nbsp;&nbsp;&nbsp;
   */
  runUnchecked,
  /**
   * Returns information about the constraints created by the callback function.
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * const result = Provable.constraintSystem(circuit);
   * console.log(result);
   * &nbsp;&nbsp;&nbsp;
   */
  constraintSystem,
  /**
   * Checks if the code is run in prover mode.
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * if (Provable.inProver()) {
   *   // Prover-specific code
   * }
   * &nbsp;&nbsp;&nbsp;
   */
  inProver,
  /**
   * Checks if the code is run in checked computation mode.
   * @example
   * &nbsp;&nbsp;&nbsp;ts
   * if (Provable.inCheckedComputation()) {
   *   // Checked computation-specific code
   * }
   * &nbsp;&nbsp;&nbsp;
   */
  inCheckedComputation,
 };

 function witness<T, S extends FlexibleProvable<T> = FlexibleProvable<T>>(
  type: S,
  compute: () => T
 ): T {
  let ctx = snarkContext.get();

  // outside provable code, we just call the callback and return its cloned result
  if (!inCheckedComputation() || ctx.inWitnessBlock) {
    return clone(type, compute());
  }
  let proverValue: T | undefined = undefined;
  let fields: Field[];

  let id = snarkContext.enter({ ...ctx, inWitnessBlock: true });
  try {
    let [, ...fieldVars] = Snarky.exists(type.sizeInFields(), () => {
      proverValue = compute();
      let fields = type.toFields(proverValue);
      let fieldConstants = fields.map((x) => x.toConstant().value[1]);

      // TODO: enable this check
      // currently it throws for Scalar.. which seems to be flexible about what length is returned by toFields
      // if (fields.length !== type.sizeInFields()) {
      //   throw Error(
      //     &nbsp;Invalid witness. Expected ${type.sizeInFields()} field elements, got ${
      //       fields.length
      //     }.&nbsp;
      //   );
      // }
      return [0, ...fieldConstants];
    });
    fields = fieldVars.map(Field);
  } finally {
    snarkContext.leave(id);
  }

  // rebuild the value from its fields (which are now variables) and aux data
  let aux = type.toAuxiliary(proverValue);
  let value = (type as Provable<T>).fromFields(fields, aux);

  // add type-specific constraints
  type.check(value);

  return value;
 }
```

```
type ToFieldable = { toFields(): Field[] };

// general provable methods

function assertEqual<T>(type: FlexibleProvable<T>, x: T, y: T): void;
function assertEqual<T extends ToFieldable>(x: T, y: T): void;
function assertEqual(typeOrX: any, xOrY: any, yOrUndefined?: any) {
  if (yOrUndefined === undefined) {
    return assertEqualImplicit(typeOrX, xOrY);
  } else {
    return assertEqualExplicit(typeOrX, xOrY, yOrUndefined);
  }
}
function assertEqualImplicit<T extends ToFieldable>(x: T, y: T) {
  let xs = x.toFields();
  let ys = y.toFields();
  let n = checkLength('Provable.assertEqual', xs, ys);
  for (let i = 0; i < n; i++) {
    xs[i].assertEquals(ys[i]);
  }
}
function assertEqualExplicit<T>(type: Provable<T>, x: T, y: T) {
  let xs = type.toFields(x);
  let ys = type.toFields(y);
  for (let i = 0; i < xs.length; i++) {
    xs[i].assertEquals(ys[i]);
  }
}

function equal<T>(type: FlexibleProvable<T>, x: T, y: T): Bool;
function equal<T extends ToFieldable>(x: T, y: T): Bool;
function equal(typeOrX: any, xOrY: any, yOrUndefined?: any) {
  if (yOrUndefined === undefined) {
    return equalImplicit(typeOrX, xOrY);
  } else {
    return equalExplicit(typeOrX, xOrY, yOrUndefined);
  }
}
// TODO: constraints can be reduced by up to 2x for large structures by using a variant
// of the &nbsp;equals&nbsp; argument where we return 1 - z(x0 - y0)(x1 - y1)...(xn - yn)
// current version will do (1 - z0(x0 - y0))(1 - z1(x1 - y1))... + constrain each factor
function equalImplicit<T extends ToFieldable>(x: T, y: T) {
  let xs = x.toFields();
  let ys = y.toFields();
  checkLength('Provable.equal', xs, ys);
  return xs.map((x, i) => x.equals(ys[i])).reduce(Bool.and);
}
function equalExplicit<T>(type: Provable<T>, x: T, y: T) {
  let xs = type.toFields(x);
  let ys = type.toFields(y);
  return xs.map((x, i) => x.equals(ys[i])).reduce(Bool.and);
```

```
}

function if_<T>(condition: Bool, type: FlexibleProvable<T>, x: T, y: T): T;
function if_<T extends ToFieldable>(condition: Bool, x: T, y: T): T;
function if_(condition: Bool, typeOrX: any, xOrY: any, yOrUndefined?: any) {
  if (yOrUndefined === undefined) {
    return ifImplicit(condition, typeOrX, xOrY);
  } else {
    return ifExplicit(condition, typeOrX, xOrY, yOrUndefined);
  }
}

function ifField(b: Field, x: Field, y: Field) {
  // b*(x - y) + y
  // NOTE: the R1CS constraint used by Field.if_ in snarky-ml
  // leads to a different but equivalent layout (same # constraints)
  // https://github.com/o1-
labs/snarky/blob/14f8e2ff981a9c9ea48c94b2cc1d8c161301537b/src/base/utils.ml#L171
  // in the case x, y are constant, the layout is the same
  return b.mul(x.sub(y)).add(y).seal();
}

function ifExplicit<T>(condition: Bool, type: Provable<T>, x: T, y: T): T {
  let xs = type.toFields(x);
  let ys = type.toFields(y);
  let b = condition.toField();

  // simple case: b is constant - it's like a normal if statement
  if (b.isConstant()) {
    return clone(type, condition.toBoolean() ? x : y);
  }

  // if b is variable, we compute if as follows:
  // if(b, x, y)[i] = b*(x[i] - y[i]) + y[i]
  let fields = xs.map((xi, i) => ifField(b, xi, ys[i]));
  let aux = auxiliary(type, () => (condition.toBoolean() ? x : y));
  return type.fromFields(fields, aux);
}

function ifImplicit<T extends ToFieldable>(condition: Bool, x: T, y: T): T {
  let type = x.constructor;
  if (type === undefined)
    throw Error(
      &nbsp;You called Provable.if(bool, x, y) with an argument x that has no constructor, which is not
supported.\n&nbsp; +
        &nbsp;If x, y are Structs or other custom types, you can use the following:\n&nbsp; +
        &nbsp;Provable.if(bool, MyType, x, y)&nbsp;
    );
  // TODO remove second condition once we have consolidated field class back into one
  // if (type !== y.constructor) {
  if (
    type !== y.constructor &&
    !(isField(x) && isField(y)) &&
    !(isBool(x) && isBool(y))
  ) {
    throw Error(
      'Provable.if: Mismatched argument types. Try using an explicit type argument:\n' +
        &nbsp;Provable.if(bool, MyType, x, y)&nbsp;
    );
  }
  if (!('fromFields' in type && 'toFields' in type)) {
    throw Error(
      'Provable.if: Invalid argument type. Try using an explicit type argument:\n' +
        &nbsp;Provable.if(bool, MyType, x, y)&nbsp;
    );
  }
  return ifExplicit(condition, type as any as Provable<T>, x, y);
}

function switch_<T, A extends FlexibleProvable<T>>(
  mask: Bool[],
  type: A,
  values: T[]
): T {
  // picks the value at the index where mask is true
  let nValues = values.length;
  if (mask.length !== nValues)
    throw Error(
      &nbsp;Provable.switch: \&nbsp;values\&nbsp; and \&nbsp;mask\&nbsp; have different lengths
(${values.length} vs. ${mask.length}), which is not allowed.&nbsp;
    );
  let checkMask = () => {
    let nTrue = mask.filter((b) => b.toBoolean()).length;
    if (nTrue > 1) {
      throw Error(
        &nbsp;Provable.switch: \&nbsp;mask\&nbsp; must have 0 or 1 true element, found ${nTrue}.&nbsp;
      );
    }
  };
  if (mask.every((b) => b.toField().isConstant())) checkMask();
  else Provable.asProver(checkMask);
  let size = type.sizeInFields();
  let fields = Array(size).fill(Field(0));
  for (let i = 0; i < nValues; i++) {
    let valueFields = type.toFields(values[i]);
    let maskField = mask[i].toField();
    for (let j = 0; j < size; j++) {
      let maybeField = valueFields[j].mul(maskField);
      fields[j] = fields[j].add(maybeField);
    }
  }
  let aux = auxiliary(type as Provable<T>, () => {
    let i = mask.findIndex((b) => b.toBoolean());
    if (i === -1) return undefined;
    return values[i];
  });
  return (type as Provable<T>).fromFields(fields, aux);
}

// logging in provable code

function log(...args: any) {
  asProver(() => {
    let prettyArgs = [];
    for (let arg of args) {
      if (arg?.toPretty !== undefined) prettyArgs.push(arg.toPretty());
      else {
        try {
          prettyArgs.push(JSON.parse(JSON.stringify(arg)));
        } catch {
          prettyArgs.push(arg);
        }
      }
    }
    console.log(...prettyArgs);
  });
}

// helpers

function checkLength(name: string, xs: Field[], ys: Field[]) {
  let n = xs.length;
  let m = ys.length;
  if (n !== m) {
    throw Error(
      &nbsp;${name}: inputs must contain the same number of field elements, got ${n} !== ${m}&nbsp;
    );
  }
  return n;
}

function clone<T, S extends FlexibleProvable<T>>(type: S, value: T): T {
  let fields = type.toFields(value);
  let aux = type.toAuxiliary?.(value) ?? [];
  return (type as Provable<T>).fromFields(fields, aux);
}

function auxiliary<T>(type: Provable<T>, compute: () => T | undefined) {
  let aux;
  // TODO: this accepts types without .toAuxiliary(), should be changed when all snarky types are moved to
TS
  Provable.asProver(() => {
    let value = compute();
    if (value !== undefined) {
      aux = type.toAuxiliary?.(value);
    }
  });
  return aux ?? type.toAuxiliary?.() ?? [];
}

type MemoizationContext = {
  memoized: { fields: Field[]; aux: any[] }[];
  currentIndex: number;
  blindingValue: Field;
};
let memoizationContext = Context.create<MemoizationContext>();

/**
 * Like Provable.witness, but memoizes the witness during transaction construction
 * for reuse by the prover. This is needed to witness non-deterministic values.
 */
function memoizeWitness<T>(type: FlexibleProvable<T>, compute: () => T) {
  return Provable.witness<T>(type as Provable<T>, () => {
    if (!memoizationContext.has()) return compute();
    let context = memoizationContext.get();
    let { memoized, currentIndex } = context;
    let currentValue = memoized[currentIndex];
    if (currentValue === undefined) {
      let value = compute();
      let fields = type.toFields(value).map((x) => x.toConstant());
      let aux = type.toAuxiliary(value);
      currentValue = { fields, aux };
      memoized[currentIndex] = currentValue;
    }
    context.currentIndex += 1;
    return (type as Provable<T>).fromFields(
      currentValue.fields,
      currentValue.aux
    );
  });
}

function getBlindingValue() {
  if (!memoizationContext.has()) return Field.random();
  let context = memoizationContext.get();
  if (context.blindingValue === undefined) {
    context.blindingValue = Field.random();
  }
  return context.blindingValue;
}

// TODO this should return a class, like Struct, so you can just use &nbsp;class Array3 extends
Provable.Array(Field, 3) {}&nbsp;
function provableArray<A extends FlexibleProvable<any>>(
  elementType: A,
  length: number
): InferredProvable<A[]> {
  type T = InferProvable<A>;
  type TJson = InferJson<A>;
  let type = elementType as ProvableExtended<T>;
  return {
    /**
     * Returns the size of this structure in {@link Field} elements.
     * @returns size of this structure
     */
    sizeInFields() {
      let elementLength = type.sizeInFields();
      return elementLength * length;
    },
    /**
     * Serializes this structure into {@link Field} elements.
     * @returns an array of {@link Field} elements
     */
    toFields(array: T[]) {
      return array.map((e) => type.toFields(e)).flat();
    },
    /**
     * Serializes this structure's auxiliary data.
     * @returns auxiliary data
     */
    toAuxiliary(array?) {
      let array_ = array ?? Array<undefined>(length).fill(undefined);
      return array_?.map((e) => type.toAuxiliary(e));
    },

    /**
     * Deserializes an array of {@link Field} elements into this structure.
     */
    fromFields(fields: Field[], aux?: any[]) {
      let array = [];
      let size = type.sizeInFields();
      let n = length;
      for (let i = 0, offset = 0; i < n; i++, offset += size) {
        array[i] = type.fromFields(
          fields.slice(offset, offset + size),
          aux?.[i]
        );
      }
      return array;
    },
    check(array: T[]) {
      for (let i = 0; i < length; i++) {
        (type as any).check(array[i]);
      }
    },
    /**
     * Encodes this structure into a JSON-like object.
     */
    toJSON(array) {
      if (!('toJSON' in type)) {
        throw Error('circuitArray.toJSON: element type has no toJSON method');
      }
      return array.map((v) => type.toJSON(v));
    },

    /**
     * Decodes a JSON-like object into this structure.
     */
    fromJSON(json) {
      if (!('fromJSON' in type)) {
        throw Error(
          'circuitArray.fromJSON: element type has no fromJSON method'
        );
      }
      return json.map((a) => type.fromJSON(a));
    },
    toInput(array) {
      if (!('toInput' in type)) {
        throw Error('circuitArray.toInput: element type has no toInput method');
      }
      return array.reduce(
        (curr, value) => HashInput.append(curr, type.toInput(value)),
        HashInput.empty
      );
    },
   } satisfies ProvableExtended<T[], TJson[]> as any;
  }
```

</file>

<file>

## path: /src/lib/scalar.test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/scalar.test.ts

```
import { shutdown, isReady, Field, Bool, Provable, Scalar } from 'o1js';

describe('scalar', () => {
 beforeAll(async () => {
   await isReady;
 });

 afterAll(async () => {
   setTimeout(async () => {
     await shutdown();
   }, 0);
 });

 describe('scalar', () => {
   describe('Inside circuit', () => {
```

```
  describe('toFields', () => {
    it('should return an array of Fields', () => {
      expect(() => {
        Provable.runAndCheck(() => {
          const x = Provable.witness(Scalar, () => Scalar.random());
          const fieldArr = x.toFields();
          expect(Array.isArray(fieldArr)).toBe(true);
        });
      }).not.toThrow();
    });
  });

  describe('toFields / fromFields', () => {
    it('should return the same', () => {
      expect(() => {
        let s0 = Scalar.random();
        Provable.runAndCheck(() => {
          let s1 = Provable.witness(Scalar, () => s0);
          Provable.assertEqual(Scalar.fromFields(s1.toFields()), s0);
        });
      }).not.toThrow();
    });
  });

  describe('fromBits', () => {
    it('should return a Scalar', () => {
      expect(() => {
        Provable.runAndCheck(() => {
          Provable.witness(Scalar, () =>
            Scalar.fromBits(Field.random().toBits())
          );
        });
      }).not.toThrow();
    });
  });

  describe('random', () => {
    it('two different calls should be different', () => {
      Provable.runAndCheck(() => {
        const x = Provable.witness(Scalar, () => Scalar.random());
        const y = Provable.witness(Scalar, () => Scalar.random());
        expect(x).not.toEqual(y);
      });
    });
  });
});

describe('Outside circuit', () => {
  describe('toFields / fromFields', () => {
    it('roundtrip works', () => {
      let x = Scalar.random();
      expect(Scalar.fromFields(x.toFields())).toEqual(x);
  });
});

describe('fromBits', () => {
  it('should return a shifted scalar', () => {
    let x = Field.random();
    let bits_ = x.toBits();
    let s = Scalar.fromBits(bits_).unshift();
    expect(x.toBigInt()).toEqual(s.toBigInt());
  });
});

describe('random', () => {
  it('two different calls should be different', () => {
    expect(Scalar.random()).not.toEqual(Scalar.random());
  });
});

describe('toJSON/fromJSON', () => {
  it("fromJSON('1') should be 1", () => {
    expect(Scalar.fromJSON('1')!.toJSON()).toEqual('1');
  });

  it("fromJSON('2^32-1') should be 2^32-1", () => {
    expect(Scalar.fromJSON(String(2 ** 32 - 1))!.toJSON()).toEqual(
      String(2 ** 32 - 1)
    );
  });

  it('fromJSON(1) should be 1', () => {
    expect(Scalar.from(1).toJSON()).toEqual('1');
  });

  it('fromJSON(0n) should be 1', () => {
    expect(Scalar.from(0n).toJSON()).toEqual('0');
  });
});

describe('neg', () => {
  it('neg(1)=-1', () => {
    const x = Scalar.from(1);
    expect(x.neg().toBigInt()).toEqual(Scalar.ORDER - 1n);
  });

  it('neg(-1)=1', () => {
    const x = Scalar.from(-1);
    expect(x.neg().toJSON()).toEqual('1');
  });

  it('neg(0)=0', () => {
    const x = Scalar.from(0);
    expect(x.neg().toJSON()).toEqual('0');
  });
});

describe('add', () => {
  it('1+1=2', () => {
    const x = Scalar.from(1);
    const y = Scalar.from(1);
    expect(x.add(y).toJSON()).toEqual('2');
  });

  it('5000+5000=10000', () => {
    const x = Scalar.from(5000);
    const y = Scalar.from(5000);
    expect(x.add(y).toJSON()).toEqual('10000');
  });

  it('((2^64/2)+(2^64/2)) adds to 2^64', () => {
    const v = (1n << 64n) - 2n;
    const x = Scalar.fromJSON(String(v / 2n));
    const y = Scalar.fromJSON(String(v / 2n));
    expect(x.add(y).toJSON()).toEqual(String(v));
  });
});

describe('sub', () => {
  it('1-1=0', () => {
    const x = Scalar.from(1);
    const y = Scalar.from(1);
    expect(x.sub(y).toJSON()).toEqual('0');
  });

  it('10000-5000=5000', () => {
    const x = Scalar.from(10000);
    const y = Scalar.from(5000);
    expect(x.sub(y).toJSON()).toEqual('5000');
  });

  it('0-1=-1', () => {
    const x = Scalar.from(0);
    const y = Scalar.from(1);
    expect(x.sub(y).toBigInt()).toEqual(Scalar.ORDER - 1n);
  });

  it('1-(-1)=2', () => {
    const x = Scalar.from(1);
    const y = Scalar.from(-1);
    expect(x.sub(y).toBigInt()).toEqual(2n);
  });
});

describe('mul', () => {
  it('1x2=2', () => {
      const x = Scalar.from(1);
      const y = Scalar.from(2);
      expect(x.mul(y).toJSON()).toEqual('2');
    });

    it('1x0=0', () => {
      const x = Scalar.from(1);
      const y = Scalar.from(0);
      expect(x.mul(y).toJSON()).toEqual('0');
    });

    it('1000x1000=1000000', () => {
      const x = Scalar.from(1000);
      const y = Scalar.from(1000);
      expect(x.mul(y).toJSON()).toEqual('1000000');
    });

    it('(2^64-1)x1=(2^64-1)', () => {
      const v = (1n << 64n) - 1n;
      const x = Scalar.from(String(v));
      const y = Scalar.from(1);
      expect(x.mul(y).toJSON()).toEqual(String(v));
    });
  });

  describe('div', () => {
    it('2/1=2', () => {
      const x = Scalar.from(2);
      const y = Scalar.from(1);
      expect(x.div(y).toJSON()).toEqual('2');
    });

    it('0/1=0', () => {
      const x = Scalar.from(0);
      const y = Scalar.from(1);
      expect(x.div(y).toJSON()).toEqual('0');
    });

    it('2000/1000=2', () => {
      const x = Scalar.from(2000);
      const y = Scalar.from(1000);
      expect(x.div(y).toJSON()).toEqual('2');
    });

    it('(2^64-1)/1=(2^64-1)', () => {
      const v = (1n << 64n) - 1n;
      const x = Scalar.from(String(v));
      const y = Scalar.from(1);
      expect(x.div(y).toJSON()).toEqual(String(v));
    });
  });
});
```

```
  });
});
```

</file>

<file>

## path: /src/lib/scalar.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/scalar.ts

```
import { Snarky, Provable } from '../snarky.js';
import { Scalar as Fq } from '../provable/curve-bigint.js';
import { Field, FieldConst, FieldVar } from './field.js';
import { MlArray } from './ml/base.js';
import { Bool } from './bool.js';

export { Scalar, ScalarConst, unshift, shift };

// internal API
export { constantScalarToBigint };

type BoolVar = FieldVar;
type ScalarConst = [0, bigint];

const ScalarConst = {
  fromBigint: constFromBigint,
  toBigint: constToBigint,
  is(x: any): x is ScalarConst {
    return Array.isArray(x) && x[0] === 0 && typeof x[1] === 'bigint';
  },
};

let scalarShift = Fq(1n + 2n ** 255n);
let oneHalf = Fq.inverse(2n)!;

type ConstantScalar = Scalar & { constantValue: ScalarConst };

/**
 * Represents a {@link Scalar}.
 */
class Scalar {
  value: MlArray<BoolVar>;
  constantValue?: ScalarConst;

  static ORDER = Fq.modulus;

  private constructor(bits: MlArray<BoolVar>, constantValue?: Fq) {
    this.value = bits;
    constantValue ??= toConstantScalar(bits);
    if (constantValue !== undefined) {
```

```
    this.constantValue = ScalarConst.fromBigint(constantValue);
  }
}

/**
 * Create a constant {@link Scalar} from a bigint, number, string or Scalar.
 *
 * If the input is too large, it is reduced modulo the scalar field size.
 */
static from(x: Scalar | ScalarConst | bigint | number | string) {
  if (x instanceof Scalar) return x;
  if (ScalarConst.is(x)) x = constToBigint(x);
  let scalar = Fq(x);
  let bits = toBits(scalar);
  return new Scalar(bits, scalar);
}

/**
 * Check whether this {@link Scalar} is a hard-coded constant in the constraint system.
 * If a {@link Scalar} is constructed outside provable code, it is a constant.
 */
isConstant(): this is Scalar & { constantValue: ScalarConst } {
  return this.constantValue !== undefined;
}

/**
 * Convert this {@link Scalar} into a constant if it isn't already.
 *
 * If the scalar is a variable, this only works inside &nbsp;asProver&nbsp; or &nbsp;witness&nbsp; blocks.
 *
 * See {@link FieldVar} for an explanation of constants vs. variables.
 */
toConstant(): ConstantScalar {
  if (this.constantValue !== undefined) return this as ConstantScalar;
  let [, ...bits] = this.value;
  let constBits = bits.map((b) => FieldVar.constant(Snarky.field.readVar(b)));
  return new Scalar([0, ...constBits]) as ConstantScalar;
}

/**
 * @deprecated use {@link Scalar.from}
 */
static fromBigInt(x: bigint) {
  return Scalar.from(x);
}

/**
 * Convert this {@link Scalar} into a bigint
 */
toBigInt() {
  return this.#assertConstant('toBigInt');
}
```

```
// TODO: fix this API. we should represent "shifted status" internally and use
// and use shifted Group.scale only if the scalar bits representation is shifted
/**
 * Creates a data structure from an array of serialized {@link Bool}.
 *
 * **Warning**: The bits are interpreted as the bits of 2s + 1 + 2^255, where s is the Scalar.
 */
static fromBits(bits: Bool[]) {
  return Scalar.fromFields(bits.map((b) => b.toField()));
}

/**
 * Returns a random {@link Scalar}.
 * Randomness can not be proven inside a circuit!
 */
static random() {
  return Scalar.from(Fq.random());
}

// operations on constant scalars

#assertConstant(name: string) {
  return constantScalarToBigint(this, &nbsp;Scalar.${name}&nbsp;);
}

/**
 * Negate a scalar field element.
 *
 * **Warning**: This method is not available for provable code.
 */
neg() {
  let x = this.#assertConstant('neg');
  let z = Fq.negate(x);
  return Scalar.from(z);
}

/**
 * Add scalar field elements.
 *
 * **Warning**: This method is not available for provable code.
 */
add(y: Scalar) {
  let x = this.#assertConstant('add');
  let y0 = y.#assertConstant('add');
  let z = Fq.add(x, y0);
  return Scalar.from(z);
}

/**
 * Subtract scalar field elements.
 *
 * **Warning**: This method is not available for provable code.
 */
sub(y: Scalar) {
  let x = this.#assertConstant('sub');
  let y0 = y.#assertConstant('sub');
  let z = Fq.sub(x, y0);
  return Scalar.from(z);
}

/**
 * Multiply scalar field elements.
 *
 * **Warning**: This method is not available for provable code.
 */
mul(y: Scalar) {
  let x = this.#assertConstant('mul');
  let y0 = y.#assertConstant('mul');
  let z = Fq.mul(x, y0);
  return Scalar.from(z);
}

/**
 * Divide scalar field elements.
 * Throws if the denominator is zero.
 *
 * **Warning**: This method is not available for provable code.
 */
div(y: Scalar) {
  let x = this.#assertConstant('div');
  let y0 = y.#assertConstant('div');
  let z = Fq.div(x, y0);
  if (z === undefined) throw Error('Scalar.div(): Division by zero');
  return Scalar.from(z);
}

// TODO don't leak 'shifting' to the user and remove these methods
shift() {
  let x = this.#assertConstant('shift');
  return Scalar.from(shift(x));
}
unshift() {
  let x = this.#assertConstant('unshift');
  return Scalar.from(unshift(x));
}

/**
 * Serialize a Scalar into a Field element plus one bit, where the bit is represented as a Bool.
 *
 * **Warning**: This method is not available for provable code.
 *
 * Note: Since the Scalar field is slightly larger than the base Field, an additional high bit
 * is needed to represent all Scalars. However, for a random Scalar, the high bit will be &nbsp;false&nbsp;
```

```
with overwhelming probability.
  */
 toFieldsCompressed(): { field: Field; highBit: Bool } {
   let s = this.#assertConstant('toFieldsCompressed');
   let lowBitSize = BigInt(Fq.sizeInBits - 1);
   let lowBitMask = (1n << lowBitSize) - 1n;
   return {
     field: new Field(s & lowBitMask),
     highBit: new Bool(s >> lowBitSize === 1n),
   };
 }

 // internal stuff

 // Provable<Scalar>

 /**
  * Part of the {@link Provable} interface.
  *
  * Serialize a {@link Scalar} into an array of {@link Field} elements.
  *
  * **Warning**: This function is for internal usage. It returns 255 field elements
  * which represent the Scalar in a shifted, bitwise format.
  * The fields are not constrained to be boolean.
  */
 static toFields(x: Scalar) {
   let [, ...bits] = x.value;
   return bits.map((b) => new Field(b));
 }

 /**
  * Serialize this Scalar to Field elements.
  *
  * **Warning**: This function is for internal usage. It returns 255 field elements
  * which represent the Scalar in a shifted, bitwise format.
  * The fields are not constrained to be boolean.
  *
  * Check out {@link Scalar.toFieldsCompressed} for a user-friendly serialization
  * that can be used outside proofs.
  */
 toFields(): Field[] {
   return Scalar.toFields(this);
 }

 /**
  * Part of the {@link Provable} interface.
  *
  * Serialize a {@link Scalar} into its auxiliary data, which are empty.
  */
 static toAuxiliary() {
   return [];
 }
```

```
/**
 * Part of the {@link Provable} interface.
 *
 * Creates a data structure from an array of serialized {@link Field} elements.
 */
static fromFields(fields: Field[]): Scalar {
  return new Scalar([0, ...fields.map((x) => x.value)]);
}

/**
 * Part of the {@link Provable} interface.
 *
 * Returns the size of this type in {@link Field} elements.
 */
static sizeInFields(): number {
  return Fq.sizeInBits;
}

/**
 * Part of the {@link Provable} interface.
 *
 * Does nothing.
 */
static check() {
  /* It is not necessary to boolean constrain the bits of a scalar for the following
   reasons:

   The only provable methods which can be called with a scalar value are

   - if
   - assertEqual
   - equal
   - Group.scale

   The only one of these whose behavior depends on the bit values of the input scalars
   is Group.scale, and that function boolean constrains the scalar input itself.
   */
}

// ProvableExtended<Scalar>

/**
 * Serialize a {@link Scalar} to a JSON string.
 * This operation does _not_ affect the circuit and can't be used to prove anything about the string
representation of the Scalar.
 */
static toJSON(x: Scalar) {
  let s = x.#assertConstant('toJSON');
  return s.toString();
}
```

```
  /**
   * Serializes this Scalar to a string
   */
  toJSON() {
    return Scalar.toJSON(this);
  }

  /**
   * Deserialize a JSON structure into a {@link Scalar}.
   * This operation does _not_ affect the circuit and can't be used to prove anything about the string
representation of the Scalar.
   */
  static fromJSON(x: string) {
    return Scalar.from(Fq.fromJSON(x));
  }
}

function toConstantScalar([, ...bits]: MlArray<BoolVar>): Fq | undefined {
  if (bits.length !== Fq.sizeInBits)
    throw Error(
      &nbsp;Scalar: expected bits array of length ${Fq.sizeInBits}, got ${bits.length}&nbsp;
    );
  let constantBits = Array<boolean>(bits.length);
  for (let i = 0; i < bits.length; i++) {
    let bool = bits[i];
    if (!FieldVar.isConstant(bool)) return undefined;
    constantBits[i] = FieldConst.equal(bool[1], FieldConst[1]);
  }
  let sShifted = Fq.fromBits(constantBits);
  return shift(sShifted);
}

function toBits(constantValue: Fq): MlArray<BoolVar> {
  return [
    0,
    ...Fq.toBits(unshift(constantValue)).map((b) =>
      FieldVar.constant(BigInt(b))
    ),
  ];
}

/**
 * s -> 2s + 1 + 2^255
 */
function shift(s: Fq): Fq {
  return Fq.add(Fq.add(s, s), scalarShift);
}

/**
 * inverse of shift, 2s + 1 + 2^255 -> s
 */
function unshift(s: Fq): Fq {
  return Fq.mul(Fq.sub(s, scalarShift), oneHalf);
}

function constToBigint(x: ScalarConst): Fq {
  return x[1];
}
function constFromBigint(x: Fq): ScalarConst {
  return [0, x];
}

function constantScalarToBigint(s: Scalar, name: string) {
  if (s.constantValue === undefined)
    throw Error(
      &nbsp;${name}() is not available in provable code.
That means it can't be called in a @method or similar environment, and there's no alternative implemented
to achieve that.&nbsp;
    );
  return ScalarConst.toBigint(s.constantValue);
}
```

</file>

<file>

## path: /src/lib/signature.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/signature.ts

```
import { Field, Bool, Group, Scalar } from './core.js';
import { prop, CircuitValue, AnyConstructor } from './circuit_value.js';
import { hashWithPrefix } from './hash.js';
import {
  deriveNonce,
  Signature as SignatureBigint,
} from '../mina-signer/src/signature.js';
import { Bool as BoolBigint } from '../provable/field-bigint.js';
import {
  Scalar as ScalarBigint,
  PrivateKey as PrivateKeyBigint,
  PublicKey as PublicKeyBigint,
} from '../provable/curve-bigint.js';
import { prefixes } from '../bindings/crypto/constants.js';
import { constantScalarToBigint } from './scalar.js';
import { toConstantField } from './field.js';

// external API
export { PrivateKey, PublicKey, Signature };

// internal API
export { scaleShifted };
```

```
/**
 * A signing key. You can generate one via {@link PrivateKey.random}.
 */
class PrivateKey extends CircuitValue {
  @prop s: Scalar;

  constructor(s: Scalar) {
    super(s);
  }

  /**
   * You can use this method to generate a private key. You can then obtain
   * the associated public key via {@link toPublicKey}. And generate signatures
   * via {@link Signature.create}.
   *
   * @returns a new {@link PrivateKey}.
   */
  static random(): PrivateKey {
    return new PrivateKey(Scalar.random());
  }

  /**
   * Deserializes a list of bits into a {@link PrivateKey}.
   *
   * @param bs a list of {@link Bool}.
   * @returns a {@link PrivateKey}.
   */
  static fromBits(bs: Bool[]): PrivateKey {
    return new PrivateKey(Scalar.fromBits(bs));
  }

  /**
   * Convert this {@link PrivateKey} to a bigint
   */
  toBigInt() {
    return constantScalarToBigint(this.s, 'PrivateKey.toBigInt');
  }

  /**
   * Create a {@link PrivateKey} from a bigint
   *
   * **Warning**: Private keys should be sampled from secure randomness with sufficient entropy.
   * Be careful that you don't use this method to create private keys that were sampled insecurely.
   */
  static fromBigInt(sk: PrivateKeyBigint) {
    return new PrivateKey(Scalar.from(sk));
  }

  /**
   * Derives the associated public key.
   *
   * @returns a {@link PublicKey}.
   */
  toPublicKey(): PublicKey {
    return PublicKey.fromPrivateKey(this);
  }

  /**
   * Decodes a base58 string into a {@link PrivateKey}.
   *
   * @returns a {@link PrivateKey}.
   */
  static fromBase58(privateKeyBase58: string) {
    let scalar = PrivateKeyBigint.fromBase58(privateKeyBase58);
    return new PrivateKey(Scalar.from(scalar));
  }

  /**
   * Encodes a {@link PrivateKey} into a base58 string.
   * @returns a base58 encoded string
   */
  toBase58() {
    return PrivateKey.toBase58(this);
  }

  // static version, to operate on non-class versions of this type
  /**
   * Static method to encode a {@link PrivateKey} into a base58 string.
   * @returns a base58 encoded string
   */
  static toBase58(privateKey: { s: Scalar }) {
    return PrivateKeyBigint.toBase58(
      constantScalarToBigint(privateKey.s, 'PrivateKey.toBase58')
    );
  }
}

// TODO: this doesn't have a non-default check method yet. does it need one?
/**
 * A public key, which is also an address on the Mina network.
 * You can derive a {@link PublicKey} directly from a {@link PrivateKey}.
 */
class PublicKey extends CircuitValue {
  // compressed representation of a curve point, where &nbsp;isOdd&nbsp; is the least significant bit of
&nbsp;y&nbsp;
  @prop x: Field;
  @prop isOdd: Bool;

  /**
   * Returns the {@link Group} representation of this {@link PublicKey}.
   * @returns A {@link Group}
   */
  toGroup(): Group {
    // compute y from elliptic curve equation y^2 = x^3 + 5
    // TODO: we have to improve constraint efficiency by using range checks
    let { x, isOdd } = this;
    let ySquared = x.mul(x).mul(x).add(5);
    let someY = ySquared.sqrt();
    let isTheRightY = isOdd.equals(someY.toBits()[0]);
    let y = isTheRightY
      .toField()
      .mul(someY)
      .add(isTheRightY.not().toField().mul(someY.neg()));
    return new Group({ x, y });
  }

  /**
   * Creates a {@link PublicKey} from a {@link Group} element.
   * @returns a {@link PublicKey}.
   */
  static fromGroup({ x, y }: Group): PublicKey {
    let isOdd = y.toBits()[0];
    return PublicKey.fromObject({ x, isOdd });
  }

  /**
   * Derives a {@link PublicKey} from a {@link PrivateKey}.
   * @returns a {@link PublicKey}.
   */
  static fromPrivateKey({ s }: PrivateKey): PublicKey {
    return PublicKey.fromGroup(Group.generator.scale(s));
  }

  /**
   * Creates a {@link PublicKey} from a JSON structure element.
   * @returns a {@link PublicKey}.
   */
  static from(g: { x: Field; isOdd: Bool }) {
    return PublicKey.fromObject(g);
  }

  /**
   * Creates an empty {@link PublicKey}.
   * @returns an empty {@link PublicKey}
   */
  static empty() {
    return PublicKey.from({ x: Field(0), isOdd: Bool(false) });
  }

  /**
   * Checks if a {@link PublicKey} is empty.
   * @returns a {@link Bool}
   */
  isEmpty() {
    // there are no curve points with x === 0
    return this.x.isZero();
  }

  /**
   * Decodes a base58 encoded {@link PublicKey} into a {@link PublicKey}.
   * @returns a {@link PublicKey}
   */
  static fromBase58(publicKeyBase58: string) {
    let { x, isOdd } = PublicKeyBigint.fromBase58(publicKeyBase58);
    return PublicKey.from({ x: Field(x), isOdd: Bool(!!isOdd) });
  }

  /**
   * Encodes a {@link PublicKey} in base58 format.
   * @returns a base58 encoded {@link PublicKey}
   */
  toBase58() {
    return PublicKey.toBase58(this);
  }

  /**
   * Static method to encode a {@link PublicKey} into base58 format.
   * @returns a base58 encoded {@link PublicKey}
   */
  static toBase58({ x, isOdd }: PublicKey) {
    x = toConstantField(x, 'toBase58', 'pk', 'public key');
    return PublicKeyBigint.toBase58({
      x: x.toBigInt(),
      isOdd: BoolBigint(isOdd.toBoolean()),
    });
  }

  /**
   * Serializes a {@link PublicKey} into its JSON representation.
   * @returns a JSON string
   */
  static toJSON(publicKey: PublicKey) {
    return publicKey.toBase58();
  }

  /**
   * Deserializes a JSON string into a {@link PublicKey}.
   * @returns a JSON string
   */
  static fromJSON<T extends AnyConstructor>(this: T, publicKey: string) {
    return PublicKey.fromBase58(publicKey) as InstanceType<T>;
  }
}

/**
 * A Schnorr {@link Signature} over the Pasta Curves.
 */
class Signature extends CircuitValue {
```

```
@prop r: Field;
@prop s: Scalar;

/**
 * Signs a message using a {@link PrivateKey}.
 * @returns a {@link Signature}
 */
static create(privKey: PrivateKey, msg: Field[]): Signature {
  const publicKey = PublicKey.fromPrivateKey(privKey).toGroup();
  const d = privKey.s;
  const kPrime = Scalar.fromBigInt(
    deriveNonce(
      { fields: msg.map((f) => f.toBigInt()) },
      { x: publicKey.x.toBigInt(), y: publicKey.y.toBigInt() },
      BigInt(d.toJSON()),
      'testnet'
    )
  );
  let { x: r, y: ry } = Group.generator.scale(kPrime);
  const k = ry.toBits()[0].toBoolean() ? kPrime.neg() : kPrime;
  let h = hashWithPrefix(
    prefixes.signatureTestnet,
    msg.concat([publicKey.x, publicKey.y, r])
  );
  // TODO: Scalar.fromBits interprets the input as a "shifted scalar"
  // therefore we have to unshift e before using it
  let e = unshift(Scalar.fromBits(h.toBits()));
  const s = e.mul(d).add(k);
  return new Signature(r, s);
}

/**
 * Verifies the {@link Signature} using a message and the corresponding {@link PublicKey}.
 * @returns a {@link Bool}
 */
verify(publicKey: PublicKey, msg: Field[]): Bool {
  const point = publicKey.toGroup();
  let h = hashWithPrefix(
    prefixes.signatureTestnet,
    msg.concat([point.x, point.y, this.r])
  );
  // TODO: Scalar.fromBits interprets the input as a "shifted scalar"
  // therefore we have to use scaleShifted which is very inefficient
  let e = Scalar.fromBits(h.toBits());
  let r = scaleShifted(point, e).neg().add(Group.generator.scale(this.s));
  return Bool.and(r.x.equals(this.r), r.y.toBits()[0].equals(false));
}

/**
 * Decodes a base58 encoded signature into a {@link Signature}.
 */
static fromBase58(signatureBase58: string) {
    let { r, s } = SignatureBigint.fromBase58(signatureBase58);
    return Signature.fromObject({
      r: Field(r),
      s: Scalar.fromJSON(s.toString()),
    });
  }
  /**
   * Encodes a {@link Signature} in base58 format.
   */
  toBase58() {
    let r = this.r.toBigInt();
    let s = BigInt(this.s.toJSON());
    return SignatureBigint.toBase58({ r, s });
  }
}

// performs scalar multiplication s*G assuming that instead of s, we got s' = 2s + 1 + 2^255
// cost: 2x scale by constant, 1x scale by variable
function scaleShifted(point: Group, shiftedScalar: Scalar) {
  let oneHalfGroup = point.scale(Scalar.fromBigInt(oneHalf));
  let shiftGroup = oneHalfGroup.scale(Scalar.fromBigInt(shift));
  return oneHalfGroup.scale(shiftedScalar).sub(shiftGroup);
}
// returns s, assuming that instead of s, we got s' = 2s + 1 + 2^255
// (only works out of snark)
function unshift(shiftedScalar: Scalar) {
  return shiftedScalar
    .sub(Scalar.fromBigInt(shift))
    .mul(Scalar.fromBigInt(oneHalf));
}

let shift = ScalarBigint(1n + 2n ** 255n);
let oneHalf = ScalarBigint.inverse(2n)!;
```

</file>

<file>

## path: /src/lib/state.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/state.ts

```
import { ProvablePure } from '../snarky.js';
import { FlexibleProvablePure } from './circuit_value.js';
import { AccountUpdate, TokenId } from './account_update.js';
import { PublicKey } from './signature.js';
import * as Mina from './mina.js';
import { fetchAccount } from './fetch.js';
import { SmartContract } from './zkapp.js';
import { Account } from './mina/account.js';
import { Provable } from './provable.js';
import { Field } from '../lib/core.js';

// external API
export { State, state, declareState };
// internal API
export { assertStatePrecondition, cleanStatePrecondition };

/**
 * Gettable and settable state that can be checked for equality.
 */
type State<A> = {
  /**
   * Get the current on-chain state.
   *
   * Caution: If you use this method alone inside a smart contract, it does not prove that your contract uses
the current on-chain state.
   * To successfully prove that your contract uses the current on-chain state, you must add an additional
&nbsp;.assertEquals()&nbsp; statement or use &nbsp;.getAndAssertEquals()&nbsp;:
   *
   * &nbsp;&nbsp;&nbsp;ts
   * let x = this.x.get();
   * this.x.assertEquals(x);
   * &nbsp;&nbsp;&nbsp;
   *
   * OR
   *
   * &nbsp;&nbsp;&nbsp;ts
   * let x = this.x.getAndAssertEquals();
   * &nbsp;&nbsp;&nbsp;
   */
  get(): A;
  /**
   * Get the current on-chain state and prove it really has to equal the on-chain state,
   * by adding a precondition which the verifying Mina node will check before accepting this transaction.
   */
  getAndAssertEquals(): A;
  /**
   * Set the on-chain state to a new value.
   */
  set(a: A): void;
  /**
   * Asynchronously fetch the on-chain state. This is intended for getting the state outside a smart contract.
   */
  fetch(): Promise<A | undefined>;
  /**
   * Prove that the on-chain state has to equal the given state,
   * by adding a precondition which the verifying Mina node will check before accepting this transaction.
   */
  assertEquals(a: A): void;
  /**
   * **DANGER ZONE**: Override the error message that warns you when you use &nbsp;.get()&nbsp;
without adding a precondition.
```

```
   */
  assertNothing(): void;
  /**
   * Get the state from the raw list of field elements on a zkApp account, for example:
   *
   * &nbsp;&nbsp;&nbsp;ts
   * let myContract = new MyContract(address);
   * let account = Mina.getAccount(address);
   *
   * let x = myContract.x.fromAppState(account.zkapp!.appState);
   * &nbsp;&nbsp;&nbsp;
   */
  fromAppState(appState: Field[]): A;
};
function State<A>(): State<A> {
  return createState<A>();
}

/**
 * A decorator to use within a zkapp to indicate what will be stored on-chain.
 * For example, if you want to store a field element &nbsp;some_state&nbsp; in a zkapp,
 * you can use the following in the declaration of your zkapp:
 *
 * &nbsp;&nbsp;&nbsp;
 * @state(Field) some_state = State<Field>();
 * &nbsp;&nbsp;&nbsp;
 *
 */
function state<A>(stateType: FlexibleProvablePure<A>) {
  return function (
    target: SmartContract & { constructor: any },
    key: string,
    _descriptor?: PropertyDescriptor
  ) {
    const ZkappClass = target.constructor;
    if (reservedPropNames.has(key)) {
      throw Error(&nbsp;Property name ${key} is reserved.&nbsp;);
    }
    let sc = smartContracts.get(ZkappClass);
    if (sc === undefined) {
      sc = { states: [], layout: undefined };
      smartContracts.set(ZkappClass, sc);
    }
    sc.states.push([key, stateType]);

    Object.defineProperty(target, key, {
      get(this) {
        return this._?.[key];
      },
      set(this, v: InternalStateType<A>) {
        if (v._contract !== undefined)
          throw Error(
            'A State should only be assigned once to a SmartContract'
          );
        if (this._?.[key]) throw Error('A @state should only be assigned once');
        v._contract = {
          key,
          stateType: stateType as ProvablePure<A>,
          instance: this,
          class: ZkappClass,
          wasConstrained: false,
          wasRead: false,
          cachedVariable: undefined,
        };
        (this._ ??= {})[key] = v;
      },
    });
  };
}

/**
 * &nbsp;declareState&nbsp; can be used in place of the &nbsp;@state&nbsp; decorator to declare on-chain
state on a SmartContract.
 * It should be placed _after_ the class declaration.
 * Here is an example of declaring a state property &nbsp;x&nbsp; of type &nbsp;Field&nbsp;.
 * &nbsp;&nbsp;&nbsp;ts
 * class MyContract extends SmartContract {
 *   x = State<Field>();
 *   // ...
 * }
 * declareState(MyContract, { x: Field });
 * &nbsp;&nbsp;&nbsp;
 *
 * If you're using pure JS, it's _not_ possible to use the built-in class field syntax,
 * i.e. the following will _not_ work:
 *
 * &nbsp;&nbsp;&nbsp;js
 * // THIS IS WRONG IN JS!
 * class MyContract extends SmartContract {
 *   x = State();
 * }
 * declareState(MyContract, { x: Field });
 * &nbsp;&nbsp;&nbsp;
 *
 * Instead, add a constructor where you assign the property:
 * &nbsp;&nbsp;&nbsp;js
 * class MyContract extends SmartContract {
 *   constructor(x) {
 *     super();
 *     this.x = State();
 *   }
 * }
 * declareState(MyContract, { x: Field });
 * &nbsp;&nbsp;&nbsp;
 */
function declareState<T extends typeof SmartContract>(
  SmartContract: T,
  states: Record<string, FlexibleProvablePure<unknown>>
) {
  for (let key in states) {
    let CircuitValue = states[key];
    state(CircuitValue)(SmartContract.prototype, key);
  }
}

// metadata defined by @state, which link state to a particular SmartContract
type StateAttachedContract<A> = {
  key: string;
  stateType: ProvablePure<A>;
  instance: SmartContract;
  class: typeof SmartContract;
  wasRead: boolean;
  wasConstrained: boolean;
  cachedVariable?: A;
};

type InternalStateType<A> = State<A> & { _contract?: StateAttachedContract<A> };

function createState<T>(): InternalStateType<T> {
  return {
    _contract: undefined as StateAttachedContract<T> | undefined,

    set(state: T) {
      if (this._contract === undefined)
        throw Error(
          'set can only be called when the State is assigned to a SmartContract @state.'
        );
      let layout = getLayoutPosition(this._contract);
      let stateAsFields = this._contract.stateType.toFields(state);
      let accountUpdate = this._contract.instance.self;
      stateAsFields.forEach((x, i) => {
        AccountUpdate.setValue(
          accountUpdate.body.update.appState[layout.offset + i],
          x
        );
      });
    },

    assertEquals(state: T) {
      if (this._contract === undefined)
        throw Error(
          'assertEquals can only be called when the State is assigned to a SmartContract @state.'
        );
      let layout = getLayoutPosition(this._contract);
      let stateAsFields = this._contract.stateType.toFields(state);
      let accountUpdate = this._contract.instance.self;
      stateAsFields.forEach((x, i) => {
        AccountUpdate.assertEquals(
          accountUpdate.body.preconditions.account.state[layout.offset + i],
          x
        );
      });
      this._contract.wasConstrained = true;
    },

    assertNothing() {
      if (this._contract === undefined)
        throw Error(
          'assertNothing can only be called when the State is assigned to a SmartContract @state.'
        );
      this._contract.wasConstrained = true;
    },

    get() {
      if (this._contract === undefined)
        throw Error(
          'get can only be called when the State is assigned to a SmartContract @state.'
        );
      // inside the circuit, we have to cache variables, so there's only one unique variable per on-chain state.
      // if we'd return a fresh variable everytime, developers could easily end up linking just *one* of them to
the precondition,
      // while using an unconstrained variable elsewhere, which would create a loophole in the proof.
      if (
        this._contract.cachedVariable !== undefined &&
        // &nbsp;inCheckedComputation() === true&nbsp; here always implies being inside a wrapped smart
contract method,
        // which will ensure that the cache is cleaned up before & after each method run.
        Provable.inCheckedComputation()
      ) {
        this._contract.wasRead = true;
        return this._contract.cachedVariable;
      }
      let layout = getLayoutPosition(this._contract);
      let contract = this._contract;
      let inProver_ = Provable.inProver();
      let stateFieldsType = Provable.Array(Field, layout.length);
      let stateAsFields = Provable.witness(stateFieldsType, () => {
        let account: Account;
        try {
          account = Mina.getAccount(
            contract.instance.address,
            contract.instance.self.body.tokenId
          );
        } catch (err: any) {
          // TODO: there should also be a reasonable error here
          if (inProver_) {
            throw err;
          }
          let message =
            &nbsp;${contract.key}.get() failed, either:\n&nbsp; +
            &nbsp;1. We can't find this zkapp account in the ledger\n&nbsp; +
            &nbsp;2. Because the zkapp account was not found in the cache. &nbsp; +
            &nbsp;Try calling \&nbsp;await fetchAccount(zkappAddress)\&nbsp; first.\n&nbsp; +
            &nbsp;If none of these are the case, then please reach out on Discord at #zkapp-developers and/or
open an issue to tell us!&nbsp;;
          if (err.message) {
            err.message = message + &nbsp;\n\n${err.message}&nbsp;;
            throw err;
          } else {
            throw Error(message);
          }
        }
        if (account.zkapp?.appState === undefined) {
          // if the account is not a zkapp account, let the default state be all zeroes
          return Array(layout.length).fill(Field(0));
        } else {
          let stateAsFields: Field[] = [];
          for (let i = 0; i < layout.length; ++i) {
            stateAsFields.push(account.zkapp.appState[layout.offset + i]);
          }
          return stateAsFields;
        }
      });

      let state = this._contract.stateType.fromFields(stateAsFields);
      if (Provable.inCheckedComputation())
        this._contract.stateType.check?.(state);
      this._contract.wasRead = true;
      this._contract.cachedVariable = state;
      return state;
    },

    getAndAssertEquals() {
      let state = this.get();
      this.assertEquals(state);
      return state;
    },

    async fetch() {
      if (this._contract === undefined)
        throw Error(
          'fetch can only be called when the State is assigned to a SmartContract @state.'
        );
      if (Mina.currentTransaction.has())
        throw Error(
          'fetch is not intended to be called inside a transaction block.'
        );
      let layout = getLayoutPosition(this._contract);
      let address: PublicKey = this._contract.instance.address;
      let { account } = await fetchAccount({
        publicKey: address,
        tokenId: TokenId.toBase58(TokenId.default),
      });
      if (account === undefined) return undefined;
      let stateAsFields: Field[];
      if (account.zkapp?.appState === undefined) {
        stateAsFields = Array(layout.length).fill(Field(0));
      } else {
        stateAsFields = [];
        for (let i = 0; i < layout.length; i++) {
          stateAsFields.push(account.zkapp.appState[layout.offset + i]);
        }
      }
      return this._contract.stateType.fromFields(stateAsFields);
    },

    fromAppState(appState: Field[]) {
      if (this._contract === undefined)
        throw Error(
          'fromAppState() can only be called when the State is assigned to a SmartContract @state.'
        );
      let layout = getLayoutPosition(this._contract);
      let stateAsFields: Field[] = [];
      for (let i = 0; i < layout.length; ++i) {
        stateAsFields.push(appState[layout.offset + i]);
      }
      return this._contract.stateType.fromFields(stateAsFields);
    },
  };
}

function getLayoutPosition<A>({
  key,
  class: contractClass,
}: StateAttachedContract<A>) {
  let layout = getLayout(contractClass);
  let stateLayout = layout.get(key);
  if (stateLayout === undefined) {
    throw new Error(&nbsp;state ${key} not found&nbsp;);
  }
  return stateLayout;
}

function getLayout(scClass: typeof SmartContract) {
  let sc = smartContracts.get(scClass);
  if (sc === undefined) throw Error('bug');
  if (sc.layout === undefined) {
    let layout = new Map();
    sc.layout = layout;
    let offset = 0;
    sc.states.forEach(([key, stateType]) => {
      let length = stateType.sizeInFields();
    layout.set(key, { offset, length });
    offset += length;
  });
 }
 return sc.layout;
}

// per-smart contract class context for keeping track of state layout
const smartContracts = new WeakMap<
 typeof SmartContract,
 {
  states: [string, ProvablePure<any>][];
  layout: Map<string, { offset: number; length: number }> | undefined;
 }
>();

const reservedPropNames = new Set(['_methods', '_']);

function assertStatePrecondition(sc: SmartContract) {
 try {
  for (let [key, context] of getStateContexts(sc)) {
    // check if every state that was read was also contrained
    if (!context?.wasRead || context.wasConstrained) continue;
    // we accessed a precondition field but not constrained it explicitly - throw an error
    let errorMessage = &nbsp;You used \&nbsp;this.${key}.get()\&nbsp; without adding a precondition that
links it to the actual on-chain state.
Consider adding this line to your code:
this.${key}.assertEquals(this.${key}.get());&nbsp;;
    throw Error(errorMessage);
  }
 } finally {
  cleanStatePrecondition(sc);
 }
}

function cleanStatePrecondition(sc: SmartContract) {
 for (let [, context] of getStateContexts(sc)) {
  if (context === undefined) continue;
  context.wasRead = false;
  context.wasConstrained = false;
  context.cachedVariable = undefined;
 }
}

function getStateContexts(
 sc: SmartContract
): [string, StateAttachedContract<unknown> | undefined][] {
 let scClass = sc.constructor as typeof SmartContract;
 let scInfo = smartContracts.get(scClass);
 if (scInfo === undefined) return [];
 return scInfo.states.map(([key]) => [key, (sc as any)[key]?._contract]);
}
```

</file>

<file>

## path: /src/lib/string.test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/string.test.ts

```
import {
  Bool,
  Character,
  Provable,
  CircuitString,
  Field,
  shutdown,
  isReady,
} from 'o1js';

describe('Circuit String', () => {
  beforeEach(() => isReady);
  afterAll(() => setTimeout(shutdown, 0));

  describe('#equals', () => {
    test('returns true when values are equal', () => {
      const str = CircuitString.fromString(
        'Everything we hear is an opinion, not a fact. Everything we see is a perspective, not the truth'
      );
      const same_str = CircuitString.fromString(
        'Everything we hear is an opinion, not a fact. Everything we see is a perspective, not the truth'
      );
      expect(str.equals(same_str)).toEqual(Bool(true));

      Provable.runAndCheck(() => {
        const str = CircuitString.fromString(
          'Everything we hear is an opinion, not a fact. Everything we see is a perspective, not the truth'
        );
        const same_str = CircuitString.fromString(
          'Everything we hear is an opinion, not a fact. Everything we see is a perspective, not the truth'
        );
        expect(str.equals(same_str)).toEqual(Bool(true));
      });
    });

    test('returns false when values are not equal', () => {
      const str = CircuitString.fromString('Your size');
      const not_same_str = CircuitString.fromString('size');
      expect(str.equals(not_same_str)).toEqual(Bool(false));

      Provable.runAndCheck(() => {
        const str = Provable.witness(CircuitString, () => {
```

```
      return CircuitString.fromString('Your size');
    });
    const not_same_str = Provable.witness(CircuitString, () => {
      return CircuitString.fromString('size');
    });
    Provable.asProver(() => {
      expect(str.equals(not_same_str).toBoolean()).toEqual(false);
    });
  });
 });
});

/*   describe('#contains', () => {
 test('returns true when str contains other str', () => {
   const str = CircuitString.fromString(
     'Everything we hear is an opinion, not a fact. Everything we see is a perspective, not the truth'
   );
   const contained_str = CircuitString.fromString(
     'Everything we hear is an opinion, not a fact.'
   );
   expect(str.contains(contained_str)).toEqual(new Bool(true));

   Provable.runAndCheck(() => {
     const str = CircuitString.fromString(
       'Everything we hear is an opinion, not a fact. Everything we see is a perspective, not the truth'
     );
     const contained_str = CircuitString.fromString(
       'Everything we hear is an opinion, not a fact.'
     );
     expect(str.contains(contained_str)).toEqual(new Bool(true));
   });
 });

 test('returns false when str does not contain other str', () => {
   const str = CircuitString.fromString('abcdefghijklmnop');
   const not_contained_str = CircuitString.fromString('defhij');
   expect(str.contains(not_contained_str)).toEqual(new Bool(false));

   Provable.runAndCheck(() => {
     const str = CircuitString.fromString('abcdefghijklmnop');
     const not_contained_str = CircuitString.fromString('defhij');
     expect(str.contains(not_contained_str)).toEqual(new Bool(false));
   });
 });

 describe('compatibility with implementing classes', () => {
   test('string8 may contain string', () => {
     const str = CircuitString8.fromString('abcd');
     const contained_str = CircuitString.fromString('ab');
     expect(str.contains(contained_str)).toEqual(new Bool(true));

     Provable.runAndCheck(() => {
        const str = CircuitString8.fromString('abcd');
        const contained_str = CircuitString.fromString('ab');
        expect(str.contains(contained_str)).toEqual(new Bool(true));
      });
    });

    test('string may contain string8', () => {
      const str = CircuitString.fromString('abcd');
      const contained_str = CircuitString8.fromString('ab');
      expect(str.contains(contained_str)).toEqual(new Bool(true));

      Provable.runAndCheck(() => {
        const str = CircuitString.fromString('abcd');
        const contained_str = CircuitString8.fromString('ab');
        expect(str.contains(contained_str)).toEqual(new Bool(true));
      });
    });
  });
}); */

describe('#toString', () => {
  test('serializes to string', () => {
    const js_str =
      'Everything we hear is an opinion, not a fact. Everything we see is a perspective, not the truth';
    const str = CircuitString.fromString(js_str);
    expect(str.toString()).toBe(js_str);

    Provable.runAndCheck(() => {
      const js_str =
        'Everything we hear is an opinion, not a fact. Everything we see is a perspective, not the truth';
      const str = CircuitString.fromString(js_str);
      expect(str.toString()).toBe(js_str);
    });
  });
});

describe('#substring', () => {
  test('selects substring', () => {
    const str = CircuitString.fromString(
      'Everything we hear is an opinion, not a fact. Everything we see is a perspective, not the truth'
    );
    expect(str.substring(46, 80).toString()).toBe(
      'Everything we see is a perspective'
    );

    Provable.runAndCheck(() => {
      const str = CircuitString.fromString(
        'Everything we hear is an opinion, not a fact. Everything we see is a perspective, not the truth'
      );
      expect(str.substring(46, 80).toString()).toBe(
        'Everything we see is a perspective'
      );
    });
  });
 });

 describe('#append', () => {
  test('appends 2 strings', () => {
    const str1 = CircuitString.fromString('abcd');
    const str2 = CircuitString.fromString('efgh');
    expect(str1.append(str2).toString()).toBe('abcdefgh');

    Provable.runAndCheck(() => {
      const str1 = CircuitString.fromString('abcd');
      const str2 = CircuitString.fromString('efgh');
      expect(str1.append(str2).toString()).toBe('abcdefgh');
    });
  });
 });

 /*   describe('CircuitString8', () => {
  test('cannot create more than 8 chars', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        Provable.witness(CircuitString8, () => {
          return CircuitString8.fromString('More than eight chars');
        });
      });
    }).toThrow();
  });
 }); */

 describe('with invalid input', () => {
  test.skip('cannot use a character out of range', () => {
    expect(() => {
      Provable.runAndCheck(() => {
        const str = Provable.witness(CircuitString, () => {
          return CircuitString.fromCharacters([
            new Character(Field(100)),
            new Character(Field(10000)),
            new Character(Field(100)),
          ]);
        });
      });
    }).toThrow();
  });
 });
});
```

</file>

<file>

path: /src/lib/string.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/string.ts

```
import { Bool, Field } from '../lib/core.js';
import { arrayProp, CircuitValue, prop } from './circuit_value.js';
import { Provable } from './provable.js';
import { Poseidon } from './hash.js';

export { Character, CircuitString };

const DEFAULT_STRING_LENGTH = 128;

class Character extends CircuitValue {
  @prop value: Field;

  isNull(): Bool {
    return this.equals(NullCharacter() as this);
  }

  toField(): Field {
    return this.value;
  }

  toString(): string {
    const charCode = Number(this.value.toString());
    return String.fromCharCode(charCode);
  }

  static fromString(str: string) {
    const char = Field(str.charCodeAt(0));
    return new Character(char);
  }

  // TODO: Add support for more character sets
  // right now it's 16 bits because 8 not supported :/
  static check(c: Character) {
    c.value.rangeCheckHelper(16).assertEquals(c.value);
  }
}

class CircuitString extends CircuitValue {
  static maxLength = DEFAULT_STRING_LENGTH;
  @arrayProp(Character, DEFAULT_STRING_LENGTH) values: Character[];

  // constructor is private because
  // * we do not want extra logic inside CircuitValue constructors, as a general pattern (to be able to create
them generically)
  // * here, not running extra logic to fill up the characters would be wrong
  private constructor(values: Character[]) {
    super(values);
```

```
}
// this is the publicly accessible constructor
static fromCharacters(chars: Character[]): CircuitString {
  return new CircuitString(fillWithNull(chars, this.maxLength));
}

private maxLength() {
  return (this.constructor as typeof CircuitString).maxLength;
}

// some O(n) computation that should be only done once in the circuit
private computeLengthAndMask() {
  let n = this.values.length;
  // length is the actual, dynamic length
  let length = Field(0);
  // mask is an array that is true where &nbsp;this&nbsp; has its first null character, false elsewhere
  let mask = [];
  let wasntNullAlready = Bool(true);
  for (let i = 0; i < n; i++) {
    let isNull = this.values[i].isNull();
    mask[i] = isNull.and(wasntNullAlready);
    wasntNullAlready = isNull.not().and(wasntNullAlready);
    length.add(wasntNullAlready.toField());
  }
  // mask has length n+1, the last element is true when &nbsp;this&nbsp; has no null char
  mask[n] = wasntNullAlready;
  (this as any)._length = length;
  (this as any)._mask = mask;
  return { mask, length };
}
private lengthMask(): Bool[] {
  return (this as any)._mask ?? this.computeLengthAndMask().mask;
}
private length(): Field {
  return (this as any)._length ?? this.computeLengthAndMask().length;
}

/**
 * appends another string to this one, returns the result and proves that it fits
 * within the &nbsp;maxLength&nbsp; of this string (the other string can have a different maxLength)
 */
append(str: CircuitString): CircuitString {
  let n = this.maxLength();
  // only allow append if the dynamic length does not overflow
  this.length().add(str.length()).assertLessThan(n);

  let chars = this.values;
  let otherChars = fillWithNull(str.values, n);

  // compute the concatenated string -- for *each* of the possible lengths of the first string
  let possibleResults = [];
  for (let length = 0; length < n + 1; length++) {
    // if the first string has this &nbsp;length&nbsp;, then this is the result:
    possibleResults[length] = chars
      .slice(0, length)
      .concat(otherChars.slice(0, n - length));
  }
  // compute the actual result, by always picking the char which correponds to the actual length
  let result: Character[] = [];
  let mask = this.lengthMask();
  for (let i = 0; i < n; i++) {
    let possibleCharsAtI = possibleResults.map((r) => r[i]);
    result[i] = Provable.switch(mask, Character, possibleCharsAtI);
  }
  return CircuitString.fromCharacters(result);
 }

 // TODO
 /**
  * returns true if &nbsp;str&nbsp; is found in this &nbsp;CircuitString&nbsp;
  */
 // contains(str: CircuitString): Bool {
 //   // only succeed if the dynamic length is smaller
 //   let otherLength = str.length();
 //   otherLength.assertLessThan(this.length());
 // }

 hash(): Field {
   return Poseidon.hash(this.values.map((x) => x.value));
 }

 substring(start: number, end: number): CircuitString {
   return CircuitString.fromCharacters(this.values.slice(start, end));
 }

 toString(): string {
   return this.values
     .map((x) => x.toString())
     .join('')
     .replace(/[^ -~]+/g, '');
 }

 static fromString(str: string): CircuitString {
   if (str.length > this.maxLength) {
     throw Error('CircuitString.fromString: input string exceeds max length!');
   }
   let characters = str.split('').map((x) => Character.fromString(x));
   return CircuitString.fromCharacters(characters);
 }
}

// TODO
// class CircuitString8 extends CircuitString {
//   static maxLength = 8;
//   @arrayProp(Character, 8) values: Character[] = [];
// }

// note: this used to be a custom class, which doesn't work
// NullCharacter must use the same circuits as normal Characters
let NullCharacter = () => new Character(Field(0));

function fillWithNull([...values]: Character[], length: number) {
  let nullChar = NullCharacter();
  for (let i = values.length; i < length; i++) {
    values[i] = nullChar;
  }
  return values;
}
```

</file>

<file>

## path: /src/lib/testing/equivalent.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/testing/equivalent.ts

```
/**
 * helpers for testing equivalence of two implementations, one of them on bigints
 */
import { test, Random } from '../testing/property.js';
import { Provable } from '../provable.js';
import { deepEqual } from 'node:assert/strict';
import { Bool, Field } from '../core.js';

export {
  equivalent,
  equivalentProvable,
  equivalentAsync,
  oneOf,
  throwError,
  handleErrors,
  deepEqual as defaultAssertEqual,
  id,
  fieldWithRng,
};
export { field, bigintField, bool, boolean, unit };
export { Spec, ToSpec, FromSpec, SpecFromFunctions, ProvableSpec };

// a &nbsp;Spec&nbsp; tells us how to compare two functions

type FromSpec<In1, In2> = {
  // &nbsp;rng&nbsp; creates random inputs to the first function
  rng: Random<In1>;
  // &nbsp;there&nbsp; converts to inputs to the second function
  there: (x: In1) => In2;

  // &nbsp;provable&nbsp; tells us how to create witnesses, to test provable code
  // note: we only allow the second function to be provable;
  // the second because it's more natural to have non-provable types as random generator output
  provable?: Provable<In2>;
};

type ToSpec<Out1, Out2> = {
  // &nbsp;back&nbsp; converts outputs of the second function back to match the first function
  back: (x: Out2) => Out1;

  // &nbsp;assertEqual&nbsp; to compare outputs against each other; defaults to &nbsp;deepEqual&nbsp;
  assertEqual?: (x: Out1, y: Out1, message: string) => void;
};

type Spec<T1, T2> = FromSpec<T1, T2> & ToSpec<T1, T2>;

type ProvableSpec<T1, T2> = Spec<T1, T2> & { provable: Provable<T2> };

type FuncSpec<In1 extends Tuple<any>, Out1, In2 extends Tuple<any>, Out2> = {
  from: {
    [k in keyof In1]: k extends keyof In2 ? FromSpec<In1[k], In2[k]> : never;
  };
  to: ToSpec<Out1, Out2>;
};

type SpecFromFunctions<
  F1 extends AnyFunction,
  F2 extends AnyFunction
> = FuncSpec<Parameters<F1>, ReturnType<F1>, Parameters<F2>, ReturnType<F2>>;

function id<T>(x: T) {
  return x;
}

// unions of specs, to cleanly model function parameters that are unions of types

type FromSpecUnion<T1, T2> = {
  _isUnion: true;
  specs: Tuple<FromSpec<T1, T2>>;
  rng: Random<[number, T1]>;
};

type OrUnion<T1, T2> = FromSpec<T1, T2> | FromSpecUnion<T1, T2>;

type Union<T> = T[keyof T & number];

function oneOf<In extends Tuple<FromSpec<any, any>>>(
  ...specs: In
): FromSpecUnion<Union<Params1<In>>, Union<Params2<In>>> {
  // the randomly generated value from a union keeps track of which spec it came from
  let rng = Random.oneOf(
    ...specs.map((spec, i) =>
      Random.map(spec.rng, (x) => [i, x] as [number, any])
    )
  );
  return { _isUnion: true, specs, rng };
}

function toUnion<T1, T2>(spec: OrUnion<T1, T2>): FromSpecUnion<T1, T2> {
  let specAny = spec as any;
  return specAny._isUnion ? specAny : oneOf(specAny);
}

// equivalence tester

function equivalent<
  In extends Tuple<FromSpec<any, any>>,
  Out extends ToSpec<any, any>
>({ from, to }: { from: In; to: Out }) {
  return function run(
    f1: (...args: Params1<In>) => Result1<Out>,
    f2: (...args: Params2<In>) => Result2<Out>,
    label = 'expect equal results'
  ) {
    let generators = from.map((spec) => spec.rng);
    let assertEqual = to.assertEqual ?? deepEqual;
    test(...(generators as any[]), (...args) => {
      args.pop();
      let inputs = args as Params1<In>;
      handleErrors(
        () => f1(...inputs),
        () =>
          to.back(
            f2(...(inputs.map((x, i) => from[i].there(x)) as Params2<In>))
          ),
        (x, y) => assertEqual(x, y, label),
        label
      );
    });
  };
}

// async equivalence

function equivalentAsync<
  In extends Tuple<FromSpec<any, any>>,
  Out extends ToSpec<any, any>
>({ from, to }: { from: In; to: Out }, { runs = 1 } = {}) {
  return async function run(
    f1: (...args: Params1<In>) => Promise<Result1<Out>> | Result1<Out>,
    f2: (...args: Params2<In>) => Promise<Result2<Out>> | Result2<Out>,
    label = 'expect equal results'
  ) {
    let generators = from.map((spec) => spec.rng);
    let assertEqual = to.assertEqual ?? deepEqual;

    let nexts = generators.map((g) => g.create());

    for (let i = 0; i < runs; i++) {
      let args = nexts.map((next) => next());
      let inputs = args as Params1<In>;
      try {
        await handleErrorsAsync(
          () => f1(...inputs),
          async () =>
            to.back(
              await f2(
                ...(inputs.map((x, i) => from[i].there(x)) as Params2<In>)
              )
            ),
          (x, y) => assertEqual(x, y, label),
          label
        );
      } catch (err) {
        console.log(...inputs);
        throw err;
      }
    }
  };
}

// equivalence tester for provable code

function equivalentProvable<
  In extends Tuple<OrUnion<any, any>>,
  Out extends ToSpec<any, any>
>({ from: fromRaw, to }: { from: In; to: Out }) {
  let fromUnions = fromRaw.map(toUnion);
  return function run(
    f1: (...args: Params1<In>) => Result1<Out>,
    f2: (...args: Params2<In>) => Result2<Out>,
    label = 'expect equal results'
  ) {
    let generators = fromUnions.map((spec) => spec.rng);
    let assertEqual = to.assertEqual ?? deepEqual;
    test(...generators, (...args) => {
      args.pop();

      // figure out which spec to use for each argument
      let from = (args as [number, unknown][]).map(
        ([j], i) => fromUnions[i].specs[j]
      );
      let inputs = (args as [number, unknown][]).map(
        ([, x]) => x
      ) as Params1<In>;
      let inputs2 = inputs.map((x, i) => from[i].there(x)) as Params2<In>;

      // outside provable code
      handleErrors(
        () => f1(...inputs),
        () => f2(...inputs2),
        (x, y) => assertEqual(x, to.back(y), label),
        label
      );

      // inside provable code
      Provable.runAndCheck(() => {
        let inputWitnesses = inputs2.map((x, i) => {
          let provable = from[i].provable;
          return provable !== undefined
            ? Provable.witness(provable, () => x)
            : x;
        }) as Params2<In>;
        handleErrors(
          () => f1(...inputs),
          () => f2(...inputWitnesses),
          (x, y) => Provable.asProver(() => assertEqual(x, to.back(y), label))
        );
      });
    });
  };
}

// some useful specs

let unit: ToSpec<void, void> = { back: id, assertEqual() {} };

let field: ProvableSpec<bigint, Field> = {
  rng: Random.field,
  there: Field,
  back: (x) => x.toBigInt(),
  provable: Field,
};

let bigintField: Spec<bigint, bigint> = {
  rng: Random.field,
  there: id,
  back: id,
};

let bool: ProvableSpec<boolean, Bool> = {
  rng: Random.boolean,
  there: Bool,
  back: (x) => x.toBoolean(),
  provable: Bool,
};
let boolean: Spec<boolean, boolean> = {
  rng: Random.boolean,
  there: id,
  back: id,
};

function fieldWithRng(rng: Random<bigint>): Spec<bigint, Field> {
  return { ...field, rng };
}

// helper to ensure two functions throw equivalent errors

function handleErrors<T, S, R>(
  op1: () => T,
  op2: () => S,
  useResults?: (a: T, b: S) => R,
  label?: string
): R | undefined {
  let result1: T, result2: S;
  let error1: Error | undefined;
  let error2: Error | undefined;
  try {
    result1 = op1();
  } catch (err) {
    error1 = err as Error;
  }
  try {
    result2 = op2();
  } catch (err) {
    error2 = err as Error;
  }
  if (!!error1 !== !!error2) {
    error1 && console.log(error1);
    error2 && console.log(error2);
  }
  let message = &nbsp;${(label && &nbsp;${label}: &nbsp;) || ''}equivalent errors&nbsp;;
  deepEqual(!!error1, !!error2, message);
  if (!(error1 || error2) && useResults !== undefined) {
    return useResults(result1!, result2!);
  }
}

async function handleErrorsAsync<T, S, R>(
  op1: () => T,
  op2: () => S,
  useResults?: (a: Awaited<T>, b: Awaited<S>) => R,
  label?: string
): Promise<R | undefined> {
  let result1: Awaited<T>, result2: Awaited<S>;
  let error1: Error | undefined;
  let error2: Error | undefined;
  try {
    result1 = await op1();
  } catch (err) {
    error1 = err as Error;
  }
  try {
    result2 = await op2();
  } catch (err) {
    error2 = err as Error;
  }
  if (!!error1 !== !!error2) {
    error1 && console.log(error1);
    error2 && console.log(error2);
  }
  let message = &nbsp;${(label && &nbsp;${label}: &nbsp;) || ''}equivalent errors&nbsp;;
  deepEqual(!!error1, !!error2, message);
  if (!(error1 || error2) && useResults !== undefined) {
    return useResults(result1!, result2!);
  }
}

function throwError(message?: string): any {
  throw Error(message);
}

// helper types

type AnyFunction = (...args: any) => any;

type Tuple<T> = [] | [T, ...T[]];

// infer input types from specs

type Param1<In extends OrUnion<any, any>> = In extends {
  there: (x: infer In) => any;
}
  ? In
  : In extends FromSpecUnion<infer T1, any>
  ? T1
  : never;
type Param2<In extends OrUnion<any, any>> = In extends {
  there: (x: any) => infer In;
}
  ? In
  : In extends FromSpecUnion<any, infer T2>
  ? T2
  : never;

type Params1<Ins extends Tuple<OrUnion<any, any>>> = {
  [k in keyof Ins]: Param1<Ins[k]>;
};
type Params2<Ins extends Tuple<OrUnion<any, any>>> = {
  [k in keyof Ins]: Param2<Ins[k]>;
};

type Result1<Out extends ToSpec<any, any>> = Out extends ToSpec<infer Out1, any>
  ? Out1
  : never;
type Result2<Out extends ToSpec<any, any>> = Out extends ToSpec<any, infer Out2>
  ? Out2
  : never;
```

</file>

<file>

## path: /src/lib/testing/property.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/testing/property.ts

```
import { Random } from './random.js';
export { test };
export { Random, sample, withHardCoded } from './random.js';

const defaultTimeBudget = 100; // ms
const defaultMinRuns = 15;
const defaultMaxRuns = 400;

const test = Object.assign(testCustom(), {
  negative: testCustom({ negative: true }),
  custom: testCustom,
});

/**
 * Create a customized test runner.
 *
 * The runner takes any number of generators (Random<T>) and a function which gets samples as inputs, and performs the test.
 * The test can be either performed by using the &nbsp;assert&nbsp; function which is passed as argument, or simply throw an error when an assertion fails:
 *
 * &nbsp;&nbsp;&nbsp;ts
 * let test = testCustom();
 *
 * test(Random.nat(5), (x, assert) => {
 *   // x is one sample of the &nbsp;Random.nat(5)&nbsp; distribution
 *   // we can make assertions about it by using &nbsp;assert&nbsp;
 *   assert(x < 6, "should not exceed max value of 5");
 *   // or by using any other assertion library which throws errors on failing assertions:
 *   expect(x).toBeLessThan(6);
 * })
 * &nbsp;&nbsp;&nbsp;
 *
 * Parameters &nbsp;minRuns&nbsp;, &nbsp;maxRuns&nbsp; and &nbsp;timeBudget&nbsp; determine how
often a test is run:
 * - We definitely run the test &nbsp;minRuns&nbsp; times
 * - Then we determine how many more test fit into the &nbsp;timeBudget&nbsp; (time the test should take,
in milliseconds)
 * - And we run the test as often as we can within that budget, but at most &nbsp;maxRuns&nbsp; times.
 *
 * If one run fails, the entire test stops immediately and the failing sample is printed to the console.
 *
 * The parameter &nbsp;negative&nbsp; inverts this behaviour: If &nbsp;negative: true&nbsp;, _every_
sample is expected to fail and the test
 * stops if one sample succeeds.
 *
 * The default behaviour of printing out failing samples can be turned off by setting &nbsp;logFailures:
false&nbsp;.
 */
function testCustom({
  minRuns = defaultMinRuns,
  maxRuns = defaultMaxRuns,
  timeBudget = defaultTimeBudget,
  negative = false,
  logFailures = true,
} = {}) {
  return function <T extends readonly Random<any>[]>(
    ...args: ArrayTestArgs<T>
  ) {
    let run: (...args: ArrayRunArgs<Nexts<T>>) => void;
    let arg = args.pop();
    if (typeof arg !== 'function') {
      if (arg !== undefined) timeBudget = (arg as any).timeBudget;
      run = args.pop() as any;
    } else {
      run = arg;
    }
    let gens = args as any as T;
    let nexts = gens.map((g) => g.create()) as Nexts<T>;
    let start = performance.now();
    // run at least &nbsp;minRuns&nbsp; times
    testN(minRuns, nexts, run, { negative, logFailures });
    let time = performance.now() - start;
    if (time > timeBudget || minRuns >= maxRuns) return minRuns;
    // (minRuns + remainingRuns) * timePerRun = timeBudget
    let remainingRuns = Math.floor(timeBudget / (time / minRuns)) - minRuns;
    // run at most &nbsp;maxRuns&nbsp; times
    if (remainingRuns > maxRuns - minRuns) remainingRuns = maxRuns - minRuns;
    testN(remainingRuns, nexts, run, { negative, logFailures });
    return minRuns + remainingRuns;
  };
}

function testN<T extends readonly (() => any)[]>(
  N: number,
  nexts: T,
  run: (...args: ArrayRunArgs<T>) => void,
  { negative = false, logFailures = true } = {}
) {
  let errorMessages: string[] = [];
  let fail = false;
  let count = 0;
  function assert(ok: boolean, message?: string) {
    count++;
    if (!ok) {
      fail = true;
      errorMessages.push(
        &nbsp;Failed: ${message ? &nbsp;"${message}"&nbsp; : &nbsp;assertion #${count}&nbsp;}&nbsp;
      );
    }
  }
  for (let i = 0; i < N; i++) {
    count = 0;
    fail = false;
    let error: Error | undefined;
    let values = nexts.map((next) => next());
    try {
      (run as any)(...values, assert);
    } catch (e: any) {
      error = e;
      fail = true;
    }
    if (fail) {
      if (negative) continue;
      if (logFailures) {
        console.log('failing inputs:');
        values.forEach((v) => console.dir(v, { depth: Infinity }));
      }
      let message = '\n' + errorMessages.join('\n');
      if (error === undefined) throw Error(message);
      error.message = &nbsp;${message}\nFailed - error during test execution:
${error.message}&nbsp;;
      throw error;
    } else {
      if (!negative) continue;
      if (logFailures) {
        console.log('succeeding inputs:');
        values.forEach((v) => console.dir(v, { depth: Infinity }));
      }
      throw Error('Negative test failed - one run succeeded');
    }
  }
}

// types

type Nexts<T extends readonly Random<any>[]> = {
  [i in keyof T]: T[i]['create'] extends () => () => infer U ? () => U : never;
};

type ArrayTestArgs<T extends readonly Random<any>[]> = [
  ...gens: T,
  run: (...args: ArrayRunArgs<Nexts<T>>) => void
];

type ArrayRunArgs<Nexts extends readonly (() => any)[]> = [
  ...values: { [i in keyof Nexts]: Nexts[i] extends () => infer U ? U : never },
  assert: (b: boolean, message?: string) => void
];
```

</file>

<file>

## path: /src/lib/testing/random.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/testing/random.ts

```
import {
  customTypes,
  Layout,
  TypeMap,
  Json,
  AccountUpdate,
  ZkappCommand,
  emptyValue,
} from '../../bindings/mina-transaction/gen/transaction-bigint.js';
import {
  AuthRequired,
  Bool,
  Events,
  Field,
  Actions,
  ActionState,
  VerificationKeyHash,
  ReceiptChainHash,
  Sign,
  TokenId,
  TokenSymbol,
  ZkappUri,
  PublicKey,
  StateHash,
} from '../../bindings/mina-transaction/transaction-leaves-bigint.js';
import { genericLayoutFold } from '../../bindings/lib/from-layout.js';
import { jsLayout } from '../../bindings/mina-transaction/gen/js-layout.js';
import {
  GenericProvable,
  primitiveTypeMap,
} from '../../bindings/lib/generic.js';
import { Scalar, PrivateKey, Group } from '../../provable/curve-bigint.js';
import { Signature } from '../../mina-signer/src/signature.js';
import { randomBytes } from '../../bindings/crypto/random.js';
import { alphabet } from '../base58.js';
import { bytesToBigInt } from '../../bindings/crypto/bigint-helpers.js';
import { Memo } from '../../mina-signer/src/memo.js';
import { ProvableExtended } from '../../bindings/lib/provable-bigint.js';
import { tokenSymbolLength } from '../../bindings/mina-transaction/derived-leaves.js';
import { stringLengthInBytes } from '../../bindings/lib/binable.js';
import { mocks } from '../../bindings/crypto/constants.js';

export { Random, sample, withHardCoded };

type Random<T> = {
  create(): () => T;
  invalid?: Random<T>;
};
type RandomWithInvalid<T> = Required<Random<T>>;

function Random_<T>(
  next: () => T,
  toInvalid?: (valid: Random<T>) => Random<T>
): Random<T> {
  let rng: Random<T> = { create: () => next };
  if (toInvalid !== undefined) rng.invalid = toInvalid(rng);
  return rng;
}

function sample<T>(rng: Random<T>, size: number) {
  let next = rng.create();
  return Array.from({ length: size }, next);
}

const boolean = Random_(() => drawOneOf8() < 4);

const bool = map(boolean, Bool);
const uint32 = biguintWithInvalid(32);
const uint64 = biguintWithInvalid(64);
const byte = Random_(drawRandomByte);

const field = fieldWithInvalid(Field);
const scalar = fieldWithInvalid(Scalar);

const sign = map(boolean, (b) => Sign(b ? 1 : -1));
const privateKey = Random_(PrivateKey.random);
const publicKey = publicKeyWithInvalid();
const keypair = map(privateKey, (privatekey) => ({
  privatekey,
  publicKey: PrivateKey.toPublicKey(privatekey),
}));
```

```
const tokenId = oneOf(TokenId.emptyValue(), field);
const stateHash = field;
const authRequired = map(
  oneOf<Json.AuthRequired[]>(
    'None',
    'Proof',
    'Signature',
    'Either',
    'Impossible'
  ),
  AuthRequired.fromJSON
);
const tokenSymbolString = reject(
  string(nat(tokenSymbolLength)),
  (s) => stringLengthInBytes(s) > 6
);
const tokenSymbol = map(tokenSymbolString, TokenSymbol.fromJSON);
const events = mapWithInvalid(
  array(array(field, int(1, 5)), nat(2)),
  Events.fromList
);
const actions = mapWithInvalid(
  array(array(field, int(1, 5)), nat(2)),
  Actions.fromList
);
const actionState = oneOf(ActionState.emptyValue(), field);
const verificationKeyHash = oneOf(VerificationKeyHash.emptyValue(), field);
const receiptChainHash = oneOf(ReceiptChainHash.emptyValue(), field);
const zkappUri = map(string(nat(50)), ZkappUri.fromJSON);

const PrimitiveMap = primitiveTypeMap<bigint>();
type Types = typeof TypeMap & typeof customTypes & typeof PrimitiveMap;
type Provable<T> = GenericProvable<T, bigint>;
type Generators = {
  [K in keyof Types]: Types[K] extends Provable<infer U> ? Random<U> : never;
};
const Generators: Generators = {
  Field: field,
  Bool: bool,
  UInt32: uint32,
  UInt64: uint64,
  Sign: sign,
  PublicKey: publicKey,
  TokenId: tokenId,
  StateHash: stateHash,
  AuthRequired: authRequired,
  TokenSymbol: tokenSymbol,
  Events: events,
  Actions: actions,
  ActionState: actionState,
  VerificationKeyHash: verificationKeyHash,
  ReceiptChainHash: receiptChainHash,
  ZkappUri: zkappUri,
  null: constant(null),
  string: base58(nat(50)), // TODO replace various strings, like signature, with parsed types
  number: nat(3),
};
let typeToBigintGenerator = new Map<Provable<any>, Random<any>>(
  [TypeMap, PrimitiveMap, customTypes]
    .map(Object.entries)
    .flat()
    .map(([key, value]) => [value, Generators[key as keyof Generators]])
);

// transaction stuff
const accountUpdate = mapWithInvalid(
  generatorFromLayout<AccountUpdate>(jsLayout.AccountUpdate as any, {
    isJson: false,
  }),
  (a) => {
    // TODO set proof to none since we can't generate a valid random one
    a.authorization.proof = undefined;
    // TODO set signature to null since the deriver encodes it as arbitrary string
    a.authorization.signature = undefined;
    // ensure authorization kind is valid
    let { isProved, isSigned } = a.body.authorizationKind;
    if (isProved && isSigned) {
      a.body.authorizationKind.isProved = Bool(false);
    }
    if (!a.body.authorizationKind.isProved) {
      a.body.authorizationKind.verificationKeyHash = Field(0);
    }
    // ensure mayUseToken is valid
    let { inheritFromParent, parentsOwnToken } = a.body.mayUseToken;
    if (inheritFromParent && parentsOwnToken) {
      a.body.mayUseToken.inheritFromParent = Bool(false);
    }
    return a;
  }
);

const feePayer = generatorFromLayout<ZkappCommand['feePayer']>(
  jsLayout.ZkappCommand.entries.feePayer as any,
  { isJson: false }
);
const memoString = reject(string(nat(32)), (s) => stringLengthInBytes(s) > 32);
const memo = map(memoString, (s) => Memo.toBase58(Memo.fromString(s)));
const signature = record({ r: field, s: scalar });

// invalid json inputs can contain invalid stringified numbers, but also non-numeric strings
const toString = <T>(rng: Random<T>) => map(rng, String);
const nonInteger = map(uint32, fraction(3), (x, frac) => Number(x) + frac);
const nonNumericString = reject(
  string(nat(20)),
  (str: any) => !isNaN(str) && !isNaN(parseFloat(str))
);
const invalidUint64Json = toString(
  oneOf(uint64.invalid, nonInteger, nonNumericString)
);
const invalidUint32Json = toString(
  oneOf(uint32.invalid, nonInteger, nonNumericString)
);

// some json versions of those types
let json_ = {
  uint64: { ...toString(uint64), invalid: invalidUint64Json },
  uint32: { ...toString(uint32), invalid: invalidUint32Json },
  publicKey: withInvalidBase58(mapWithInvalid(publicKey, PublicKey.toBase58)),
  privateKey: withInvalidBase58(map(privateKey, PrivateKey.toBase58)),
  keypair: map(keypair, ({ privatekey, publicKey }) => ({
    privateKey: PrivateKey.toBase58(privatekey),
    publicKey: PublicKey.toBase58(publicKey),
  })),
  signature: withInvalidBase58(map(signature, Signature.toBase58)),
  signatureJson: map(signature, Signature.toJSON),
  field: mapWithInvalid(field, Field.toJSON),
};

function withInvalidRandomString<T extends string>(rng: Random<T>) {
  return { ...rng, invalid: string(30) as Random<T> };
}

type JsonGenerators = {
  [K in keyof Types]: Types[K] extends ProvableExtended<any, infer J>
    ? Random<J>
    : never;
};
const JsonGenerators: JsonGenerators = {
  Field: json_.field,
  Bool: boolean,
  UInt32: json_.uint32,
  UInt64: json_.uint64,
  Sign: withInvalidRandomString(map(sign, Sign.toJSON)),
  PublicKey: json_.publicKey,
  TokenId: withInvalidBase58(map(tokenId, TokenId.toJSON)),
  StateHash: withInvalidBase58(map(stateHash, StateHash.toJSON)),
  AuthRequired: withInvalidRandomString(map(authRequired, AuthRequired.toJSON)),
  TokenSymbol: Object.assign(tokenSymbolString, {
    invalid: string(int(tokenSymbolLength + 1, 20)),
  }),
  Events: mapWithInvalid(events, Events.toJSON),
  Actions: mapWithInvalid(actions, Actions.toJSON),
  ActionState: mapWithInvalid(actionState, ActionState.toJSON),
  VerificationKeyHash: mapWithInvalid(verificationKeyHash, Field.toJSON),
  ReceiptChainHash: mapWithInvalid(receiptChainHash, ReceiptChainHash.toJSON),
  ZkappUri: string(nat(50)),
  null: constant(null),
  string: base58(nat(50)),
  number: nat(3),
};
let typeToJsonGenerator = new Map<Provable<any>, Random<any>>(
  [TypeMap, PrimitiveMap, customTypes]
    .map(Object.entries)
    .flat()
    .map(([key, value]) => [value, JsonGenerators[key as keyof JsonGenerators]])
);

const accountUpdateJson = mapWithInvalid(
  generatorFromLayout<Json.AccountUpdate>(jsLayout.AccountUpdate as any, {
    isJson: true,
  }),
  (a) => {
    // TODO set proof to null since we can't generate a valid random one
    a.authorization.proof = null;
    // TODO set signature to null since the deriver encodes it as arbitrary string
    a.authorization.signature = null;
    // ensure authorization kind is valid
    let { isProved, isSigned } = a.body.authorizationKind;
    if (isProved && isSigned) {
      a.body.authorizationKind.isProved = false;
    }

    if (!a.body.authorizationKind.isProved) {
      a.body.authorizationKind.verificationKeyHash =
        mocks.dummyVerificationKeyHash;
    }
    // ensure mayUseToken is valid
    let { inheritFromParent, parentsOwnToken } = a.body.mayUseToken;
    if (inheritFromParent && parentsOwnToken) {
      a.body.mayUseToken.inheritFromParent = false;
    }
    return a;
  }
);
const feePayerJson = generatorFromLayout<Json.ZkappCommand['feePayer']>(
  jsLayout.ZkappCommand.entries.feePayer as any,
  { isJson: true }
);

const json = {
  ...json_,
  accountUpdate: accountUpdateJson,
  feePayer: feePayerJson,
  memoString,
};

const Random = Object.assign(Random_, {
  constant,
  int,
  nat,
  fraction,
  boolean,
  byte,
  bytes,
  string,
  base58,
  array: Object.assign(array, { ofSize: arrayOfSizeValid }),
  record,
  map: Object.assign(map, { withInvalid: mapWithInvalid }),
  step,
  oneOf,
  withHardCoded,
  dependent,
  apply,
  reject,
  dice: Object.assign(dice, { ofSize: diceOfSize() }),
  field,
  bool,
  uint32,
  uint64,
  biguint: biguintWithInvalid,
  bignat: bignatWithInvalid,
  privateKey,
  publicKey,
  scalar,
  signature,
  accountUpdate,
  feePayer,
  memo,
  json,
});

function generatorFromLayout<T>(
  typeData: Layout,
  { isJson }: { isJson: boolean }
): Random<T> {
  let typeToGenerator = isJson ? typeToJsonGenerator : typeToBigintGenerator;
  return genericLayoutFold<undefined, Random<any>, TypeMap, Json.TypeMap>(
    TypeMap,
    customTypes,
    {
      map(type, _, name) {
        let rng = typeToGenerator.get(type);
        if (rng === undefined)
          throw Error(&nbsp;could not find generator for type ${name}&nbsp;);
        return rng;
      },
      reduceArray(_, typeData) {
        let element = generatorFromLayout(typeData.inner, { isJson });
        let size = typeData.staticLength ?? Random.nat(20);
        return array(element, size);
      },
      reduceObject(keys, object) {
        // hack to not sample invalid vk hashes (because vk hash is correlated with other fields, and has to be
overriden)
        if (keys.includes('verificationKeyHash')) {
          (object as any).verificationKeyHash = noInvalid(
            (object as any).verificationKeyHash
          );
        }
        return record(object);
      },
      reduceFlaggedOption({ isSome, value }, typeData) {
        if (isJson) {
          return oneOf(null, value);
        } else {
          return mapWithInvalid(isSome, value, (isSome, value) => {
            let isSomeBoolean = TypeMap.Bool.toJSON(isSome);
            if (!isSomeBoolean) return emptyValue(typeData);
            return { isSome, value };
          });
        }
      },
      reduceOrUndefined(_, innerTypeData) {
        return oneOf(
          isJson ? null : undefined,
          generatorFromLayout(innerTypeData, { isJson })
        );
      },
    },
    typeData,
    undefined
  );
}

function constant<T>(t: T) {
  return Random_(() => t);
}

function bytes(size: number | Random<number>): Random<number[]> {
  return arrayValid(byte, size);
}

function uniformBytes(size: number | Random<number>): Random<number[]> {
  let size_ = typeof size === 'number' ? constant(size) : size;
  return {
    create() {
      let nextSize = size_.create();
      return () => [...randomBytes(nextSize())];
    },
  };
}
```

```
function string(size: number | Random<number>) {
  return map(uniformBytes(size), (b) => String.fromCharCode(...b));
}
function base58(size: number | Random<number>) {
  return map(arrayValid(oneOf(...alphabet), size), (a) => a.join(''));
}

function isGenerator<T>(rng: any): rng is Random<T> {
  return typeof rng === 'object' && rng && 'create' in rng;
}

function oneOf<Types extends readonly any[]>(
  ...values: {
    [K in keyof Types]:
      | Random<Types[K]>
      | RandomWithInvalid<Types[K]>
      | Types[K];
  }
): Random<Types[number]> {
  let gens = values.map(maybeConstant);
  let valid = {
    create() {
      let nexts = gens.map((rng) => rng.create());
      return () => {
        let i = drawUniformUint(values.length - 1);
        return nexts[i]();
      };
    },
  };
  let invalidGens = gens
    .filter((g) => g.invalid !== undefined)
    .map((g) => g.invalid!);
  let nInvalid = invalidGens.length;
  if (nInvalid === 0) return valid;
  let invalid = {
    create() {
      let nexts = invalidGens.map((rng) => rng.create());
      return () => {
        let i = drawUniformUint(nInvalid - 1);
        return nexts[i]();
      };
    },
  };
  return Object.assign(valid, { invalid });
}

/**
 * map a list of generators to a new generator, by specifying the transformation which maps samples
 * of the input generators to a sample of the result.
 */
function map<T extends readonly any[], S>(
  ...args: [...rngs: { [K in keyof T]: Random<T[K]> }, to: (...values: T) => S]
```

```
): Random<S> {
  const to = args.pop()! as (...values: T) => S;
  let rngs = args as { [K in keyof T]: Random<T[K]> };
  return {
    create() {
      let nexts = rngs.map((rng) => rng.create());
      return () => to(...(nexts.map((next) => next()) as any));
    },
  };
}
/**
 * dependent is like {@link map}, with the difference that the mapping contains a free variable
 * whose samples have to be provided as inputs separately. this is useful to create correlated generators,
where
 * multiple generators are all dependent on the same extra variable which is sampled independently.
 *
 * dependent can be used in two different ways:
 * - as a function from a random generator of the free variable to a random generator of the result
 * - as a random generator whose samples are _functions_ from free variable to result: &nbsp;Random<(arg:
Free) => Result>&nbsp;
 */
function dependent<T extends readonly any[], Result, Free>(
  ...args: [
    ...rngs: { [K in keyof T]: Random<T[K]> },
    to: (free: Free, values: T) => Result
  ]
): Random<(arg: Free) => Result> & ((arg: Random<Free>) => Random<Result>) {
  const to = args.pop()! as (free: Free, values: T) => Result;
  let rngs = args as { [K in keyof T]: Random<T[K]> };
  let rng: Random<(arg: Free) => Result> = {
    create() {
      let nexts = rngs.map((rng) => rng.create());
      return () => (free) => to(free, nexts.map((next) => next()) as any);
    },
  };
  return Object.assign(function (free: Random<Free>): Random<Result> {
    return {
      create() {
        let freeNext = free.create();
        let nexts = rngs.map((rng) => rng.create());
        return () => to(freeNext(), nexts.map((next) => next()) as any);
      },
    };
  }, rng);
}

function step<T extends readonly any[], S>(
  ...args: [
    ...rngs: { [K in keyof T]: Random<T[K]> },
    step: (current: S, ...values: T) => S,
    initial: S
  ]
```

```
): Random<S> {
  let initial = args.pop()! as S;
  const step = args.pop()! as (current: S, ...values: T) => S;
  let rngs = args as { [K in keyof T]: Random<T[K]> };
  return {
    create() {
      let nexts = rngs.map((rng) => rng.create());
      let next = initial;
      let current = initial;
      return () => {
        current = next;
        next = step(current, ...(nexts.map((next) => next()) as any as T));
        return current;
      };
    },
  };
}

function arrayValid<T>(
  element: Random<T>,
  size: number | Random<number>,
  { reset = false } = {}
): Random<T[]> {
  let size_ = typeof size === 'number' ? constant(size) : size;
  return {
    create() {
      let nextSize = size_.create();
      let nextElement = element.create();
      return () => {
        let nextElement_ = reset ? element.create() : nextElement;
        return Array.from({ length: nextSize() }, nextElement_);
      };
    },
  };
}
function arrayOfSizeValid<T>(
  element: Random<T>,
  { reset = false } = {}
): Random<(n: number) => T[]> {
  return {
    create() {
      let nextElement = element.create();
      return () => (length: number) => {
        let nextElement_ = reset ? element.create() : nextElement;
        return Array.from({ length }, nextElement_);
      };
    },
  };
}

function recordValid<T extends {}>(gens: {
  [K in keyof T]: Random<T[K]>;
```

```
}): Random<T> {
  return {
    create() {
      let keys = Object.keys(gens);
      let nexts = keys.map((key) => gens[key as keyof T].create());
      return () =>
        Object.fromEntries(keys.map((key, i) => [key, nexts[i]()])) as T;
    },
  };
}

function tupleValid<T extends readonly any[]>(
  gens: {
    [i in keyof T & number]: Random<T[i]>;
  } & Random<any>[]
): Random<T> {
  return {
    create() {
      let nexts = gens.map((gen) => gen.create());
      return () => nexts.map((next) => next()) as any;
    },
  };
}

function reject<T>(rng: Random<T>, isRejected: (t: T) => boolean): Random<T> {
  return {
    create() {
      let next = rng.create();
      return () => {
        while (true) {
          let t = next();
          if (!isRejected(t)) return t;
        }
      };
    },
  };
}

type Action<S> = Random<(s: S) => S>;
function apply<S>(
  rng: Random<S>,
  howMany: number | Random<number>,
  ...actions: Action<S>[]
): Random<S> {
  let howMany_ = maybeConstant(howMany);
  let action = oneOf(...actions);
  return {
    create() {
      let next = rng.create();
      let nextSize = howMany_.create();
      let nextAction = action.create();
      return () => {
```

```
      let state = next();
      let size = nextSize();
      for (let i = 0; i < size; i++) {
        let action = nextAction();
        state = action(state);
      }
      return state;
    };
  },
 };
}

function withHardCoded<T>(rng: Random<T>, ...hardCoded: T[]): Random<T> {
 return {
  create() {
    let next = rng.create();
    let i = 0;
    return () => {
      if (i < hardCoded.length) return hardCoded[i++];
      return next();
    };
  },
 };
}

function maybeConstant<T>(c: T | Random<T>): Random<T> {
 return isGenerator(c) ? c : constant(c);
}

/**
 * uniform distribution over range [min, max]
 * with bias towards special values 0, 1, -1, 2, min, max
 */
function int(min: number, max: number): Random<number> {
 if (max < min) throw Error('max < min');
 // set of special numbers that will appear more often in tests
 let specialSet = new Set<number>();
 if (-1 >= min && -1 <= max) specialSet.add(-1);
 if (1 >= min && 1 <= max) specialSet.add(1);
 if (2 >= min && 2 <= max) specialSet.add(2);
 specialSet.add(min);
 specialSet.add(max);
 let special = [...specialSet];
 if (0 >= min && 0 <= max) special.unshift(0, 0);
 let nSpecial = special.length;
 return {
  create: () => () => {
    // 25% of test cases are special numbers
    if (drawOneOf8() < 3) {
      let i = drawUniformUint(nSpecial - 1);
      return special[i];
    }
```

```
      // the remaining follow a uniform distribution
      return min + drawUniformUint(max - min);
    },
  };
}

/**
 * log-uniform distribution over range [0, max]
 * with bias towards 0, 1, 2
 */
function bignat(max: bigint): Random<bigint> {
  if (max < 0n) throw Error('max < 0');
  if (max === 0n) return constant(0n);
  let bits = max.toString(2).length;
  let bitBits = bits.toString(2).length;
  // set of special numbers that will appear more often in tests
  let special = [0n, 0n, 1n];
  if (max > 1n) special.push(2n);
  let nSpecial = special.length;
  return {
    create: () => () => {
      // 25% of test cases are special numbers
      if (drawOneOf8() < 3) {
        let i = drawUniformUint(nSpecial - 1);
        return special[i];
      }
      // the remaining follow a log-uniform / cut off exponential distribution:
      // we sample a bit length (within a target range) and then a number with that length
      while (true) {
        // draw bit length from [1, 2**bitBits); reject if > bit length of max
        let bitLength = 1 + drawUniformUintBits(bitBits);
        if (bitLength > bits) continue;
        // draw number from [0, 2**bitLength); reject if > max
        let n = drawUniformBigUintBits(bitLength);
        if (n <= max) return n;
      }
    },
  };
}

/**
 * log-uniform distribution over range [0, max]
 * with bias towards 0, 1, 2
 */
function nat(max: number): Random<number> {
  return map(bignat(BigInt(max)), (n) => Number(n));
}

function fraction(fixedPrecision = 3) {
  let denom = 10 ** fixedPrecision;
  if (fixedPrecision < 1) throw Error('precision must be > 1');
  let next = () => (drawUniformUint(denom - 2) + 1) / denom;
  return { create: () => next };
}

/**
 * unbiased, uniform distribution over range [0, max-1]
 */
function dice(max: number): Random<number> {
  if (max < 1) throw Error('max as to be > 0');
  return {
    create: () => () => drawUniformUint(max - 1),
  };
}
function diceOfSize(): Random<(size: number) => number> {
  return {
    create: () => () => (max: number) => {
      if (max < 1) throw Error('max as to be > 0');
      return drawUniformUint(max - 1);
    },
  };
}

let specialBytes = [0, 0, 0, 1, 1, 2, 255, 255];
/**
 * log-uniform distribution over range [0, 255]
 * with bias towards 0, 1, 2, 255
 */
function drawRandomByte() {
  // 25% of test cases are special numbers
  if (drawOneOf8() < 2) return specialBytes[drawOneOf8()];
  // the remaining follow log-uniform / cut off exponential distribution:
  // we sample a bit length from [1, 8] and then a number with that length
  let bitLength = 1 + drawOneOf8();
  return drawUniformUintBits(bitLength);
}

/**
 * log-uniform distribution over 2^n-bit range
 * with bias towards 0, 1, 2, max
 * outputs are bigints
 */
function biguint(bits: number): Random<bigint> {
  let max = (1n << BigInt(bits)) - 1n;
  let special = [0n, 0n, 0n, 1n, 1n, 2n, max, max];
  let bitsBits = Math.log2(bits);
  if (!Number.isInteger(bitsBits)) throw Error('bits must be a power of 2');
  return {
    create: () => () => {
      // 25% of test cases are special numbers
      if (drawOneOf8() < 2) return special[drawOneOf8()];
      // the remaining follow log-uniform / cut off exponential distribution:
      // we sample a bit length from [1, 8] and then a number with that length
      let bitLength = 1 + drawUniformUintBits(bitsBits);
      return drawUniformBigUintBits(bitLength);
    },
  };
}

/**
 * uniform positive integer in [0, max] drawn from secure randomness,
 */
function drawUniformUint(max: number) {
  if (max === 0) return 0;
  let bitLength = Math.floor(Math.log2(max)) + 1;
  while (true) {
    // values with same bit length can be too large by a factor of at most 2; those are rejected
    let n = drawUniformUintBits(bitLength);
    if (n <= max) return n;
  }
}

/**
 * uniform positive integer drawn from secure randomness,
 * given a target bit length
 */
function drawUniformUintBits(bitLength: number) {
  let byteLength = Math.ceil(bitLength / 8);
  // draw random bytes, zero the excess bits
  let bytes = randomBytes(byteLength);
  if (bitLength % 8 !== 0) {
    bytes[byteLength - 1] &= (1 << bitLength % 8) - 1;
  }
  // accumulate bytes to integer
  let n = 0;
  let bitPosition = 0;
  for (let byte of bytes) {
    n += byte << bitPosition;
    bitPosition += 8;
  }
  return n;
}

/**
 * uniform positive bigint drawn from secure randomness,
 * given a target bit length
 */
function drawUniformBigUintBits(bitLength: number) {
  let byteLength = Math.ceil(bitLength / 8);
  // draw random bytes, zero the excess bits
  let bytes = randomBytes(byteLength);
  if (bitLength % 8 !== 0) {
    bytes[byteLength - 1] &= (1 << bitLength % 8) - 1;
  }
  return bytesToBigInt(bytes);
}
```

```
/**
 * draw number between 0,..,7 using secure randomness
 */
function drawOneOf8() {
  return randomBytes(1)[0] >> 5;
}

// generators for invalid samples
// note: these only cover invalid samples with a _valid type_.
// for example, numbers that are out of range or base58 strings with invalid characters.
// what we don't cover is something like passing numbers where strings are expected

// convention is that invalid generators sit next to valid ones
// so you can use uint64.invalid, array(uint64, 10).invalid, etc

/**
 * we get invalid uints by sampling from a larger range plus negative numbers
 */
function biguintWithInvalid(bits: number): RandomWithInvalid<bigint> {
  let valid = biguint(bits);
  let max = 1n << BigInt(bits);
  let double = biguint(2 * bits);
  let negative = map(double, (uint) => -uint - 1n);
  let tooLarge = map(valid, (uint) => uint + max);
  let invalid = oneOf(negative, tooLarge);
  return Object.assign(valid, { invalid });
}

function bignatWithInvalid(max: bigint): RandomWithInvalid<bigint> {
  let valid = bignat(max);
  let double = bignat(2n * max);
  let negative = map(double, (uint) => -uint - 1n);
  let tooLarge = map(valid, (uint) => uint + max);
  let invalid = oneOf(negative, tooLarge);
  return Object.assign(valid, { invalid });
}

function fieldWithInvalid(
  F: typeof Field | typeof Scalar
): RandomWithInvalid<bigint> {
  let randomField = Random_(F.random);
  let specialField = oneOf(0n, 1n, F(-1));
  let field = oneOf<bigint[]>(randomField, randomField, uint64, specialField);
  let tooLarge = map(field, (x) => x + F.modulus);
  let negative = map(field, (x) => -x - 1n);
  let invalid = oneOf(tooLarge, negative);
  return Object.assign(field, { invalid });
}

function publicKeyWithInvalid() {
  let publicKey = map(privateKey, PrivateKey.toPublicKey);
  let invalidX = reject(field, (x) =>
    Field.isSquare(Field.add(Field.power(x, 3n), Group.b))
  );
  let invalid = map(invalidX, bool, (x, isOdd): PublicKey => ({ x, isOdd }));
  return Object.assign(publicKey, { invalid });
}

/**
 * invalid arrays are sampled by generating an array with exactly one invalid input (and any number of valid
inputs);
 * (note: invalid arrays have the same length distribution as valid ones, except that they are never empty)
 */
function array<T>(
  element: Random<T>,
  size: number | Random<number>,
  options?: { reset?: boolean }
): Random<T[]> {
  let valid = arrayValid(element, size, options);
  if (element.invalid === undefined) return valid;
  let invalid = map(valid, element.invalid, (arr, invalid) => {
    if (arr.length === 0) return [invalid];
    let i = drawUniformUint(arr.length - 1);
    arr[i] = invalid;
    return arr;
  });
  return { ...valid, invalid };
}
/**
 * invalid records are similar to arrays: randomly choose one of the fields that have an invalid generator,
 * and set it to its invalid value
 */
function record<T extends {}>(gens: {
  [K in keyof T]: Random<T[K]>;
}): Random<T> {
  let valid = recordValid(gens);
  let invalidFields: [string & keyof T, Random<any>][] = [];
  for (let key in gens) {
    let invalid = gens[key].invalid;
    if (invalid !== undefined) {
      invalidFields.push([key, invalid]);
    }
  }
  let nInvalid = invalidFields.length;
  if (nInvalid === 0) return valid;
  let invalid = {
    create() {
      let next = valid.create();
      let invalidNexts = invalidFields.map(
        ([key, rng]) => [key, rng.create()] as const
      );
      return () => {
        let value = next();
      let i = drawUniformUint(nInvalid - 1);
      let [key, invalidNext] = invalidNexts[i];
      value[key] = invalidNext();
      return value;
    };
  },
 };
 return { ...valid, invalid };
}
/**
 * invalid tuples are like invalid records
 */
function tuple<T extends readonly any[]>(
 gens: {
  [K in keyof T & number]: Random<T[K]>;
 } & Random<any>[]
): Random<T> {
 let valid = tupleValid<T>(gens);
 let invalidFields: [number & keyof T, Random<any>][] = [];
 gens.forEach((gen, i) => {
  let invalid = gen.invalid;
  if (invalid !== undefined) {
    invalidFields.push([i, invalid]);
  }
 });
 let nInvalid = invalidFields.length;
 if (nInvalid === 0) return valid;
 let invalid = {
  create() {
    let next = valid.create();
    let invalidNexts = invalidFields.map(
      ([key, rng]) => [key, rng.create()] as const
    );
    return () => {
      let value = next();
      let i = drawUniformUint(nInvalid - 1);
      let [key, invalidNext] = invalidNexts[i];
      value[key] = invalidNext();
      return value;
    };
  },
 };
 return { ...valid, invalid };
}
/**
 * map assuming that invalid inputs can be mapped just like valid ones.
 * _one_ of the inputs is sampled as invalid
 */
function mapWithInvalid<T extends readonly any[], S>(
 ...args: [...rngs: { [K in keyof T]: Random<T[K]> }, to: (...values: T) => S]
): Random<S> {
 const to = args.pop()! as (...values: T) => S;
```

```
  let rngs = args as { [K in keyof T]: Random<T[K]> };
  let valid = map<T, S>(...rngs, to);
  let invalidInput = tuple<T>(rngs as Random<any>[]).invalid;
  if (invalidInput === undefined) return valid;
  let invalid = {
    create() {
      let nextInput = invalidInput!.create();
      return () => to(...nextInput());
    },
  };
  return { ...valid, invalid };
}

function noInvalid<T>(rng: Random<T>): Random<T> {
  return { ...rng, invalid: undefined };
}

// functions to create invalid base58
let n = alphabet.length;

function replaceCharacter(string: string, i: number, char: string) {
  return string.slice(0, i) + char + string.slice(i + 1);
}

function makeCheckSumInvalid(base58: string) {
  if (base58.length === 0) return base58;
  // pick any character, and change it to any different one
  let iChar = drawUniformUint(base58.length - 1);
  let iAlph = alphabet.indexOf(base58[iChar]);
  let iAlphNew = (iAlph + 1 + drawUniformUint(n - 2)) % n;
  return replaceCharacter(base58, iChar, alphabet[iAlphNew]);
}

function makeBase58Invalid(base58: string) {
  let iChar = drawUniformUint(base58.length - 1);
  // sample a character that is not in the alphabet
  let char: string;
  while (true) {
    let [byte] = randomBytes(1);
    char = String.fromCharCode(byte);
    if (!alphabet.includes(char)) break;
  }
  return replaceCharacter(base58, iChar, char);
}

function withInvalidBase58(rng: Random<string>): RandomWithInvalid<string> {
  let invalidBase58 = apply(
    rng,
    1,
    constant(makeBase58Invalid),
    constant(makeCheckSumInvalid)
  );
  let invalid =
    rng.invalid === undefined
      ? invalidBase58
      : oneOf(invalidBase58, rng.invalid);
  return { ...rng, invalid };
}
```

</file>

<file>

## path: /src/lib/testing/testing.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/testing/testing.unit-test.ts

```
import { expect } from 'expect';
import { jsLayout } from '../../bindings/mina-transaction/gen/js-layout.js';
import { Signature } from '../../mina-signer/src/signature.js';
import {
  AccountUpdate,
  PublicKey,
  UInt32,
  UInt64,
  provableFromLayout,
  ZkappCommand,
  Json,
} from '../../bindings/mina-transaction/gen/transaction-bigint.js';
import { test, Random, sample } from './property.js';

// some trivial roundtrip tests
test(Random.accountUpdate, (accountUpdate, assert) => {
  let json = AccountUpdate.toJSON(accountUpdate);
  let jsonString = JSON.stringify(json);
  assert(
    jsonString ===
      JSON.stringify(AccountUpdate.toJSON(AccountUpdate.fromJSON(json)))
  );
  let fields = AccountUpdate.toFields(accountUpdate);
  let auxiliary = AccountUpdate.toAuxiliary(accountUpdate);
  let recovered = AccountUpdate.fromFields(fields, auxiliary);
  assert(jsonString === JSON.stringify(AccountUpdate.toJSON(recovered)));
});
test(Random.json.accountUpdate, (json) => {
  let jsonString = JSON.stringify(json);
  expect(jsonString).toEqual(
    JSON.stringify(AccountUpdate.toJSON(AccountUpdate.fromJSON(json)))
  );
});

// check that test fails for a property that does not hold in general
let testSilent = test.custom({ logFailures: false });
```

```
expect(() => {
  testSilent(Random.nat(100), Random.nat(100), (x, y, assert) => {
    assert(x !== y, 'two different numbers can never be the same');
  });
}).toThrow('two different numbers');

// check that invalid JSON cannot be parsed
// note: test.negative asserts that _every_ sample fails
test.negative(Random.json.uint64.invalid, UInt64.fromJSON);
test.negative(Random.json.uint32.invalid, UInt32.fromJSON);
test.negative(Random.json.publicKey.invalid, PublicKey.fromJSON);
test.negative(Random.json.signature.invalid, Signature.fromBase58);

test.custom({ negative: true, timeBudget: 1000 })(
  Random.json.accountUpdate.invalid!,
  AccountUpdate.fromJSON
);

const FeePayer = provableFromLayout<
  ZkappCommand['feePayer'],
  Json.ZkappCommand['feePayer']
>(jsLayout.ZkappCommand.entries.feePayer as any);

test.negative(Random.json.feePayer.invalid!, FeePayer.fromJSON);
```

</file>

<file>

## path: /src/lib/token.test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/token.test.ts

```
import {
  State,
  state,
  UInt64,
  Bool,
  SmartContract,
  Mina,
  PrivateKey,
  AccountUpdate,
  method,
  PublicKey,
  Permissions,
  VerificationKey,
  Field,
  Experimental,
  Int64,
  TokenId,
} from 'o1js';
```

```
const tokenSymbol = 'TOKEN';

class TokenContract extends SmartContract {
 SUPPLY = UInt64.from(10n ** 18n);
 @state(UInt64) totalAmountInCirculation = State<UInt64>();

 /**
  * This deploy method lets a another token account deploy their zkApp and verification key as a child of this
token contract.
  * This is important since we want the native token id of the deployed zkApp to be the token id of the token
contract.
  */
 @method deployZkapp(address: PublicKey, verificationKey: VerificationKey) {
   let tokenId = this.token.id;
   let zkapp = AccountUpdate.defaultAccountUpdate(address, tokenId);
   this.approve(zkapp);
   zkapp.account.permissions.set(Permissions.default());
   zkapp.account.verificationKey.set(verificationKey);
   zkapp.requireSignature();
 }

 init() {
   super.init();
   let address = this.address;
   let receiver = this.token.mint({
     address,
     amount: this.SUPPLY,
   });
   receiver.account.isNew.assertEquals(Bool(true));
   this.balance.subInPlace(Mina.accountCreationFee());
   this.totalAmountInCirculation.set(this.SUPPLY.sub(100_000_000));
   this.account.permissions.set({
     ...Permissions.default(),
     editState: Permissions.proofOrSignature(),
     receive: Permissions.proof(),
     access: Permissions.proofOrSignature(),
   });
 }

 @method mint(receiverAddress: PublicKey, amount: UInt64) {
   let totalAmountInCirculation = this.totalAmountInCirculation.get();
   this.totalAmountInCirculation.assertEquals(totalAmountInCirculation);
   let newTotalAmountInCirculation = totalAmountInCirculation.add(amount);
   newTotalAmountInCirculation.value.assertLessThanOrEqual(
     this.SUPPLY.value,
     "Can't mint more than the total supply"
   );
   this.token.mint({
     address: receiverAddress,
     amount,
   });
```

```
    this.totalAmountInCirculation.set(newTotalAmountInCirculation);
  }

  @method burn(receiverAddress: PublicKey, amount: UInt64) {
    let totalAmountInCirculation = this.totalAmountInCirculation.get();
    this.totalAmountInCirculation.assertEquals(totalAmountInCirculation);
    let newTotalAmountInCirculation = totalAmountInCirculation.sub(amount);
    totalAmountInCirculation.value.assertGreaterThanOrEqual(
      UInt64.from(0).value,
      "Can't burn less than 0"
    );
    this.token.burn({
      address: receiverAddress,
      amount,
    });
    this.totalAmountInCirculation.set(newTotalAmountInCirculation);
  }

  @method approveTransferCallback(
    senderAddress: PublicKey,
    receiverAddress: PublicKey,
    amount: UInt64,
    callback: Experimental.Callback<any>
  ) {
    let layout = AccountUpdate.Layout.NoChildren; // Allow only 1 accountUpdate with no children
    let senderAccountUpdate = this.approve(callback, layout);
    let negativeAmount = Int64.fromObject(
      senderAccountUpdate.body.balanceChange
    );
    negativeAmount.assertEquals(Int64.from(amount).neg());
    let tokenId = this.token.id;
    senderAccountUpdate.body.tokenId.assertEquals(tokenId);
    senderAccountUpdate.body.publicKey.assertEquals(senderAddress);
    let receiverAccountUpdate = Experimental.createChildAccountUpdate(
      this.self,
      receiverAddress,
      tokenId
    );
    receiverAccountUpdate.balance.addInPlace(amount);
  }
}

class ZkAppB extends SmartContract {
  @method approveZkapp(amount: UInt64) {
    this.balance.subInPlace(amount);
  }
}

class ZkAppC extends SmartContract {
  @method approveZkapp(amount: UInt64) {
    this.balance.subInPlace(amount);
  }
  @method approveIncorrectLayout(amount: UInt64) {
    this.balance.subInPlace(amount);
    let update = AccountUpdate.defaultAccountUpdate(this.address);
    this.self.approve(update);
  }
}

let feePayerKey: PrivateKey;
let feePayer: PublicKey;
let tokenZkappKey: PrivateKey;
let tokenZkappAddress: PublicKey;
let tokenZkapp: TokenContract;
let tokenId: Field;

let zkAppBKey: PrivateKey;
let zkAppBAddress: PublicKey;
let zkAppB: ZkAppB;

let zkAppCKey: PrivateKey;
let zkAppCAddress: PublicKey;
let zkAppC: ZkAppC;

function setupAccounts() {
  let Local = Mina.LocalBlockchain({
    proofsEnabled: true,
    enforceTransactionLimits: false,
  });
  Mina.setActiveInstance(Local);
  feePayerKey = Local.testAccounts[0].privateKey;
  feePayer = Local.testAccounts[0].publicKey;

  tokenZkappKey = PrivateKey.random();
  tokenZkappAddress = tokenZkappKey.toPublicKey();

  tokenZkapp = new TokenContract(tokenZkappAddress);
  tokenId = tokenZkapp.token.id;

  zkAppBKey = Local.testAccounts[1].privateKey;
  zkAppBAddress = zkAppBKey.toPublicKey();
  zkAppB = new ZkAppB(zkAppBAddress, tokenId);

  zkAppCKey = Local.testAccounts[2].privateKey;
  zkAppCAddress = zkAppCKey.toPublicKey();
  zkAppC = new ZkAppC(zkAppCAddress, tokenId);
  return Local;
}

async function setupLocal() {
  setupAccounts();
  let tx = await Mina.transaction(feePayer, () => {
    let feePayerUpdate = AccountUpdate.fundNewAccount(feePayer);
    feePayerUpdate.send({
      to: tokenZkappAddress,
      amount: Mina.accountCreationFee(),
    });
    tokenZkapp.deploy();
  });
  tx.sign([tokenZkappKey, feePayerKey]);
  await tx.send();
}

async function setupLocalProofs() {
  let Local = setupAccounts();
  zkAppC = new ZkAppC(zkAppCAddress, tokenId);
  // don't use proofs for the setup, takes too long to do this every time
  Local.setProofsEnabled(false);
  let tx = await Mina.transaction({ sender: feePayer }, () => {
    let feePayerUpdate = AccountUpdate.fundNewAccount(feePayer, 3);
    feePayerUpdate.send({
      to: tokenZkappAddress,
      amount: Mina.accountCreationFee(),
    });
    tokenZkapp.deploy();
    tokenZkapp.deployZkapp(zkAppBAddress, ZkAppB._verificationKey!);
    tokenZkapp.deployZkapp(zkAppCAddress, ZkAppC._verificationKey!);
  });
  await tx.prove();
  tx.sign([tokenZkappKey, zkAppBKey, zkAppCKey, feePayerKey]);
  await tx.send();
  Local.setProofsEnabled(true);
}

describe('Token', () => {
  beforeAll(async () => {
    await TokenContract.compile();
    await ZkAppB.compile();
    await ZkAppC.compile();
  });

  describe('Signature Authorization', () => {
    /*
      test case description:
      Check token contract can be deployed and initialized
      tested cases:
        - create a new token
        - deploy a zkApp under a custom token
        - create a new valid token with a different parentTokenId
        - set the token symbol after deployment
    */
    describe('Token Contract Creation/Deployment', () => {
      beforeEach(async () => {
        await setupLocal();
      });
      test('correct token id can be derived with an existing token owner', () => {
        expect(tokenId).toEqual(TokenId.derive(tokenZkappAddress));
      });

      test('deployed token contract exists in the ledger', () => {
        expect(Mina.getAccount(tokenZkappAddress, tokenId)).toBeDefined();
      });

      test('setting a valid token symbol on a token contract', async () => {
        await (
          await Mina.transaction({ sender: feePayer }, () => {
            let tokenZkapp = AccountUpdate.createSigned(tokenZkappAddress);
            tokenZkapp.account.tokenSymbol.set(tokenSymbol);
          })
        )
          .sign([feePayerKey, tokenZkappKey])
          .send();
        const symbol = Mina.getAccount(tokenZkappAddress).tokenSymbol;
        expect(tokenSymbol).toBeDefined();
        expect(symbol).toEqual(tokenSymbol);
      });
    });

    /*
      test case description:
      token contract can mint new tokens with a signature
      tested cases:
        - mints and updates the token balance of the receiver
        - fails if we mint over an overflow amount
    */
    describe('Mint token', () => {
      beforeEach(async () => {
        await setupLocal();
      });

      test('token contract can successfully mint and updates the balances in the ledger (signature)', async ()
=> {
        await (
          await Mina.transaction({ sender: feePayer }, () => {
            AccountUpdate.fundNewAccount(feePayer);
            tokenZkapp.mint(zkAppBAddress, UInt64.from(100_000));
            tokenZkapp.requireSignature();
          })
        )
          .sign([feePayerKey, tokenZkappKey])
          .send();
        expect(
          Mina.getBalance(zkAppBAddress, tokenId).value.toBigInt()
        ).toEqual(100_000n);
      });
      test('minting should fail if overflow occurs ', async () => {
        await Mina.transaction(feePayer, () => {
          AccountUpdate.fundNewAccount(feePayer);
          tokenZkapp.mint(zkAppBAddress, UInt64.from(100_000_000_000));
          tokenZkapp.requireSignature();
        }).catch((e) => {
          expect(e).toBeDefined();
        });
      });
    });

    /*
      test case description:
      token contract can burn tokens with a signature
      tested cases:
        - burns and updates the token balance of the receiver
        - fails if we burn more than the balance amount
    */
    describe('Burn token', () => {
      beforeEach(async () => {
        await setupLocal();
      });
      test('token contract can successfully burn and updates the balances in the ledger (signature)', async ()
=> {
        await (
          await Mina.transaction(feePayer, () => {
            AccountUpdate.fundNewAccount(feePayer);
            tokenZkapp.mint(zkAppBAddress, UInt64.from(100_000));
            tokenZkapp.requireSignature();
          })
        )
          .sign([feePayerKey, tokenZkappKey])
          .send();
        await (
          await Mina.transaction(feePayer, () => {
            tokenZkapp.burn(zkAppBAddress, UInt64.from(10_000));
            tokenZkapp.requireSignature();
          })
        )
          .sign([zkAppBKey, feePayerKey, tokenZkappKey])
          .send();
        expect(
          Mina.getBalance(zkAppBAddress, tokenId).value.toBigInt()
        ).toEqual(90_000n);
      });

      test('throw error if token owner burns more tokens than token account has', async () => {
        await (
          await Mina.transaction(feePayer, () => {
            AccountUpdate.fundNewAccount(feePayer);
            tokenZkapp.mint(zkAppBAddress, UInt64.from(1_000));
            tokenZkapp.requireSignature();
          })
        )
          .sign([feePayerKey, tokenZkappKey])
          .send();
        let tx = (
          await Mina.transaction(feePayer, () => {
            tokenZkapp.burn(zkAppBAddress, UInt64.from(10_000));
            tokenZkapp.requireSignature();
          })
        ).sign([zkAppBKey, feePayerKey, tokenZkappKey]);
        await expect(tx.send()).rejects.toThrow();
      });
    });

    /*
      test case description:
      token contract can transfer tokens with a signature
      tested cases:
        - sends tokens and updates the balance of the receiver
        - fails if no account creation fee is payed for the new token account
        - fails if we transfer more than the balance amount
    */
    describe('Transfer', () => {
      beforeEach(async () => {
        await setupLocal();
      });

      test('change the balance of a token account after sending', async () => {
        let tx = await Mina.transaction(feePayer, () => {
          AccountUpdate.fundNewAccount(feePayer);
          tokenZkapp.mint(zkAppBAddress, UInt64.from(100_000));
          tokenZkapp.requireSignature();
        });
        await tx.sign([feePayerKey, tokenZkappKey]).send();

        tx = await Mina.transaction(feePayer, () => {
          AccountUpdate.fundNewAccount(feePayer);
          tokenZkapp.token.send({
            from: zkAppBAddress,
            to: zkAppCAddress,
            amount: UInt64.from(10_000),
          });
          AccountUpdate.attachToTransaction(tokenZkapp.self);
          tokenZkapp.requireSignature();
        });
        tx.sign([zkAppBKey, zkAppCKey, feePayerKey, tokenZkappKey]);
        await tx.send();

        expect(
          Mina.getBalance(zkAppBAddress, tokenId).value.toBigInt()
        ).toEqual(90_000n);
        expect(
        Mina.getBalance(zkAppCAddress, tokenId).value.toBigInt()
      ).toEqual(10_000n);
    });

    test('should error creating a token account if no account creation fee is specified', async () => {
      await (
        await Mina.transaction(feePayer, () => {
          AccountUpdate.fundNewAccount(feePayer);
          tokenZkapp.mint(zkAppBAddress, UInt64.from(100_000));
          tokenZkapp.requireSignature();
        })
      )
        .sign([feePayerKey, tokenZkappKey])
        .send();
      let tx = (
        await Mina.transaction(feePayer, () => {
          tokenZkapp.token.send({
            from: zkAppBAddress,
            to: zkAppCAddress,
            amount: UInt64.from(10_000),
          });
          AccountUpdate.attachToTransaction(tokenZkapp.self);
          tokenZkapp.requireSignature();
        })
      ).sign([zkAppBKey, feePayerKey, tokenZkappKey]);

      await expect(tx.send()).rejects.toThrow();
    });

    test('should error if sender sends more tokens than they have', async () => {
      await (
        await Mina.transaction(feePayer, () => {
          AccountUpdate.fundNewAccount(feePayer);
          tokenZkapp.mint(zkAppBAddress, UInt64.from(100_000));
          tokenZkapp.requireSignature();
        })
      )
        .sign([feePayerKey, tokenZkappKey])
        .send();
      let tx = (
        await Mina.transaction(feePayer, () => {
          tokenZkapp.token.send({
            from: zkAppBAddress,
            to: zkAppCAddress,
            amount: UInt64.from(100_000),
          });
          AccountUpdate.attachToTransaction(tokenZkapp.self);
          tokenZkapp.requireSignature();
        })
      ).sign([zkAppBKey, feePayerKey, tokenZkappKey]);
      await expect(tx.send()).rejects.toThrow();
    });
```

```
  });
});

describe('Proof Authorization', () => {
  /*
    test case description:
    Check token contract can be deployed and initialized with proofs
    tested cases:
      - can deploy and initialize child contracts of the parent token contract
  */
  describe('Token Contract Creation/Deployment', () => {
    beforeEach(async () => {
      await setupLocalProofs().catch((err) => {
        console.log(err);
        throw err;
      });
    });

    test('should successfully deploy a token account under a zkApp', async () => {
      expect(Mina.getAccount(zkAppBAddress, tokenId)).toBeDefined();
      expect(Mina.getAccount(zkAppBAddress, tokenId).tokenId).toEqual(
        tokenId
      );
      expect(Mina.getAccount(zkAppCAddress, tokenId)).toBeDefined();
      expect(Mina.getAccount(zkAppCAddress, tokenId).tokenId).toEqual(
        tokenId
      );
    });
  });

  /*
    test case description:
    token contract can mint new tokens with a proof
    tested cases:
      - mints and updates the token balance of the receiver
  */
  describe('Mint token', () => {
    beforeEach(async () => {
      await setupLocal();
    });

    test('token contract can successfully mint and updates the balances in the ledger (proof)', async () => {
      let tx = await Mina.transaction(feePayer, () => {
        AccountUpdate.fundNewAccount(feePayer);
        tokenZkapp.mint(zkAppBAddress, UInt64.from(100_000));
      });
      await tx.prove();
      tx.sign([tokenZkappKey, feePayerKey]);
      await tx.send();
      expect(
        Mina.getBalance(zkAppBAddress, tokenId).value.toBigInt()
      ).toEqual(100_000n);
  });
});

describe('Burn token', () => {
  beforeEach(async () => {
    await setupLocal();
  });

  /*
  test case description:
  token contract can burn tokens with a proof
  tested cases:
    - burns and updates the token balance of the receiver
*/
  test('token contract can successfully burn and updates the balances in the ledger (proof)', async () => {
    let tx = await Mina.transaction(feePayer, () => {
      AccountUpdate.fundNewAccount(feePayer);
      tokenZkapp.mint(zkAppBAddress, UInt64.from(100_000));
      tokenZkapp.requireSignature();
    });
    await tx.sign([feePayerKey, tokenZkappKey]).send();
    tx = await Mina.transaction(feePayer, () => {
      tokenZkapp.burn(zkAppBAddress, UInt64.from(10_000));
    });
    await tx.prove();
    tx.sign([zkAppBKey, feePayerKey]);
    await tx.send();
    expect(
      Mina.getBalance(zkAppBAddress, tokenId).value.toBigInt()
    ).toEqual(90_000n);
  });
});

/*
  test case description:
  token contract can transfer tokens with a proof
  tested cases:
    - approves a transfer and updates the token balance of the sender and receiver
    - fails if we specify an incorrect layout to witness when authorizing a transfer
    - fails if we specify an empty parent accountUpdate to bypass authorization
*/
describe('Transfer', () => {
  beforeEach(async () => {
    await setupLocalProofs();
  });

  test('should approve and the balance of a token account after sending', async () => {
    let tx = await Mina.transaction(feePayer, () => {
      tokenZkapp.mint(zkAppBAddress, UInt64.from(100_000));
      tokenZkapp.requireSignature();
    });
    await tx.prove();
    await tx.sign([feePayerKey, tokenZkappKey]).send();

    tx = await Mina.transaction(feePayer, () => {
      let approveSendingCallback = Experimental.Callback.create(
        zkAppB,
        'approveZkapp',
        [UInt64.from(10_000)]
      );
      tokenZkapp.approveTransferCallback(
        zkAppBAddress,
        zkAppCAddress,
        UInt64.from(10_000),
        approveSendingCallback
      );
    });
    await tx.prove();
    await tx.sign([feePayerKey]).send();

    expect(
      Mina.getBalance(zkAppBAddress, tokenId).value.toBigInt()
    ).toEqual(90_000n);
    expect(
      Mina.getBalance(zkAppCAddress, tokenId).value.toBigInt()
    ).toEqual(10_000n);
  });

  test('should fail to approve with an incorrect layout', async () => {
    await (
      await Mina.transaction(feePayer, () => {
        tokenZkapp.mint(zkAppCAddress, UInt64.from(100_000));
        tokenZkapp.requireSignature();
      })
    )
      .sign([feePayerKey, tokenZkappKey])
      .send();

    await expect(() =>
      Mina.transaction(feePayer, () => {
        let approveSendingCallback = Experimental.Callback.create(
          zkAppC,
          'approveIncorrectLayout',
          [UInt64.from(10_000)]
        );
        tokenZkapp.approveTransferCallback(
          zkAppBAddress,
          zkAppCAddress,
          UInt64.from(10_000),
          approveSendingCallback
        );
      })
    ).rejects.toThrow();
  });
```

```
      test('should reject tx if user bypasses the token contract by using an empty account update', async () =>
{
        let tx = await Mina.transaction(feePayer, () => {
          AccountUpdate.fundNewAccount(feePayer);
          tokenZkapp.token.mint({
            address: zkAppBAddress,
            amount: UInt64.from(100_000),
          });
          AccountUpdate.attachToTransaction(tokenZkapp.self);
        });
        await expect(tx.sign([feePayerKey]).send()).rejects.toThrow(
          /Update_not_permitted_access/
        );
      });
    });
  });
});
```

</file>

<file>

## path: /src/lib/util/fs.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/util/fs.ts

```
import cachedir from 'cachedir';

export { writeFileSync, readFileSync, mkdirSync } from 'node:fs';
export { resolve } from 'node:path';
export { cachedir as cacheDir };
```

</file>

<file>

## path: /src/lib/util/fs.web.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/util/fs.web.ts

```
export {
  dummy as writeFileSync,
  dummy as readFileSync,
  dummy as resolve,
  dummy as mkdirSync,
  cacheDir,
};

function dummy() {
  throw Error('not implemented');
}

function cacheDir() {
  return '';
}
```

</file>

<file>

## path: /src/lib/util/types.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/util/types.ts

```
import { assert } from '../errors.js';

export { Tuple, TupleN };

type Tuple<T> = [T, ...T[]] | [];

const Tuple = {
  map<T extends Tuple<any>, B>(
    tuple: T,
    f: (a: T[number]) => B
  ): [...{ [i in keyof T]: B }] {
    return tuple.map(f) as any;
  },
};

/**
 * tuple type that has the length as generic parameter
 */
type TupleN<T, N extends number> = N extends N
  ? number extends N
    ? [...T[]] // N is not typed as a constant => fall back to array
    : [...TupleRec<T, N, []>]
  : never;

const TupleN = {
  map<T extends Tuple<any>, B>(
    tuple: T,
    f: (a: T[number]) => B
  ): [...{ [i in keyof T]: B }] {
    return tuple.map(f) as any;
  },

  fromArray<T, N extends number>(n: N, arr: T[]): TupleN<T, N> {
    assert(
      arr.length === n,
      &nbsp;Expected array of length ${n}, got ${arr.length}&nbsp;
    );
    return arr as any;
  },
};

type TupleRec<T, N extends number, R extends unknown[]> = R['length'] extends N
  ? R
  : TupleRec<T, N, [T, ...R]>;
```

</file>

<file>

## path: /src/lib/zkapp.ts

url: https://github.com/o1-labs/o1js/blob/main/src/lib/zkapp.ts

```
import { Gate, Pickles, ProvablePure } from '../snarky.js';
import { Field, Bool } from './core.js';
import {
  AccountUpdate,
  AccountUpdatesLayout,
  Authorization,
  Body,
  Events,
  Permissions,
  Actions,
  SetOrKeep,
  smartContractContext,
  TokenId,
  ZkappCommand,
  zkAppProver,
  ZkappPublicInput,
  ZkappStateLength,
  SmartContractContext,
} from './account_update.js';
import {
  cloneCircuitValue,
  FlexibleProvablePure,
  InferProvable,
  provable,
  Struct,
  toConstant,
} from './circuit_value.js';
import { Provable, getBlindingValue, memoizationContext } from './provable.js';
import * as Encoding from '../bindings/lib/encoding.js';
import { Poseidon, hashConstant } from './hash.js';
import { UInt32, UInt64 } from './int.js';
import * as Mina from './mina.js';
import {
  assertPreconditionInvariants,
  cleanPreconditionsCache,
} from './precondition.js';
import {
  analyzeMethod,
  compileProgram,
  Empty,
  emptyValue,
  GenericArgument,
  getPreviousProofsForProver,
  isAsFields,
  methodArgumentsToConstant,
  methodArgumentTypesAndValues,
  MethodInterface,
  Proof,
  sortMethodArguments,
} from './proof_system.js';
```

```
import { PrivateKey, PublicKey } from './signature.js';
import { assertStatePrecondition, cleanStatePrecondition } from './state.js';
import {
  inAnalyze,
  inCompile,
  inProver,
  snarkContext,
} from './provable-context.js';
import { Cache } from './proof-system/cache.js';

// external API
export {
  SmartContract,
  method,
  DeployArgs,
  declareMethods,
  Callback,
  Account,
  VerificationKey,
  Reducer,
};

const reservedPropNames = new Set(['_methods', '_']);

/**
 * A decorator to use in a zkApp to mark a method as callable by anyone.
 * You can use inside your zkApp class as:
 *
 * &nbsp;&nbsp;&nbsp;
 * \@method myMethod(someArg: Field) {
 *   // your code here
 * }
 * &nbsp;&nbsp;&nbsp;
 */
function method<T extends SmartContract>(
  target: T & { constructor: any },
  methodName: keyof T & string,
  descriptor: PropertyDescriptor
) {
  const ZkappClass = target.constructor;
  if (reservedPropNames.has(methodName)) {
    throw Error(&nbsp;Property name ${methodName} is reserved.&nbsp;);
  }
  if (typeof target[methodName] !== 'function') {
    throw Error(
      &nbsp;@method decorator was applied to \&nbsp;${methodName}\&nbsp;, which is not a
function.&nbsp;
    );
  }
  let paramTypes: Provable<any>[] = Reflect.getMetadata(
    'design:paramtypes',
    target,
  methodName
 );
 let returnType: Provable<any> = Reflect.getMetadata(
  'design:returntype',
  target,
  methodName
 );

 class SelfProof extends Proof<ZkappPublicInput, Empty> {
  static publicInputType = ZkappPublicInput;
  static publicOutputType = Empty;
  static tag = () => ZkappClass;
 }
 let internalMethodEntry = sortMethodArguments(
  ZkappClass.name,
  methodName,
  paramTypes,
  SelfProof
 );
 // add witness arguments for the publicKey (address) and tokenId
 let methodEntry = sortMethodArguments(
  ZkappClass.name,
  methodName,
  [PublicKey, Field, ...paramTypes],
  SelfProof
 );

 if (isAsFields(returnType)) {
  internalMethodEntry.returnType = returnType;
  methodEntry.returnType = returnType;
 }
 ZkappClass._methods ??= [];
 // FIXME: overriding a method implies pushing a separate method entry here, yielding two entries with the
same name
 // this should only be changed once we no longer share the _methods array with the parent class
(otherwise a subclass declaration messes up the parent class)
 ZkappClass._methods.push(methodEntry);
 ZkappClass._maxProofsVerified ??= 0;
 ZkappClass._maxProofsVerified = Math.max(
  ZkappClass._maxProofsVerified,
  methodEntry.proofArgs.length
 );
 let func = descriptor.value;
 descriptor.value = wrapMethod(func, ZkappClass, internalMethodEntry);
}

// do different things when calling a method, depending on the circumstance
function wrapMethod(
 method: Function,
 ZkappClass: typeof SmartContract,
 methodIntf: MethodInterface
) {
 let methodName = methodIntf.methodName;
 return function wrappedMethod(this: SmartContract, ...actualArgs: any[]) {
  cleanStatePrecondition(this);
  // special case: any AccountUpdate that is passed as an argument to a method
  // is unlinked from its current location, to allow the method to link it to itself
  actualArgs.forEach((arg) => {
    if (arg instanceof AccountUpdate) {
      AccountUpdate.unlink(arg);
    }
  });

  let insideContract = smartContractContext.get();
  if (!insideContract) {
    const context: SmartContractContext = {
      this: this,
      methodCallDepth: 0,
      selfUpdate: selfAccountUpdate(this, methodName),
    };
    let id = smartContractContext.enter(context);
    try {
      if (inCompile() || inProver() || inAnalyze()) {
        // important to run this with a fresh accountUpdate everytime, otherwise compile messes up our
circuits
        // because it runs this multiple times
        let proverData = inProver() ? zkAppProver.getData() : undefined;
        let txId = Mina.currentTransaction.enter({
          sender: proverData?.transaction.feePayer.body.publicKey,
          accountUpdates: [],
          fetchMode: inProver() ? 'cached' : 'test',
          isFinalRunOutsideCircuit: false,
          numberOfRuns: undefined,
        });
        try {
          // inside prover / compile, the method is always called with the public input as first argument
          // -- so we can add assertions about it
          let publicInput = actualArgs.shift();
          let accountUpdate = this.self;

          // the blinding value is important because otherwise, putting callData on the transaction would leak
information about the private inputs
          let blindingValue = Provable.witness(Field, getBlindingValue);
          // it's also good if we prove that we use the same blinding value across the method
          // that's why we pass the variable (not the constant) into a new context
          let context = memoizationContext() ?? {
            memoized: [],
            currentIndex: 0,
          };
          let id = memoizationContext.enter({ ...context, blindingValue });
          let result: unknown;
          try {
            result = method.apply(this, actualArgs.map(cloneCircuitValue));
          } finally {
```

```
      memoizationContext.leave(id);
    }

    // connects our input + result with callData, so this method can be called
    let callDataFields = computeCallData(
      methodIntf,
      actualArgs,
      result,
      blindingValue
    );
    accountUpdate.body.callData = Poseidon.hash(callDataFields);
    Authorization.setProofAuthorizationKind(accountUpdate);

    // TODO: currently commented out, but could come back in some form when we add caller to the
public input
    // // compute &nbsp;caller&nbsp; field from &nbsp;isDelegateCall&nbsp; and a context determined by
the transaction
    // let callerContext = Provable.witness(
    //   CallForest.callerContextType,
    //   () => {
    //     let { accountUpdate } = zkAppProver.getData();
    //     return CallForest.computeCallerContext(accountUpdate);
    //   }
    // );
    // CallForest.addCallers([accountUpdate], callerContext);

    // connect the public input to the account update & child account updates we created
    if (DEBUG_PUBLIC_INPUT_CHECK) {
      Provable.asProver(() => {
        // TODO: print a nice diff string instead of the two objects
        // something like &nbsp;expect&nbsp; or &nbsp;json-diff&nbsp;, but web-compatible
        function diff(prover: any, input: any) {
          delete prover.id;
          delete prover.callDepth;
          delete input.id;
          delete input.callDepth;
          if (JSON.stringify(prover) !== JSON.stringify(input)) {
            console.log(
              'transaction:',
              ZkappCommand.toPretty(transaction)
            );
            console.log('index', index);
            console.log('inconsistent account updates:');
            console.log('update created by the prover:');
            console.log(prover);
            console.log('update created in transaction block:');
            console.log(input);
          }
        }
        function diffRecursive(
          prover: AccountUpdate,
          input: AccountUpdate
        ) {
          diff(prover.toPretty(), input.toPretty());
          let nChildren = input.children.accountUpdates.length;
          for (let i = 0; i < nChildren; i++) {
            let inputChild = input.children.accountUpdates[i];
            let child = prover.children.accountUpdates[i];
            if (!inputChild || !child) return;
            diffRecursive(child, inputChild);
          }
        }

        let {
          accountUpdate: inputUpdate,
          transaction,
          index,
        } = zkAppProver.getData();
        diffRecursive(accountUpdate, inputUpdate);
      });
    }
    checkPublicInput(publicInput, accountUpdate);

    // check the self accountUpdate right after calling the method
    // TODO: this needs to be done in a unified way for all account updates that are created
    assertPreconditionInvariants(accountUpdate);
    cleanPreconditionsCache(accountUpdate);
    assertStatePrecondition(this);
    return result;
  } finally {
    Mina.currentTransaction.leave(txId);
  }
} else if (!Mina.currentTransaction.has()) {
  // outside a transaction, just call the method, but check precondition invariants
  let result = method.apply(this, actualArgs);
  // check the self accountUpdate right after calling the method
  // TODO: this needs to be done in a unified way for all account updates that are created
  assertPreconditionInvariants(this.self);
  cleanPreconditionsCache(this.self);
  assertStatePrecondition(this);
  return result;
} else {
  // called smart contract at the top level, in a transaction!
  // => attach ours to the current list of account updates
  let accountUpdate = context.selfUpdate;
  Mina.currentTransaction()?.accountUpdates.push(accountUpdate);

  // first, clone to protect against the method modifying arguments!
  // TODO: double-check that this works on all possible inputs, e.g. CircuitValue, o1js primitives
  let clonedArgs = cloneCircuitValue(actualArgs);

  // we run this in a "memoization context" so that we can remember witnesses for reuse when proving
  let blindingValue = getBlindingValue();
  let memoContext = { memoized: [], currentIndex: 0, blindingValue };
```

```
      let memoId = memoizationContext.enter(memoContext);
      let result: any;
      try {
        result = method.apply(
          this,
          actualArgs.map((a, i) => {
            let arg = methodIntf.allArgs[i];
            if (arg.type === 'witness') {
              let type = methodIntf.witnessArgs[arg.index];
              return Provable.witness(type, () => a);
            }
            return a;
          })
        );
      } finally {
        memoizationContext.leave(memoId);
      }
      let { memoized } = memoContext;

      assertStatePrecondition(this);

      // connect our input + result with callData, so this method can be called
      let callDataFields = computeCallData(
        methodIntf,
        clonedArgs,
        result,
        blindingValue
      );
      accountUpdate.body.callData = Poseidon.hash(callDataFields);

      if (!Authorization.hasAny(accountUpdate)) {
        Authorization.setLazyProof(
          accountUpdate,
          {
            methodName: methodIntf.methodName,
            args: clonedArgs,
            // proofs actually don't have to be cloned
            previousProofs: getPreviousProofsForProver(
              actualArgs,
              methodIntf
            ),
            ZkappClass,
            memoized,
            blindingValue,
          },
          Mina.currentTransaction()!.accountUpdates
        );
      }
      return result;
    }
  } finally {
    smartContractContext.leave(id);
    }
  }

  // if we're here, this method was called inside _another_ smart contract method
  let parentAccountUpdate = insideContract.this.self;
  let methodCallDepth = insideContract.methodCallDepth;
  let innerContext: SmartContractContext = {
    this: this,
    methodCallDepth: methodCallDepth + 1,
    selfUpdate: selfAccountUpdate(this, methodName),
  };
  let id = smartContractContext.enter(innerContext);
  try {
    // if the call result is not undefined but there's no known returnType, the returnType was probably not
annotated properly,
    // so we have to explain to the user how to do that
    let { returnType } = methodIntf;
    let noReturnTypeError =
      &nbsp;To return a result from ${methodIntf.methodName}() inside another zkApp, you need to declare
the return type.\n&nbsp; +
      &nbsp;This can be done by annotating the type at the end of the function signature. For
example:\n\n&nbsp; +
      &nbsp;@method ${methodIntf.methodName}(): Field {\n&nbsp; +
      &nbsp;  // ...\n&nbsp; +
      &nbsp;}\n\n&nbsp; +
      &nbsp;Note: Only types built out of \&nbsp;Field\&nbsp; are valid return types. This includes o1js
primitive types and custom CircuitValues.&nbsp;;
    // if we're lucky, analyzeMethods was already run on the callee smart contract, and we can catch this
error early
    if (
      ZkappClass._methodMetadata?.[methodIntf.methodName]?.hasReturn &&
      returnType === undefined
    ) {
      throw Error(noReturnTypeError);
    }
    // we just reuse the blinding value of the caller for the callee
    let blindingValue = getBlindingValue();

    let runCalledContract = () => {
      let constantArgs = methodArgumentsToConstant(methodIntf, actualArgs);
      let constantBlindingValue = blindingValue.toConstant();
      let accountUpdate = this.self;
      accountUpdate.body.callDepth = parentAccountUpdate.body.callDepth + 1;
      accountUpdate.parent = parentAccountUpdate;

      let memoContext = {
        memoized: [],
        currentIndex: 0,
        blindingValue: constantBlindingValue,
      };
      let memoId = memoizationContext.enter(memoContext);
      let result: any;
    try {
      result = method.apply(this, constantArgs.map(cloneCircuitValue));
    } finally {
      memoizationContext.leave(memoId);
    }
    let { memoized } = memoContext;
    assertStatePrecondition(this);

    if (result !== undefined) {
      if (returnType === undefined) {
        throw Error(noReturnTypeError);
      } else {
        result = toConstant(returnType, result);
      }
    }

    // store inputs + result in callData
    let callDataFields = computeCallData(
      methodIntf,
      constantArgs,
      result,
      constantBlindingValue
    );
    accountUpdate.body.callData = hashConstant(callDataFields);

    if (!Authorization.hasAny(accountUpdate)) {
      Authorization.setLazyProof(
        accountUpdate,
        {
          methodName: methodIntf.methodName,
          args: constantArgs,
          previousProofs: getPreviousProofsForProver(
            constantArgs,
            methodIntf
          ),
          ZkappClass,
          memoized,
          blindingValue: constantBlindingValue,
        },
        Mina.currentTransaction()!.accountUpdates
      );
    }
    return { accountUpdate, result: result ?? null };
  };

  // we have to run the called contract inside a witness block, to not affect the caller's circuit
  // however, if this is a nested call -- the caller is already called by another contract --,
  // then we're already in a witness block, and shouldn't open another one
  let { accountUpdate, result } =
    methodCallDepth === 0
      ? AccountUpdate.witness<any>(
          returnType ?? provable(null),
            runCalledContract,
            { skipCheck: true }
          )
        : runCalledContract();

      // we're back in the _caller's_ circuit now, where we assert stuff about the method call

      // overwrite this.self with the witnessed update, so it's this one we access later in the caller method
      innerContext.selfUpdate = accountUpdate;

      // connect accountUpdate to our own. outside Provable.witness so compile knows the right structure
when hashing children
      accountUpdate.body.callDepth = parentAccountUpdate.body.callDepth + 1;
      accountUpdate.parent = parentAccountUpdate;
      // beware: we don't include the callee's children in the caller circuit
      // nothing is asserted about them -- it's the callee's task to check their children
      accountUpdate.children.callsType = { type: 'Witness' };
      parentAccountUpdate.children.accountUpdates.push(accountUpdate);

      // assert that we really called the right zkapp
      accountUpdate.body.publicKey.assertEquals(this.address);
      accountUpdate.body.tokenId.assertEquals(this.self.body.tokenId);

      // assert that the inputs & outputs we have match what the callee put on its callData
      let callDataFields = computeCallData(
        methodIntf,
        actualArgs,
        result,
        blindingValue
      );
      let callData = Poseidon.hash(callDataFields);
      accountUpdate.body.callData.assertEquals(callData);
      return result;
    } finally {
      smartContractContext.leave(id);
    }
  };
}

function checkPublicInput(
  { accountUpdate, calls }: ZkappPublicInput,
  self: AccountUpdate
) {
  let otherInput = self.toPublicInput();
  accountUpdate.assertEquals(otherInput.accountUpdate);
  calls.assertEquals(otherInput.calls);
}

/**
 * compute fields to be hashed as callData, in a way that the hash & circuit changes whenever
 * the method signature changes, i.e., the argument / return types represented as lists of field elements and
the methodName.
```

```
 * see https://github.com/o1-labs/o1js/issues/303#issuecomment-1196441140
 */
function computeCallData(
  methodIntf: MethodInterface,
  argumentValues: any[],
  returnValue: any,
  blindingValue: Field
) {
  let { returnType, methodName } = methodIntf;
  let args = methodArgumentTypesAndValues(methodIntf, argumentValues);
  let argSizesAndFields: Field[][] = args.map(({ type, value }) => [
    Field(type.sizeInFields()),
    ...type.toFields(value),
  ]);
  let totalArgSize = Field(
    args.map(({ type }) => type.sizeInFields()).reduce((s, t) => s + t, 0)
  );
  let totalArgFields = argSizesAndFields.flat();
  let returnSize = Field(returnType?.sizeInFields() ?? 0);
  let returnFields = returnType?.toFields(returnValue) ?? [];
  let methodNameFields = Encoding.stringToFields(methodName);
  return [
    // we have to encode the sizes of arguments / return value, so that fields can't accidentally shift
    // from one argument to another, or from arguments to the return value, or from the return value to the
method name
    totalArgSize,
    ...totalArgFields,
    returnSize,
    ...returnFields,
    // we don't have to encode the method name size because the blinding value is fixed to one field element,
    // so method name fields can't accidentally become the blinding value and vice versa
    ...methodNameFields,
    blindingValue,
  ];
}

class Callback<Result> extends GenericArgument {
  instance: SmartContract;
  methodIntf: MethodInterface & { returnType: Provable<Result> };
  args: any[];

  result?: Result;
  accountUpdate: AccountUpdate;

  static create<T extends SmartContract, K extends keyof T>(
    instance: T,
    methodName: K,
    args: T[K] extends (...args: infer A) => any ? A : never
  ) {
    let ZkappClass = instance.constructor as typeof SmartContract;
    let methodIntf_ = (ZkappClass._methods ?? []).find(
      (i) => i.methodName === methodName
    );
    if (methodIntf_ === undefined)
      throw Error(
        &nbsp;Callback: could not find method ${ZkappClass.name}.${String(
          methodName
        )}&nbsp;
      );
    let methodIntf = {
      ...methodIntf_,
      returnType: methodIntf_.returnType ?? provable(null),
    };

    // call the callback, leveraging composability (if this is inside a smart contract method)
    // to prove to the outer circuit that we called it
    let result = (instance[methodName] as Function)(...args);
    let accountUpdate = instance.self;

    let callback = new Callback<any>({
      instance,
      methodIntf,
      args,
      result,
      accountUpdate,
      isEmpty: false,
    });

    return callback;
  }

  private constructor(self: Callback<any>) {
    super();
    Object.assign(this, self);
  }
}

/**
 * The main zkapp class. To write a zkapp, extend this class as such:
 *
 * &nbsp;&nbsp;&nbsp;
 * class YourSmartContract extends SmartContract {
 *   // your smart contract code here
 * }
 * &nbsp;&nbsp;&nbsp;
 *
 */
class SmartContract {
  address: PublicKey;
  tokenId: Field;

  #executionState: ExecutionState | undefined;

  // here we store various metadata associated with a SmartContract subclass.
  // by initializing all of these to &nbsp;undefined&nbsp;, we ensure that
  // subclasses aren't sharing the same property with the base class and each other
  // FIXME: these are still shared between a subclass and its own subclasses, which means extending
SmartContracts is broken
  static _methods?: MethodInterface[];
  static _methodMetadata?: Record<
    string,
    {
      actions: number;
      rows: number;
      digest: string;
      hasReturn: boolean;
      gates: Gate[];
    }
  >; // keyed by method name
  static _provers?: Pickles.Prover[];
  static _maxProofsVerified?: 0 | 1 | 2;
  static _verificationKey?: { data: string; hash: Field };

  /**
   * Returns a Proof type that belongs to this {@link SmartContract}.
   */
  static Proof() {
    let Contract = this;
    return class extends Proof<ZkappPublicInput, Empty> {
      static publicInputType = ZkappPublicInput;
      static publicOutputType = Empty;
      static tag = () => Contract;
    };
  }

  constructor(address: PublicKey, tokenId?: Field) {
    this.address = address;
    this.tokenId = tokenId ?? TokenId.default;
    Object.defineProperty(this, 'reducer', {
      set(this, reducer: Reducer<any>) {
        ((this as any)._ ??= {}).reducer = reducer;
      },
      get(this) {
        return getReducer(this);
      },
    });
  }

  /**
   * Compile your smart contract.
   *
   * This generates both the prover functions, needed to create proofs for running &nbsp;@method&nbsp;s,
   * and the verification key, needed to deploy your zkApp.
   *
   * Although provers and verification key are returned by this method, they are also cached internally and
used when needed,
   * so you don't actually have to use the return value of this function.
   *
   * Under the hood, "compiling" means calling into the lower-level [Pickles and Kimchi libraries](https://o1-labs.github.io/proof-systems/kimchi/overview.html) to
   * create multiple prover & verifier indices (one for each smart contract method as part of a "step circuit" and one for the "wrap circuit" which recursively wraps
   * it so that proofs end up in the original finite field). These are fairly expensive operations, so **expect compiling to take at least 20 seconds**,
   * up to several minutes if your circuit is large or your hardware is not optimal for these operations.
   */
  static async compile({ cache = Cache.FileSystemDefault } = {}) {
    let methodIntfs = this._methods ?? [];
    let methods = methodIntfs.map(({ methodName }) => {
      return (
        publicInput: unknown,
        publicKey: PublicKey,
        tokenId: Field,
        ...args: unknown[]
      ) => {
        let instance = new this(publicKey, tokenId);
        (instance as any)[methodName](publicInput, ...args);
      };
    });
    // run methods once to get information that we need already at compile time
    let methodsMeta = this.analyzeMethods();
    let gates = methodIntfs.map((intf) => methodsMeta[intf.methodName].gates);
    let {
      verificationKey: verificationKey_,
      provers,
      verify,
    } = await compileProgram({
      publicInputType: ZkappPublicInput,
      publicOutputType: Empty,
      methodIntfs,
      methods,
      gates,
      proofSystemTag: this,
      cache,
    });
    let verificationKey = {
      data: verificationKey_.data,
      hash: Field(verificationKey_.hash),
    } satisfies VerificationKey;
    this._provers = provers;
    this._verificationKey = verificationKey;
    // TODO: instead of returning provers, return an artifact from which provers can be recovered
    return { verificationKey, provers, verify };
  }

  /**
   * Computes a hash of your smart contract, which will reliably change _whenever one of your method circuits changes_.
   * This digest is quick to compute. it is designed to help with deciding whether a contract should be re-
 compiled or
   * a cached verification key can be used.
   * @returns the digest, as a hex string
   */
  static digest() {
    // TODO: this should use the method digests in a deterministic order!
    let methodData = this.analyzeMethods();
    let hash = hashConstant(
      Object.values(methodData).map((d) => Field(BigInt('0x' + d.digest)))
    );
    return hash.toBigInt().toString(16);
  }

  /**
   * Deploys a {@link SmartContract}.
   *
   * &nbsp;&nbsp;&nbsp;ts
   * let tx = await Mina.transaction(sender, () => {
   *   AccountUpdate.fundNewAccount(sender);
   *   zkapp.deploy();
   * });
   * tx.sign([senderKey, zkAppKey]);
   * &nbsp;&nbsp;&nbsp;
   */
  deploy({
    verificationKey,
    zkappKey,
  }: {
    verificationKey?: { data: string; hash: Field | string };
    zkappKey?: PrivateKey;
  } = {}) {
    let accountUpdate = this.newSelf();
    verificationKey ??= (this.constructor as typeof SmartContract)
      ._verificationKey;
    if (verificationKey === undefined) {
      if (!Mina.getProofsEnabled()) {
        let [, data, hash] = Pickles.dummyVerificationKey();
        verificationKey = { data, hash: Field(hash) };
      } else {
        throw Error(
          &nbsp;\&nbsp;${this.constructor.name}.deploy()\&nbsp; was called but no verification key was
 found.\n&nbsp; +
            &nbsp;Try calling \&nbsp;await ${this.constructor.name}.compile()\&nbsp; first, this will cache the
 verification key in the background.&nbsp;
        );
      }
    }
    let { hash: hash_, data } = verificationKey;
    let hash = Field.from(hash_);
    accountUpdate.account.verificationKey.set({ hash, data });
    accountUpdate.account.permissions.set(Permissions.default());
```

```
    accountUpdate.sign(zkappKey);
    AccountUpdate.attachToTransaction(accountUpdate);

    // init if this account is not yet deployed or has no verification key on it
    let shouldInit =
      !Mina.hasAccount(this.address) ||
      Mina.getAccount(this.address).zkapp?.verificationKey === undefined;
    if (!shouldInit) return;
    else this.init();
    let initUpdate = this.self;
    // switch back to the deploy account update so the user can make modifications to it
    this.#executionState = {
      transactionId: this.#executionState!.transactionId,
      accountUpdate,
    };
    // check if the entire state was overwritten, show a warning if not
    let isFirstRun = Mina.currentTransaction()?.numberOfRuns === 0;
    if (!isFirstRun) return;
    Provable.asProver(() => {
      if (
        initUpdate.update.appState.some(({ isSome }) => !isSome.toBoolean())
      ) {
        console.warn(
          &nbsp;WARNING: the \&nbsp;init()\&nbsp; method was called without overwriting the entire state.
This means that your zkApp will lack
the \&nbsp;provedState === true\&nbsp; status which certifies that the current state was verifiably produced
by proofs (and not arbitrarily set by the zkApp developer).
To make sure the entire state is reset, consider adding this line to the beginning of your \&nbsp;init()\&nbsp;
method:
super.init();
&nbsp;
        );
      }
    });
  }
  // TODO make this a @method and create a proof during &nbsp;zk deploy&nbsp; (+ add mechanism to
skip this)
  /**
   * &nbsp;SmartContract.init()&nbsp; will be called only when a {@link SmartContract} will be first deployed,
not for redeployment.
   * This method can be overridden as follows
   * &nbsp;&nbsp;&nbsp;
   * class MyContract extends SmartContract {
   *  init() {
   *    super.init();
   *    this.account.permissions.set(...);
   *    this.x.set(Field(1));
   *  }
   * }
   * &nbsp;&nbsp;&nbsp;
   */
  init() {
    // let accountUpdate = this.newSelf(); // this would emulate the behaviour of init() being a @method
    this.account.provedState.assertEquals(Bool(false));
    let accountUpdate = this.self;
    for (let i = 0; i < ZkappStateLength; i++) {
      AccountUpdate.setValue(accountUpdate.body.update.appState[i], Field(0));
    }
    AccountUpdate.attachToTransaction(accountUpdate);
  }

  /**
   * Use this command if the account update created by this SmartContract should be signed by the account
owner,
   * instead of authorized with a proof.
   *
   * Note that the smart contract's {@link Permissions} determine which updates have to be (can be)
authorized by a signature.
   *
   * If you only want to avoid creating proofs for quicker testing, we advise you to
   * use &nbsp;LocalBlockchain({ proofsEnabled: false })&nbsp; instead of &nbsp;requireSignature()&nbsp;.
Setting
   * &nbsp;proofsEnabled&nbsp; to &nbsp;false&nbsp; allows you to test your transactions with the same
authorization flow as in production,
   * with the only difference being that quick mock proofs are filled in instead of real proofs.
   */
  requireSignature() {
    this.self.requireSignature();
  }
  /**
   * @deprecated &nbsp;this.sign()&nbsp; is deprecated in favor of &nbsp;this.requireSignature()&nbsp;
   */
  sign(zkappKey?: PrivateKey) {
    this.self.sign(zkappKey);
  }
  /**
   * Use this command if the account update created by this SmartContract should have no authorization on
it,
   * instead of being authorized with a proof.
   *
   * WARNING: This is a method that should rarely be useful. If you want to disable proofs for quicker testing,
take a look
   * at &nbsp;LocalBlockchain({ proofsEnabled: false })&nbsp;, which causes mock proofs to be created and
doesn't require changing the
   * authorization flow.
   */
  skipAuthorization() {
    Authorization.setLazyNone(this.self);
  }

  /**
   * Returns the current {@link AccountUpdate} associated to this {@link SmartContract}.
   */
  get self(): AccountUpdate {
    let inTransaction = Mina.currentTransaction.has();
    let inSmartContract = smartContractContext.get();
    if (!inTransaction && !inSmartContract) {
      // TODO: it's inefficient to return a fresh account update everytime, would be better to return a constant
"non-writable" account update,
      // or even expose the .get() methods independently of any account update (they don't need one)
      return selfAccountUpdate(this);
    }
    let transactionId = inTransaction ? Mina.currentTransaction.id() : NaN;
    // running a method changes which is the "current account update" of this smart contract
    // this logic also implies that when calling &nbsp;this.self&nbsp; inside a method on &nbsp;this&nbsp;, it
will always
    // return the same account update uniquely associated with that method call.
    // it won't create new updates and add them to a transaction implicitly
    if (inSmartContract && inSmartContract.this === this) {
      let accountUpdate = inSmartContract.selfUpdate;
      this.#executionState = { accountUpdate, transactionId };
      return accountUpdate;
    }
    let executionState = this.#executionState;
    if (
      executionState !== undefined &&
      executionState.transactionId === transactionId
    ) {
      return executionState.accountUpdate;
    }
    // if in a transaction, but outside a @method call, we implicitly create an account update
    // which is stable during the current transaction -- as long as it doesn't get overridden by a method call
    let accountUpdate = selfAccountUpdate(this);
    this.#executionState = { transactionId, accountUpdate };
    return accountUpdate;
  }
  // same as this.self, but explicitly creates a _new_ account update
  /**
   * Same as &nbsp;SmartContract.self&nbsp; but explicitly creates a new {@link AccountUpdate}.
   */
  newSelf(): AccountUpdate {
    let inTransaction = Mina.currentTransaction.has();
    let transactionId = inTransaction ? Mina.currentTransaction.id() : NaN;
    let accountUpdate = selfAccountUpdate(this);
    this.#executionState = { transactionId, accountUpdate };
    return accountUpdate;
  }

  #_senderState: { sender: PublicKey; transactionId: number };

  /**
   * The public key of the current transaction's sender account.
   *
   * Throws an error if not inside a transaction, or the sender wasn't passed in.
   *
   * **Warning**: The fact that this public key equals the current sender is not part of the proof.
   * A malicious prover could use any other public key without affecting the validity of the proof.
   */
  get sender(): PublicKey {
    // TODO this logic now has some overlap with this.self, we should combine them somehow
    // (but with care since the logic in this.self is a bit more complicated)
    if (!Mina.currentTransaction.has()) {
      throw Error(
        &nbsp;this.sender is not available outside a transaction. Make sure you only use it within
\&nbsp;Mina.transaction\&nbsp; blocks or smart contract methods.&nbsp;
      );
    }
    let transactionId = Mina.currentTransaction.id();
    if (this.#_senderState?.transactionId === transactionId) {
      return this.#_senderState.sender;
    } else {
      let sender = Provable.witness(PublicKey, () => Mina.sender());
      this.#_senderState = { transactionId, sender };
      return sender;
    }
  }

  /**
   * Current account of the {@link SmartContract}.
   */
  get account() {
    return this.self.account;
  }
  /**
   * Current network state of the {@link SmartContract}.
   */
  get network() {
    return this.self.network;
  }
  /**
   * Current global slot on the network. This is the slot at which this transaction is included in a block. Since
we cannot know this value
   * at the time of transaction construction, this only has the &nbsp;assertBetween()&nbsp; method but no
&nbsp;get()&nbsp; (impossible to implement)
   * or &nbsp;assertEquals()&nbsp; (confusing, because the developer can't know the exact slot at which this
will be included either)
   */
  get currentSlot() {
    return this.self.currentSlot;
  }
  /**
   * Token of the {@link SmartContract}.
   */
  get token() {
    return this.self.token();
  }

  /**
```

```
   * Approve an account update or callback. This will include the account update in the zkApp's public input,
   * which means it allows you to read and use its content in a proof, make assertions about it, and modify it.
   *
   * If this is called with a callback as the first parameter, it will first extract the account update produced by
that callback.
   * The extracted account update is returned.
   *
   * &nbsp;&nbsp;&nbsp;ts
   * \@method myApprovingMethod(callback: Callback) {
   *   let approvedUpdate = this.approve(callback);
   * }
   * &nbsp;&nbsp;&nbsp;
   *
   * Under the hood, "approving" just means that the account update is made a child of the zkApp in the
   * tree of account updates that forms the transaction.
   * The second parameter &nbsp;layout&nbsp; allows you to also make assertions about the approved
update's _own_ children,
   * by specifying a certain expected layout of children. See {@link AccountUpdate.Layout}.
   *
   * @param updateOrCallback
   * @param layout
   * @returns The account update that was approved (needed when passing in a Callback)
   */
  approve(
    updateOrCallback: AccountUpdate | Callback<any>,
    layout?: AccountUpdatesLayout
  ) {
    let accountUpdate =
      updateOrCallback instanceof AccountUpdate
        ? updateOrCallback
        : Provable.witness(AccountUpdate, () => updateOrCallback.accountUpdate);
    this.self.approve(accountUpdate, layout);
    return accountUpdate;
  }

  send(args: {
    to: PublicKey | AccountUpdate | SmartContract;
    amount: number | bigint | UInt64;
  }) {
    return this.self.send(args);
  }

  /**
   * @deprecated use &nbsp;this.account.tokenSymbol&nbsp;
   */
  get tokenSymbol() {
    return this.self.tokenSymbol;
  }
  /**
   * Balance of this {@link SmartContract}.
   */
  get balance() {
    return this.self.balance;
  }
  /**
   * A list of event types that can be emitted using this.emitEvent()&nbsp;.
   */
  events: { [key: string]: FlexibleProvablePure<any> } = {};

  // TODO: not able to type event such that it is inferred correctly so far
  /**
   * Emits an event. Events will be emitted as a part of the transaction and can be collected by archive
nodes.
   */
  emitEvent<K extends keyof this['events']>(type: K, event: any) {
    let accountUpdate = this.self;
    let eventTypes: (keyof this['events'])[] = Object.keys(this.events);
    if (eventTypes.length === 0)
      throw Error(
        'emitEvent: You are trying to emit an event without having declared the types of your events.\n' +
          &nbsp;Make sure to add a property \&nbsp;events\&nbsp; on ${this.constructor.name}, for example:
\n&nbsp; +
          &nbsp;class ${this.constructor.name} extends SmartContract {\n&nbsp; +
          &nbsp;  events = { 'my-event': Field }\n&nbsp; +
          &nbsp;}&nbsp;
      );
    let eventNumber = eventTypes.sort().indexOf(type as string);
    if (eventNumber === -1)
      throw Error(
        &nbsp;emitEvent: Unknown event type "${
          type as string
        }". The declared event types are: ${eventTypes.join(', ')}.&nbsp;
      );
    let eventType = (this.events as this['events'])[type];
    let eventFields: Field[];
    if (eventTypes.length === 1) {
      // if there is just one event type, just store it directly as field elements
      eventFields = eventType.toFields(event);
    } else {
      // if there is more than one event type, also store its index, like in an enum, to identify the type later
      eventFields = [Field(eventNumber), ...eventType.toFields(event)];
    }
    accountUpdate.body.events = Events.pushEvent(
      accountUpdate.body.events,
      eventFields
    );
  }

  /**
   * Asynchronously fetches events emitted by this {@link SmartContract} and returns an array of events with
their corresponding types.
   * @async
   * @param [start=UInt32.from(0)] - The start height of the events to fetch.
   * @param end - The end height of the events to fetch. If not provided, fetches events up to the latest
```

```
height.
   * @returns A promise that resolves to an array of objects, each containing the event type and event data
for the specified range.
   * @throws If there is an error fetching events from the Mina network.
   * @example
   * const startHeight = UInt32.from(1000);
   * const endHeight = UInt32.from(2000);
   * const events = await myZkapp.fetchEvents(startHeight, endHeight);
   * console.log(events);
   */
  async fetchEvents(
    start: UInt32 = UInt32.from(0),
    end?: UInt32
  ): Promise<
    {
      type: string;
      event: {
        data: ProvablePure<any>;
        transactionInfo: {
          transactionHash: string;
          transactionStatus: string;
          transactionMemo: string;
        };
      };
      blockHeight: UInt32;
      blockHash: string;
      parentBlockHash: string;
      globalSlot: UInt32;
      chainStatus: string;
    }[]
  > {
    // filters all elements so that they are within the given range
    // only returns { type: "", event: [] } in a flat format
    let events = (
      await Mina.fetchEvents(this.address, this.self.body.tokenId, {
        from: start,
        to: end,
      })
    )
      .filter((eventData) => {
        let height = UInt32.from(eventData.blockHeight);
        return end === undefined
          ? start.lessThanOrEqual(height).toBoolean()
          : start.lessThanOrEqual(height).toBoolean() &&
              height.lessThanOrEqual(end).toBoolean();
      })
      .map((event) => {
        return event.events.map((eventData) => {
          let { events, ...rest } = event;
          return {
            ...rest,
            event: eventData,
          };
        });
      })
      .flat();

    // used to match field values back to their original type
    let sortedEventTypes = Object.keys(this.events).sort();

    return events.map((eventData) => {
      // if there is only one event type, the event structure has no index and can directly be matched to the
    event type
      if (sortedEventTypes.length === 1) {
        let type = sortedEventTypes[0];
        let event = this.events[type].fromFields(
          eventData.event.data.map((f: string) => Field(f))
        );
        return {
          ...eventData,
          type,
          event: {
            data: event,
            transactionInfo: {
              transactionHash: eventData.event.transactionInfo.hash,
              transactionStatus: eventData.event.transactionInfo.status,
              transactionMemo: eventData.event.transactionInfo.memo,
            },
          },
        };
      } else {
        // if there are multiple events we have to use the index event[0] to find the exact event type
        let eventObjectIndex = Number(eventData.event.data[0]);
        let type = sortedEventTypes[eventObjectIndex];
        // all other elements of the array are values used to construct the original object, we can drop the first
    value since its just an index
        let eventProps = eventData.event.data.slice(1);
        let event = this.events[type].fromFields(
          eventProps.map((f: string) => Field(f))
        );
        return {
          ...eventData,
          type,
          event: {
            data: event,
            transactionInfo: {
              transactionHash: eventData.event.transactionInfo.hash,
              transactionStatus: eventData.event.transactionInfo.status,
              transactionMemo: eventData.event.transactionInfo.memo,
            },
          },
        };
      }
    });
```

```
}

static runOutsideCircuit(run: () => void) {
  if (Mina.currentTransaction()?.isFinalRunOutsideCircuit || inProver())
    Provable.asProver(run);
}

// TODO: this could also be used to quickly perform any invariant checks on account updates construction
/**
  * This function is run internally before compiling a smart contract, to collect metadata about what each of
your
  * smart contract methods does.
  *
  * For external usage, this function can be handy because calling it involves running all methods in the
same "mode" as &nbsp;compile()&nbsp; does,
  * so it serves as a quick-to-run check for whether your contract can be compiled without errors, which can
greatly speed up iterating.
  *
  * &nbsp;analyzeMethods()&nbsp; will also return the number of &nbsp;rows&nbsp; of each of your method
circuits (i.e., the number of constraints in the underlying proof system),
  * which is a good indicator for circuit size and the time it will take to create proofs.
  * To inspect the created circuit in detail, you can look at the returned &nbsp;gates&nbsp;.
  *
  * Note: If this function was already called before, it will short-circuit and just return the metadata collected
the first time.
  *
  * @returns an object, keyed by method name, each entry containing:
  *  - &nbsp;rows&nbsp; the size of the constraint system created by this method
  *  - &nbsp;digest&nbsp; a digest of the method circuit
  *  - &nbsp;hasReturn&nbsp; a boolean indicating whether the method returns a value
  *  - &nbsp;actions&nbsp; the number of actions the method dispatches
  *  - &nbsp;gates&nbsp; the constraint system, represented as an array of gates
  */
static analyzeMethods() {
  let ZkappClass = this as typeof SmartContract;
  let methodMetadata = (ZkappClass._methodMetadata ??= {});
  let methodIntfs = ZkappClass._methods ?? [];
  if (
    !methodIntfs.every((m) => m.methodName in methodMetadata) &&
    !inAnalyze()
  ) {
    if (snarkContext.get().inRunAndCheck) {
      let err = new Error(
        'Can not analyze methods inside Provable.runAndCheck, because this creates a circuit nested in
another circuit'
      );
      // EXCEPT if the code that calls this knows that it can first run &nbsp;analyzeMethods&nbsp; OUTSIDE
runAndCheck and try again
      (err as any).bootstrap = () => ZkappClass.analyzeMethods();
      throw err;
    }
    let id: number;
    let insideSmartContract = !!smartContractContext.get();
    if (insideSmartContract) id = smartContractContext.enter(null);
    try {
      for (let methodIntf of methodIntfs) {
        let accountUpdate: AccountUpdate;
        let { rows, digest, result, gates } = analyzeMethod(
          ZkappPublicInput,
          methodIntf,
          (publicInput, publicKey, tokenId, ...args) => {
            let instance: SmartContract = new ZkappClass(publicKey, tokenId);
            let result = (instance as any)[methodIntf.methodName](
              publicInput,
              ...args
            );
            accountUpdate = instance.#executionState!.accountUpdate;
            return result;
          }
        );
        methodMetadata[methodIntf.methodName] = {
          actions: accountUpdate!.body.actions.data.length,
          rows,
          digest,
          hasReturn: result !== undefined,
          gates,
        };
      }
    } finally {
      if (insideSmartContract) smartContractContext.leave(id!);
    }
  }
  return methodMetadata;
 }

 /**
  * @deprecated use &nbsp;this.account.<field>.set()&nbsp;
  */
 setValue<T>(maybeValue: SetOrKeep<T>, value: T) {
  AccountUpdate.setValue(maybeValue, value);
 }

 /**
  * @deprecated use &nbsp;this.account.permissions.set()&nbsp;
  */
 setPermissions(permissions: Permissions) {
  this.self.account.permissions.set(permissions);
 }
}

type Reducer<Action> = {
 actionType: FlexibleProvablePure<Action>;
};
```

```
type ReducerReturn<Action> = {
  /**
   * Dispatches an {@link Action}. Similar to normal {@link Event}s,
   * {@link Action}s can be stored by archive nodes and later reduced within a {@link SmartContract} method
   * to change the state of the contract accordingly
   *
   * &nbsp;&nbsp;&nbsp;ts
   * this.reducer.dispatch(Field(1)); // emits one action
   * &nbsp;&nbsp;&nbsp;
   *
   * */
  dispatch(action: Action): void;
  /**
   * Reduces a list of {@link Action}s, similar to &nbsp;Array.reduce()&nbsp;.
   *
   * &nbsp;&nbsp;&nbsp;ts
   *  let pendingActions = this.reducer.getActions({
   *    fromActionState: actionState,
   *  });
   *
   *  let { state: newState, actionState: newActionState } =
   *  this.reducer.reduce(
   *     pendingActions,
   *     Field,
   *     (state: Field, _action: Field) => {
   *       return state.add(1);
   *     },
   *     { state: initialState, actionState: initialActionState  }
   *   );
   * &nbsp;&nbsp;&nbsp;
   *
   */
  reduce<State>(
    actions: Action[][],
    stateType: Provable<State>,
    reduce: (state: State, action: Action) => State,
    initial: { state: State; actionState: Field },
    options?: {
      maxTransactionsWithActions?: number;
      skipActionStatePrecondition?: boolean;
    }
  ): { state: State; actionState: Field };
  /**
   * Perform circuit logic for every {@link Action} in the list.
   *
   * This is a wrapper around {@link reduce} for when you don't need &nbsp;state&nbsp;.
   * Accepts the &nbsp;fromActionState&nbsp; and returns the updated action state.
   */
  forEach(
    actions: Action[][],
    reduce: (action: Action) => void,
    fromActionState: Field,
  options?: {
    maxTransactionsWithActions?: number;
    skipActionStatePrecondition?: boolean;
  }
 ): Field;
 /**
  * Fetches the list of previously emitted {@link Action}s by this {@link SmartContract}.
  * &nbsp;&nbsp;&nbsp;ts
  * let pendingActions = this.reducer.getActions({
  *    fromActionState: actionState,
  * });
  * &nbsp;&nbsp;&nbsp;
  */
 getActions({
  fromActionState,
  endActionState,
 }?: {
  fromActionState?: Field;
  endActionState?: Field;
 }): Action[][];
 /**
  * Fetches the list of previously emitted {@link Action}s by zkapp {@link SmartContract}.
  * &nbsp;&nbsp;&nbsp;ts
  * let pendingActions = await zkapp.reducer.fetchActions({
  *    fromActionState: actionState,
  * });
  * &nbsp;&nbsp;&nbsp;
  */
 fetchActions({
  fromActionState,
  endActionState,
 }: {
  fromActionState?: Field;
  endActionState?: Field;
 }): Promise<Action[][]>;
};

function getReducer<A>(contract: SmartContract): ReducerReturn<A> {
 let reducer: Reducer<A> = ((contract as any)._ ??= {}).reducer;
 if (reducer === undefined)
  throw Error(
    'You are trying to use a reducer without having declared its type.\n' +
     &nbsp;Make sure to add a property \&nbsp;reducer\&nbsp; on ${contract.constructor.name}, for
example:
class ${contract.constructor.name} extends SmartContract {
 reducer = { actionType: Field };
}&nbsp;
  );
 return {
  dispatch(action: A) {
    let accountUpdate = contract.self;
    let eventFields = reducer.actionType.toFields(action);
    accountUpdate.body.actions = Actions.pushEvent(
      accountUpdate.body.actions,
      eventFields
    );
  },

  reduce<S>(
    actionLists: A[][],
    stateType: Provable<S>,
    reduce: (state: S, action: A) => S,
    { state, actionState }: { state: S; actionState: Field },
    {
      maxTransactionsWithActions = 32,
      skipActionStatePrecondition = false,
    } = {}
  ): { state: S; actionState: Field } {
    if (actionLists.length > maxTransactionsWithActions) {
      throw Error(
        &nbsp;reducer.reduce: Exceeded the maximum number of lists of actions,
${maxTransactionsWithActions}.
Use the optional \&nbsp;maxTransactionsWithActions\&nbsp; argument to increase this number.&nbsp;
      );
    }
    let methodData = (
      contract.constructor as typeof SmartContract
    ).analyzeMethods();
    let possibleActionsPerTransaction = [
      ...new Set(Object.values(methodData).map((o) => o.actions)).add(0),
    ].sort((x, y) => x - y);

    let possibleActionTypes = possibleActionsPerTransaction.map((n) =>
      Provable.Array(reducer.actionType, n)
    );
    for (let i = 0; i < maxTransactionsWithActions; i++) {
      let actions = i < actionLists.length ? actionLists[i] : [];
      let length = actions.length;
      let lengths = possibleActionsPerTransaction.map((n) =>
        Provable.witness(Bool, () => Bool(length === n))
      );
      // create dummy actions for the other possible action lengths,
      // -> because this needs to be a statically-sized computation we have to operate on all of them
      let actionss = possibleActionsPerTransaction.map((n, i) => {
        let type = possibleActionTypes[i];
        return Provable.witness(type, () =>
          length === n ? actions : emptyValue(type)
        );
      });
      // for each action length, compute the events hash and then pick the actual one
      let eventsHashes = actionss.map((actions) => {
        let events = actions.map((a) => reducer.actionType.toFields(a));
        return Actions.hash(events);
      });
      let eventsHash = Provable.switch(lengths, Field, eventsHashes);
      let newActionsHash = Actions.updateSequenceState(
        actionState,
        eventsHash
      );
      let isEmpty = lengths[0];
      // update state hash, if this is not an empty action
      actionState = Provable.if(isEmpty, actionState, newActionsHash);
      // also, for each action length, compute the new state and then pick the actual one
      let newStates = actionss.map((actions) => {
        // we generate a new witness for the state so that this doesn't break if &nbsp;apply&nbsp; modifies the
state
        let newState = Provable.witness(stateType, () => state);
        Provable.assertEqual(stateType, newState, state);
        // apply actions in reverse order since that's how they were stored at dispatch
        [...actions].reverse().forEach((action) => {
          newState = reduce(newState, action);
        });
        return newState;
      });
      // update state
      state = Provable.switch(lengths, stateType, newStates);
    }
    if (!skipActionStatePrecondition) {
      contract.account.actionState.assertEquals(actionState);
    }
    return { state, actionState };
  },

  forEach(
    actionLists: A[][],
    callback: (action: A) => void,
    fromActionState: Field,
    config
  ): Field {
    const stateType = provable(null);
    let { actionState } = this.reduce(
      actionLists,
      stateType,
      (_, action) => {
        callback(action);
        return null;
      },
      { state: null, actionState: fromActionState },
      config
    );
    return actionState;
  },

  getActions(config?: {
    fromActionState?: Field;
    endActionState?: Field;
  }): A[][] {
    let actionsForAccount: A[][] = [];
    Provable.asProver(() => {
      let actions = Mina.getActions(
        contract.address,
        config,
        contract.self.tokenId
      );
      actionsForAccount = actions.map((event) =>
        // putting our string-Fields back into the original action type
        event.actions.map((action) =>
          (reducer.actionType as ProvablePure<A>).fromFields(
            action.map(Field)
          )
        )
      );
    });
    return actionsForAccount;
  },
  async fetchActions(config?: {
    fromActionState?: Field;
    endActionState?: Field;
  }): Promise<A[][]> {
    let result = await Mina.fetchActions(
      contract.address,
      config,
      contract.self.tokenId
    );
    if ('error' in result) {
      throw Error(JSON.stringify(result));
    }
    return result.map((event) =>
      // putting our string-Fields back into the original action type
      event.actions.map((action) =>
        (reducer.actionType as ProvablePure<A>).fromFields(action.map(Field))
      )
    );
  },
};
}

class VerificationKey extends Struct({
...provable({ data: String, hash: Field }),
toJSON({ data }: { data: string }) {
  return data;
},
}) {}

function selfAccountUpdate(zkapp: SmartContract, methodName?: string) {
let body = Body.keepAll(zkapp.address, zkapp.tokenId);
let update = new (AccountUpdate as any)(body, {}, true) as AccountUpdate;
update.label = methodName
```

```
    ? &nbsp;${zkapp.constructor.name}.${methodName}()&nbsp;
    : &nbsp;${zkapp.constructor.name}, no method&nbsp;;
  return update;
}

// per-smart-contract context for transaction construction
type ExecutionState = {
  transactionId: number;
  accountUpdate: AccountUpdate;
};

type DeployArgs =
  | {
      verificationKey?: { data: string; hash: string | Field };
      zkappKey?: PrivateKey;
    }
  | undefined;

function Account(address: PublicKey, tokenId?: Field) {
  if (smartContractContext.get()) {
    return AccountUpdate.create(address, tokenId).account;
  } else {
    return AccountUpdate.defaultAccountUpdate(address, tokenId).account;
  }
}

// alternative API which can replace decorators, works in pure JS

/**
 * &nbsp;declareMethods&nbsp; can be used in place of the &nbsp;@method&nbsp; decorator
 * to declare SmartContract methods along with their list of arguments.
 * It should be placed _after_ the class declaration.
 * Here is an example of declaring a method &nbsp;update&nbsp;, which takes a single argument of type
&nbsp;Field&nbsp;:
 * &nbsp;&nbsp;&nbsp;ts
 * class MyContract extends SmartContract {
 *   // ...
 *   update(x: Field) {
 *     // ...
 *   }
 * }
 * declareMethods(MyContract, { update: [Field] }); // &nbsp;[Field]&nbsp; is the list of arguments!
 * &nbsp;&nbsp;&nbsp;
 * Note that a method of the same name must still be defined on the class, just without the decorator.
 */
function declareMethods<T extends typeof SmartContract>(
  SmartContract: T,
  methodArguments: Record<string, Provable<unknown>[]>
) {
  for (let key in methodArguments) {
    let argumentTypes = methodArguments[key];
    let target = SmartContract.prototype;
    Reflect.metadata('design:paramtypes', argumentTypes)(target, key);
    let descriptor = Object.getOwnPropertyDescriptor(target, key)!;
    method(SmartContract.prototype, key as any, descriptor);
    Object.defineProperty(target, key, descriptor);
  }
}

const Reducer: (<
  T extends FlexibleProvablePure<any>,
  A extends InferProvable<T> = InferProvable<T>
>(reducer: {
  actionType: T;
}) => ReducerReturn<A>) & {
  initialActionState: Field;
} = Object.defineProperty(
  function (reducer: any) {
    // we lie about the return value here, and instead overwrite this.reducer with
    // a getter, so we can get access to &nbsp;this&nbsp; inside functions on this.reducer (see constructor)
    return reducer;
  },
  'initialActionState',
  { get: Actions.emptyActionState }
) as any;

/**
 * this is useful to debug a very common error: when the consistency check between
 * -) the account update that went into the public input, and
 * -) the account update constructed by the prover
 * fails.
 * toggling this will print the two account updates in addition to the unhelpful failed assertion error when the
check fails,
 * making it easier to see where the problem lies.
 * TODO refine this into a good error message that's always used, not just for debugging
 * TODO find or write library that can print nice JS object diffs
 */
const DEBUG_PUBLIC_INPUT_CHECK = false;
```

</file>

<file>

## path: /src/mina-signer/MinaSigner.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/MinaSigner.ts

```
import { PrivateKey, PublicKey } from '../provable/curve-bigint.js';
import * as Json from './src/TSTypes.js';
import type { SignedLegacy, Signed, Network } from './src/TSTypes.js';

import {
  isPayment,
  isSignedDelegation,
  isSignedPayment,
  isSignedString,
  isSignedZkappCommand,
  isStakeDelegation,
  isZkappCommand,
} from './src/Utils.js';
import * as TransactionJson from '../bindings/mina-transaction/gen/transaction-json.js';
import { ZkappCommand } from '../bindings/mina-transaction/gen/transaction-bigint.js';
import {
  signZkappCommand,
  verifyZkappCommandSignature,
} from './src/sign-zkapp-command.js';
import {
  signPayment,
  signStakeDelegation,
  signString,
  verifyPayment,
  verifyStakeDelegation,
  verifyStringSignature,
} from './src/sign-legacy.js';
import { hashPayment, hashStakeDelegation } from './src/transaction-hash.js';
import { Memo } from './src/memo.js';
import {
  publicKeyToHex,
  rosettaTransactionToSignedCommand,
} from './src/rosetta.js';
import { sign, Signature, verify } from './src/signature.js';
import { createNullifier } from './src/nullifier.js';

export { Client as default };

const defaultValidUntil = '4294967295';

class Client {
  private network: Network;

  constructor(options: { network: Network }) {
    if (!options?.network) {
      throw Error('Invalid Specified Network');
    }
    const specifiedNetwork = options.network.toLowerCase();
    if (specifiedNetwork !== 'mainnet' && specifiedNetwork !== 'testnet') {
      throw Error('Invalid Specified Network');
    }
    this.network = specifiedNetwork;
  }

  /**
   * Generates a public/private key pair
   *
   * @returns A Mina key pair
```

```
 */
genKeys(): Json.Keypair {
  let privateKey = PrivateKey.random();
  let publicKey = PrivateKey.toPublicKey(privateKey);
  return {
    privateKey: PrivateKey.toBase58(privateKey),
    publicKey: PublicKey.toBase58(publicKey),
  };
}

/**
 * Verifies if a key pair is valid by checking if the public key can be derived from
 * the private key and additionally checking if we can use the private key to
 * sign a transaction. If the key pair is invalid, an exception is thrown.
 *
 * @param keypair A key pair
 * @returns True if the &nbsp;keypair&nbsp; is a verifiable key pair, otherwise throw an exception
 */
verifyKeypair({ privateKey, publicKey }: Json.Keypair): boolean {
  let derivedPublicKey = PrivateKey.toPublicKey(
    PrivateKey.fromBase58(privateKey)
  );
  let originalPublicKey = PublicKey.fromBase58(publicKey);
  if (
    derivedPublicKey.x !== originalPublicKey.x ||
    derivedPublicKey.isOdd !== originalPublicKey.isOdd
  ) {
    throw Error('Public key not derivable from private key');
  }
  let dummy = ZkappCommand.toJSON(ZkappCommand.emptyValue());
  dummy.feePayer.body.publicKey = publicKey;
  dummy.memo = Memo.toBase58(Memo.emptyValue());
  let signed = signZkappCommand(dummy, privateKey, this.network);
  let ok = verifyZkappCommandSignature(signed, publicKey, this.network);
  if (!ok) throw Error('Could not sign a transaction with private key');
  return true;
}

/**
 * Derives the public key of the corresponding private key
 *
 * @param privateKey The private key used to get the corresponding public key
 * @returns A public key
 */
derivePublicKey(privateKeyBase58: Json.PrivateKey): Json.PublicKey {
  let privateKey = PrivateKey.fromBase58(privateKeyBase58);
  let publicKey = PrivateKey.toPublicKey(privateKey);
  return PublicKey.toBase58(publicKey);
}

/**
 * Signs an arbitrary list of field elements in a SNARK-compatible way.
 * The resulting signature can be verified in o1js as follows:
 * &nbsp;&nbsp;&nbsp;ts
 * // sign field elements with mina-signer
 * let signed = client.signFields(fields, privateKey);
 *
 * // read signature in o1js and verify
 * let signature = Signature.fromBase58(signed.signature);
 * let isValid: Bool = signature.verify(publicKey, fields.map(Field));
 * &nbsp;&nbsp;&nbsp;
 *
 * @param fields An arbitrary list of field elements
 * @param privateKey The private key used for signing
 * @returns The signed field elements
 */
signFields(fields: bigint[], privateKey: Json.PrivateKey): Signed<bigint[]> {
  let privateKey_ = PrivateKey.fromBase58(privateKey);
  let signature = sign({ fields }, privateKey_, 'testnet');
  return {
    signature: Signature.toBase58(signature),
    publicKey: PublicKey.toBase58(PrivateKey.toPublicKey(privateKey_)),
    data: fields,
  };
}

/**
 * Verifies a signature created by {@link signFields}.
 *
 * @param signedFields The signed field elements
 * @returns True if the &nbsp;signedFields&nbsp; contains a valid signature matching
 * the fields and publicKey.
 */
verifyFields({ data, signature, publicKey }: Signed<bigint[]>) {
  return verify(
    Signature.fromBase58(signature),
    { fields: data },
    PublicKey.fromBase58(publicKey),
    'testnet'
  );
}

/**
 * Signs an arbitrary message
 *
 * @param message An arbitrary string message to be signed
 * @param privateKey The private key used to sign the message
 * @returns A signed message
 */
signMessage(
  message: string,
  privateKey: Json.PrivateKey
): SignedLegacy<string> {
  let privateKey_ = PrivateKey.fromBase58(privateKey);
  let publicKey = PublicKey.toBase58(PrivateKey.toPublicKey(privateKey_));
  return {
    signature: signString(message, privateKey, this.network),
    publicKey,
    data: message,
  };
}

/**
 * Verifies a signature created by {@link signMessage}.
 *
 * @param signedMessage A signed message
 * @returns True if the &nbsp;signedMessage&nbsp; contains a valid signature matching
 * the message and publicKey.
 */
verifyMessage({ data, signature, publicKey }: SignedLegacy<string>): boolean {
  return verifyStringSignature(data, signature, publicKey, this.network);
}

/**
 * Signs a payment transaction using a private key.
 *
 * This type of transaction allows a user to transfer funds from one account
 * to another over the network.
 *
 * @param payment An object describing the payment
 * @param privateKey The private key used to sign the transaction
 * @returns A signed payment transaction
 */
signPayment(
  payment: Json.Payment,
  privateKey: Json.PrivateKey
): SignedLegacy<Json.Payment> {
  let { fee, to, from, nonce, validUntil, memo } = validCommon(payment);
  let amount = String(payment.amount);
  let signature = signPayment(
    {
      common: { fee, feePayer: from, nonce, validUntil, memo },
      body: { receiver: to, amount },
    },
    privateKey,
    this.network
  );
  return {
    signature,
    publicKey: from,
    data: { to, from, fee, amount, nonce, memo, validUntil },
  };
}

/**
 * Verifies a signed payment.
 *
 * @param signedPayment A signed payment transaction
 * @returns True if the &nbsp;signedPayment&nbsp; is a verifiable payment
 */
verifyPayment({
  data,
  signature,
  publicKey,
}: SignedLegacy<Json.Payment>): boolean {
  let { fee, to, from, nonce, validUntil, memo } = validCommon(data);
  let amount = validNonNegative(data.amount);
  return verifyPayment(
    {
      common: { fee, feePayer: from, nonce, validUntil, memo },
      body: { receiver: to, amount },
    },
    signature,
    publicKey,
    this.network
  );
}

/**
 * Signs a stake delegation transaction using a private key.
 *
 * This type of transaction allows a user to delegate their
 * funds from one account to another for use in staking. The
 * account that is delegated to is then considered as having these
 * funds when determining whether it can produce a block in a given slot.
 *
 * @param delegation An object describing the stake delegation
 * @param privateKey The private key used to sign the transaction
 * @returns A signed stake delegation
 */
signStakeDelegation(
  delegation: Json.StakeDelegation,
  privateKey: Json.PrivateKey
): SignedLegacy<Json.StakeDelegation> {
  let { fee, to, from, nonce, validUntil, memo } = validCommon(delegation);
  let signature = signStakeDelegation(
    {
      common: { fee, feePayer: from, nonce, validUntil, memo },
      body: { newDelegate: to },
    },
    privateKey,
    this.network
  );
  return {
    signature,
    publicKey: from,
    data: { to, from, fee, nonce, memo, validUntil },
  };
}

/**
 * Verifies a signed stake delegation.
 *
 * @param signedStakeDelegation A signed stake delegation
 * @returns True if the &nbsp;signedStakeDelegation&nbsp; is a verifiable stake delegation
 */
verifyStakeDelegation({
  data,
  signature,
  publicKey,
}: SignedLegacy<Json.StakeDelegation>): boolean {
  let { fee, to, from, nonce, validUntil, memo } = validCommon(data);
  return verifyStakeDelegation(
    {
      common: { fee, feePayer: from, nonce, validUntil, memo },
      body: { newDelegate: to },
    },
    signature,
    publicKey,
    this.network
  );
}

/**
 * Compute the hash of a signed payment.
 *
 * @param signedPayment A signed payment transaction
 * @returns A transaction hash
 */
hashPayment(
  { data, signature }: SignedLegacy<Json.Payment>,
  options?: { berkeley?: boolean }
): string {
  let { fee, to, from, nonce, validUntil, memo } = validCommon(data);
  let amount = validNonNegative(data.amount);
  return hashPayment(
    {
      signature,
      data: {
        common: { fee, feePayer: from, nonce, validUntil, memo },
        body: { receiver: to, amount },
      },
    },
    options
  );
}

/**
 * Compute the hash of a signed stake delegation.
 *
 * @param signedStakeDelegation A signed stake delegation
 * @returns A transaction hash
 */
hashStakeDelegation(
  { data, signature }: SignedLegacy<Json.StakeDelegation>,
  options?: { berkeley?: boolean }
): string {
  let { fee, to, from, nonce, validUntil, memo } = validCommon(data);
  return hashStakeDelegation(
    {
      signature,
      data: {
        common: { fee, feePayer: from, nonce, validUntil, memo },
        body: { newDelegate: to },
      },
    },
    options
  );
}

/**
 * Sign a zkapp command transaction using a private key.
 *
 * This type of transaction allows a user to update state on a given
 * Smart Contract running on Mina.
 *
 * @param zkappCommand An object representing a zkApp transaction
 * @param privateKey The fee payer private key
 * @returns Signed &nbsp;zkappCommand&nbsp;
 */
signZkappCommand(
  { feePayer: feePayer_, zkappCommand }: Json.ZkappCommand,
  privateKey: Json.PrivateKey
): Signed<Json.ZkappCommand> {
  let accountUpdates = zkappCommand.accountUpdates;
  let minimumFee = this.getAccountUpdateMinimumFee(accountUpdates);
  let feePayer = validFeePayer(feePayer_, minimumFee);
  let { fee, nonce, validUntil, feePayer: publicKey, memo } = feePayer;
  let command: TransactionJson.ZkappCommand = {
    feePayer: {
      body: { publicKey, fee, nonce, validUntil },
      authorization: '', // gets filled below
    },
    accountUpdates,
    memo: Memo.toBase58(Memo.fromString(memo)),
  };
  let signed = signZkappCommand(command, privateKey, this.network);
  let signature = signed.feePayer.authorization;
  return { signature, publicKey, data: { zkappCommand: signed, feePayer } };
}

/**
```

```
 * Verifies a signed zkApp transaction.
 *
 * @param signedZkappCommand A signed zkApp transaction
 * @returns True if the signature is valid
 */
verifyZkappCommand({
  data,
  publicKey,
  signature,
}: Signed<Json.ZkappCommand>): boolean {
  return (
    signature === data.zkappCommand.feePayer.authorization &&
    verifyZkappCommandSignature(data.zkappCommand, publicKey, this.network)
  );
}

/**
 * Converts a Rosetta signed transaction to a JSON string that is
 * compatible with GraphQL. The JSON string is a representation of
 * a &nbsp;Signed_command&nbsp; which is what our GraphQL expects.
 *
 * @param signedRosettaTxn A signed Rosetta transaction
 * @returns A string that represents the JSON conversion of a signed Rosetta transaction&nbsp;.
 */
signedRosettaTransactionToSignedCommand(signedRosettaTxn: string): string {
  let parsedTx = JSON.parse(signedRosettaTxn);
  let command = rosettaTransactionToSignedCommand(parsedTx);
  return JSON.stringify({ data: command });
}

/**
 * Return the hex-encoded format of a valid public key. This will throw an exception if
 * the key is invalid or the conversion fails.
 *
 * @param publicKey A valid public key
 * @returns A string that represents the hex encoding of a public key.
 */
publicKeyToRaw(publicKeyBase58: string): string {
  let publicKey = PublicKey.fromBase58(publicKeyBase58);
  return publicKeyToHex(publicKey);
}

/**
 * Signs an arbitrary payload using a private key. This function can sign strings,
 * payments, stake delegations, and zkapp commands. If the payload is unrecognized, an Error
 * is thrown.
 *
 * @param payload A signable payload
 * @param privateKey A private key
 * @returns A signed payload
 */
signTransaction<T extends Json.SignableData | Json.ZkappCommand>(
  payload: T,
  privateKey: Json.PrivateKey
): T extends Json.SignableData ? SignedLegacy<T> : Signed<T> {
  type Return = T extends Json.SignableData ? SignedLegacy<T> : Signed<T>;
  if (typeof payload === 'string') {
    return this.signMessage(payload, privateKey) as Return;
  }
  if (isPayment(payload)) {
    return this.signPayment(payload, privateKey) as Return;
  }
  if (isStakeDelegation(payload)) {
    return this.signStakeDelegation(payload, privateKey) as Return;
  }
  if (isZkappCommand(payload)) {
    return this.signZkappCommand(payload, privateKey) as Return;
  } else {
    throw Error(&nbsp;Expected signable payload, got '${payload}'.&nbsp;);
  }
}

/**
 * Verifies a signed payload. The payload can be a string, payment, stake delegation or zkApp transaction.
 * If the payload is unrecognized, an Error is thrown.
 *
 * @param signedPayload A signed payload
 * @returns True if the signature is valid
 */
verifyTransaction(
  signed: SignedLegacy<Json.SignableData> | Signed<Json.ZkappCommand>
): boolean {
  if (isSignedString(signed)) {
    return this.verifyMessage(signed);
  }
  if (isSignedPayment(signed)) {
    return this.verifyPayment(signed);
  }
  if (isSignedDelegation(signed)) {
    return this.verifyStakeDelegation(signed);
  }
  if (isSignedZkappCommand(signed)) {
    return this.verifyZkappCommand(signed);
  } else {
    throw Error(
      &nbsp;Expected signable payload, got '${JSON.stringify(signed.data)}'.&nbsp;
    );
  }
}

/**
 * Calculates the minimum fee of a zkapp command transaction. A fee for a zkapp command transaction is
 * the sum of all account updates plus the specified fee amount. If no fee is passed in, &nbsp;0.001&nbsp;
 * is used (according to the Mina spec) by default.
```

```
  * @param accountUpdates A list of account updates
  * @returns  The fee to be paid by the fee payer accountUpdate
  */
 getAccountUpdateMinimumFee(accountUpdates: TransactionJson.AccountUpdate[]) {
   return 0.001 * accountUpdates.length;
 }

 /**
  * Creates a nullifier
  *
  * @param message A unique message that belongs to a specific nullifier
  * @param privateKeyBase58 The private key used to create the nullifier
  * @returns A nullifier
  */
 createNullifier(
   message: bigint[],
   privateKeyBase58: Json.PrivateKey
 ): Json.Nullifier {
   let sk = PrivateKey.fromBase58(privateKeyBase58);
   return createNullifier(message, sk);
 }
}

function validNonNegative(n: number | string | bigint): string {
 let n0 = BigInt(n); // validates that string represents an integer; also throws runtime errors for nullish inputs
 if (n0 < 0) throw Error('input must be non-negative');
 return n0.toString();
}

function validCommon(common: Json.Common): Json.StrictCommon {
 let memo = Memo.toValidString(common.memo);
 return {
   to: common.to,
   from: common.from,
   fee: validNonNegative(common.fee),
   nonce: validNonNegative(common.nonce),
   memo,
   validUntil: validNonNegative(common.validUntil ?? defaultValidUntil),
 };
}

function validFeePayer(
 feePayer: Json.ZkappCommand['feePayer'],
 minimumFee: number
): Json.StrictFeePayer {
 if (feePayer.fee === undefined) throw Error('Missing fee in fee payer');
 let fee = validNonNegative(feePayer.fee);
 if (Number(fee) < minimumFee)
   throw Error(&nbsp;Fee must be greater than ${minimumFee}&nbsp;);
 return {
   feePayer: feePayer.feePayer,
   fee,
   nonce: validNonNegative(feePayer.nonce),
   memo: Memo.toValidString(feePayer.memo),
   validUntil:
     feePayer.validUntil === undefined || feePayer.validUntil === null
       ? null
       : validNonNegative(feePayer.validUntil),
 };
}
```

</file>

<file>

## path: /src/mina-signer/README.md

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/README.md

# Mina Signer

This is a NodeJS SDK that allows you to sign strings, payments, and delegations using Mina's key pairs for various specified networks.

# Install

```
yarn add mina-signer
# or with npm:
npm install --save mina-signer
```

# Usage

```
import Client from 'mina-signer';
const client = new Client({ network: 'mainnet' });

// Generate keys
let keypair = client.genKeys();

// Sign and verify message
let signed = client.signMessage('hello', keypair.privateKey);
if (client.verifyMessage(signed)) {
  console.log('Message was verified successfully');
}

// Sign and verify a payment
let signedPayment = client.signPayment(
  {
    to: keypair.publicKey,
    from: keypair.publicKey,
    amount: 1,
    fee: 1,
    nonce: 0,
  },
  keypair.privateKey
);
if (client.verifyPayment(signedPayment)) {
  console.log('Payment was verified successfully');
}

// Sign and verify a stake delegation
const signedDelegation = client.signStakeDelegation(
  {
    to: keypair.publicKey,
    from: keypair.publicKey,
    fee: '1',
    nonce: '0',
  },
  keypair.privateKey
);
if (client.verifyStakeDelegation(signedDelegation)) {
  console.log('Delegation was verified successfully');
}
```

</file>

<file>

## path: /src/mina-signer/build-cjs.sh

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/build-cjs.sh

```
#!/bin/bash
set -e

cp index.cjs dist/node/mina-signer/index.cjs
cp index.d.ts dist/node/mina-signer/index.d.ts

npx esbuild --bundle --minify dist/node/mina-signer/index.cjs --outfile=./dist/node/mina-signer/index.cjs --
format=cjs --target=es2021 --platform=node --allow-overwrite=true
```

</file>

<file>

## path: /src/mina-signer/build-web.sh

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/build-web.sh

```
#!/bin/bash
set -e

tsc -p ../../tsconfig.mina-signer-web.json
node moveWebFiles.js
npx esbuild --bundle --minify dist/tmp/mina-signer/MinaSigner.js --outfile=./dist/web/index.js --format=esm --
target=es2021
npx rimraf dist/tmp
```

</file>

<file>

## path: /src/mina-signer/index.cjs

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/index.cjs

```
// this file is a wrapper for supporting commonjs imports

let Client = require('./MinaSigner.js');

module.exports = Client.default;
```

</file>

<file>

## path: /src/mina-signer/index.d.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/index.d.ts

```
// this file is a wrapper for supporting types in both commonjs and esm projects

import Client from './MinaSigner.js';

export = Client;
```

</file>

<file>

## path: /src/mina-signer/jest.config.js

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/jest.config.js

```
export default {
  preset: 'ts-jest/presets/js-with-ts',
  testEnvironment: 'node',
  extensionsToTreatAsEsm: ['.ts'],
  transformIgnorePatterns: ['node_modules/', 'dist/node/'],
  globals: {
    'ts-jest': {
      useESM: true,
    },
  },
};
```

</file>

<file>

## path: /src/mina-signer/moveWebFiles.js

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/moveWebFiles.js

```
import glob from 'glob';
import { move } from 'fs-extra';

let webFiles = glob.sync('./dist/tmp/**/*.web.js');

await Promise.all(
  webFiles.map((file) =>
    move(file, file.replace('.web.js', '.js'), { overwrite: true })
  )
);
```

</file>

<file>

## path: /src/mina-signer/package.json

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/package.json

```json
{
  "name": "mina-signer",
  "description": "Node API for signing transactions on various networks for Mina Protocol",
  "version": "2.1.1",
  "type": "module",
  "scripts": {
    "build": "tsc -p ../../tsconfig.mina-signer.json",
    "build:cjs": "./build-cjs.sh",
    "build:web": "./build-web.sh",
    "test": "for f in ./**/*.test.ts; do NODE_OPTIONS=--experimental-vm-modules npx jest $f || exit 1; done",
    "prepublishOnly": "npx rimraf ./dist && npm run build && npm run build:cjs && npm run build:web"
  },
  "keywords": [
    "mina",
    "coda",
    "cryptocurrency"
  ],
  "author": "o1labs",
  "license": "Apache-2.0",
  "homepage": "https://minaprotocol.com/",
  "repository": "https://github.com/o1-labs/o1js",
  "bugs": "https://github.com/o1-labs/o1js/issues",
  "main": "dist/node/mina-signer/MinaSigner.js",
  "types": "dist/node/mina-signer/index.d.ts",
  "exports": {
    "web": "./dist/web/index.js",
    "require": "./dist/node/mina-signer/index.cjs",
    "node": "./dist/node/mina-signer/MinaSigner.js",
    "default": "./dist/web/index.js"
  },
  "files": [
    "dist",
    "README.md"
  ],
  "dependencies": {
    "blakejs": "^1.2.1",
    "js-sha256": "^0.9.0"
  }
}
```

</file>

<file>

## path: /src/mina-signer/src/TSTypes.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/src/TSTypes.ts

```
import type { ZkappCommand as ZkappCommandJson } from '../../bindings/mina-
transaction/gen/transaction-json.js';
import type { SignatureJson } from './signature.js';

export type UInt32 = number | bigint | string;
export type UInt64 = number | bigint | string;

export type Field = number | bigint | string;

export type PublicKey = string;
export type PrivateKey = string;
export type Signature = SignatureJson;
export type Network = 'mainnet' | 'testnet';

export type Keypair = {
  readonly privateKey: PrivateKey;
  readonly publicKey: PublicKey;
};

export type Common = {
  readonly to: PublicKey;
  readonly from: PublicKey;
  readonly fee: UInt64;
  readonly nonce: UInt32;
  readonly memo?: string;
  readonly validUntil?: UInt32;
};
export type StrictCommon = {
  readonly to: string;
  readonly from: string;
  readonly fee: string;
  readonly nonce: string;
  readonly memo: string;
  readonly validUntil: string;
};

export type StakeDelegation = Common;
export type Payment = Common & { readonly amount: UInt64 };

type FeePayer = {
  readonly feePayer: PublicKey;
  readonly fee: UInt64;
  readonly nonce: UInt32;
  readonly memo?: string;
  readonly validUntil?: UInt32 | null;
};
export type StrictFeePayer = {
  readonly feePayer: PublicKey;
  readonly fee: string;
  readonly nonce: string;
  readonly memo: string;
  readonly validUntil: string | null;
};

export type ZkappCommand = {
  readonly zkappCommand: ZkappCommandJson;
  readonly feePayer: FeePayer;
};

export type SignableData = string | StakeDelegation | Payment;

export type SignedLegacy<T> = {
  signature: SignatureJson;
  publicKey: PublicKey;
  data: T;
};
export type Signed<T> = {
  signature: string; // base58
  publicKey: PublicKey;
  data: T;
};

export type SignedAny = SignedLegacy<SignableData> | Signed<ZkappCommand>;

export type Group = {
  x: Field;
  y: Field;
};

export type Nullifier = {
  publicKey: Group;
  public: {
    nullifier: Group;
    s: Field;
  };
  private: {
    c: Field;
    g_r: Group;
    h_m_pk_r: Group;
  };
};
```

</file>

<file>

## path: /src/mina-signer/src/Utils.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/src/Utils.ts

```
import { SignatureJson } from './signature.js';
import type {
  Payment,
  StakeDelegation,
  ZkappCommand,
  Signed,
  SignedAny,
  SignedLegacy,
  SignableData,
} from './TSTypes.js';

function hasCommonProperties(data: SignableData | ZkappCommand) {
  return (
    data.hasOwnProperty('to') &&
    data.hasOwnProperty('from') &&
    data.hasOwnProperty('fee') &&
    data.hasOwnProperty('nonce')
  );
}

export function isZkappCommand(
  p: SignableData | ZkappCommand
): p is ZkappCommand {
  return p.hasOwnProperty('zkappCommand') && p.hasOwnProperty('feePayer');
}

export function isPayment(p: SignableData | ZkappCommand): p is Payment {
  return hasCommonProperties(p) && p.hasOwnProperty('amount');
}

export function isStakeDelegation(
  p: SignableData | ZkappCommand
): p is StakeDelegation {
  return hasCommonProperties(p) && !p.hasOwnProperty('amount');
}

function isLegacySignature(s: string | SignatureJson): s is SignatureJson {
  return typeof s === 'object' && 'field' in s && 'scalar' in s;
}

export function isSignedZkappCommand(p: SignedAny): p is Signed<ZkappCommand> {
  return (
    p.data.hasOwnProperty('zkappCommand') &&
    p.data.hasOwnProperty('feePayer') &&
    typeof p.signature === 'string'
  );
}

export function isSignedPayment(p: SignedAny): p is SignedLegacy<Payment> {
  return (
    hasCommonProperties(p.data) &&
    isLegacySignature(p.signature) &&
    p.data.hasOwnProperty('amount')
  );
```

```
}

export function isSignedDelegation(
  p: SignedAny
): p is SignedLegacy<StakeDelegation> {
  return (
    hasCommonProperties(p.data) &&
    isLegacySignature(p.signature) &&
    !p.data.hasOwnProperty('amount')
  );
}

export function isSignedString(p: SignedAny): p is SignedLegacy<string> {
  return typeof p.data === 'string' && isLegacySignature(p.signature);
}
```

</file>

<file>

## path: /src/mina-signer/src/memo.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/src/memo.ts

```
import {
  Binable,
  defineBinable,
  stringFromBytes,
  stringLengthInBytes,
  stringToBytes,
  withBits,
} from '../../bindings/lib/binable.js';
import { base58 } from '../../lib/base58.js';
import {
  HashInputLegacy,
  hashWithPrefix,
  packToFieldsLegacy,
  prefixes,
} from '../../provable/poseidon-bigint.js';
import { versionBytes } from '../../bindings/crypto/constants.js';

export { Memo };

function fromString(memo: string) {
  let length = stringLengthInBytes(memo);
  if (length > 32) throw Error('Memo.fromString: string too long');
  return (
    &nbsp;\x01${String.fromCharCode(length)}${memo}&nbsp; + '\x00'.repeat(32 - length)
  );
}
function toString(memo: string) {
  let totalLength = stringLengthInBytes(memo);
  if (totalLength !== 34) {
    throw Error(&nbsp;Memo.toString: length ${totalLength} does not equal 34&nbsp;);
  }
  if (memo[0] !== '\x01') {
    throw Error('Memo.toString: expected memo to start with 0x01 byte');
  }
  let length = memo.charCodeAt(1);
  if (length > 32) throw Error('Memo.toString: invalid length encoding');
  let bytes = stringToBytes(memo).slice(2, 2 + length);
  return stringFromBytes(bytes);
}

function hash(memo: string) {
  let bits = Memo.toBits(memo);
  let fields = packToFieldsLegacy(HashInputLegacy.bits(bits));
  return hashWithPrefix(prefixes.zkappMemo, fields);
}

const SIZE = 34;
const Binable: Binable<string> = defineBinable({
  toBytes(memo) {
    return stringToBytes(memo);
  },
  readBytes(bytes, start) {
    let end = start + SIZE;
    let memo = stringFromBytes(bytes.slice(start, end));
    return [memo, end];
  },
});

const Memo = {
  fromString,
  toString,
  hash,
  ...withBits(Binable, SIZE * 8),
  ...base58(Binable, versionBytes.userCommandMemo),
  sizeInBytes() {
    return SIZE;
  },
  emptyValue() {
    return Memo.fromString('');
  },
  toValidString(memo = '') {
    if (stringLengthInBytes(memo) > 32) throw Error('Memo: string too long');
    return memo;
  },
};
```

</file>

<file>

## path: /src/mina-signer/src/nullifier.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/src/nullifier.ts

```
import { Fq } from '../../bindings/crypto/finite_field.js';
import { Poseidon } from '../../bindings/crypto/poseidon.js';
import {
  Group,
  PublicKey,
  Scalar,
  PrivateKey,
} from '../../provable/curve-bigint.js';
import { Field } from '../../provable/field-bigint.js';
import { Nullifier } from './TSTypes.js';

export { createNullifier };

/**
 * PLUME: An ECDSA Nullifier Scheme for Unique
 * Pseudonymity within Zero Knowledge Proofs
 * https://eprint.iacr.org/2022/1255.pdf chapter 3 page 14
 */
function createNullifier(message: Field[], sk: PrivateKey): Nullifier {
  const Hash2 = Poseidon.hash;
  const Hash = Poseidon.hashToGroup;

  const pk = PublicKey.toGroup(PrivateKey.toPublicKey(sk));

  const G = Group.generatorMina;

  const r = Scalar.random();

  const gm = Hash([...message, ...Group.toFields(pk)]);
  if (!gm) throw Error('hashToGroup: Point is undefined');
  const h_m_pk = { x: gm.x, y: gm.y.x0 };

  const nullifier = Group.scale(h_m_pk, sk);
  const h_m_pk_r = Group.scale(h_m_pk, r);

  const g_r = Group.scale(G, r);

  const c = Hash2([
    ...Group.toFields(G),
    ...Group.toFields(pk),
    ...Group.toFields(h_m_pk),
    ...Group.toFields(nullifier),
    ...Group.toFields(g_r),
    ...Group.toFields(h_m_pk_r),
  ]);
  // operations on scalars (r) should be in Fq, rather than Fp
  // while c is in Fp (due to Poseidon.hash), c needs to be handled as an element from Fq
  const s = Fq.add(r, Fq.mul(sk, c));

  return {
    publicKey: toString(pk),
    private: {
      c: c.toString(),
      g_r: toString(g_r),
      h_m_pk_r: toString(h_m_pk_r),
    },
    public: {
      nullifier: toString(nullifier),
      s: s.toString(),
    },
  };
}

function toString({ x, y }: Group): { x: string; y: string } {
  return { x: x.toString(), y: y.toString() };
}
```

</file>

<file>

## path: /src/mina-signer/src/random-transaction.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/src/random-transaction.ts

```
import { SignedLegacy } from './transaction-hash.js';
import { DelegationJson, PaymentJson } from './sign-legacy.js';
import { Random } from '../../lib/testing/property.js';
import {
  PublicKey,
  ZkappCommand,
} from '../../bindings/mina-transaction/gen/transaction-bigint.js';
import { PrivateKey } from '../../provable/curve-bigint.js';
import { NetworkId, Signature } from './signature.js';

export { RandomTransaction };

let { record } = Random;

// legacy transactions
const common = record({
  fee: Random.map(Random.uint32, (x) => (x * BigInt(1e9)).toString()),
  feePayer: Random.json.publicKey,
  nonce: Random.json.uint32,
  validUntil: Random.json.uint32,
  memo: Random.json.memoString,
});
const payment = record<PaymentJson>({
  common,
  body: record({
    receiver: Random.json.publicKey,
    amount: Random.json.uint64,
  }),
});
const signedPayment = record<SignedLegacy<PaymentJson>>({
  data: payment,
  signature: Random.json.signatureJson,
});

const delegation = record<DelegationJson>({
  common,
  body: record({
    newDelegate: Random.json.publicKey,
  }),
});
const signedDelegation = record<SignedLegacy<DelegationJson>>({
  data: delegation,
  signature: Random.json.signatureJson,
});

// zkapp transactions
// generator for { feePayer, accountUpdates, memo } with 2 modifications to the naive version

// modification #1: call depth
// in every list of account updates, call depth starts from 0,
// can increase by at most one, decrease by any amount up to 0.
// we increment with 1/3 chance
let incrementDepth = Random.map(Random.dice(3), (x) => x === 0);
let callDepth = Random.step(
  incrementDepth,
  Random.dice.ofSize,
  (depth, increment, diceOfSize) => {
    return increment ? depth + 1 : diceOfSize(depth + 1);
  },
  0
);
// account update with a call depth that takes valid steps
let accountUpdate = Random.map(
  Random.accountUpdate,
  callDepth,
  (accountUpdate, callDepth) => {
    accountUpdate.body.callDepth = callDepth;
    return accountUpdate;
  }
);

// modification #2: use the fee payer's public key for account updates, with 1/3 chance
// this is an important test case and won't happen by chance
let useFeePayerKey = Random.map(Random.dice(3), (x) => x === 0);
let accountUpdateFrom = Random.dependent(
  accountUpdate,
  useFeePayerKey,
  (feePayer: PublicKey, [accountUpdate, useFeePayerKey]) => {
    if (useFeePayerKey) accountUpdate.body.publicKey = feePayer;
    return accountUpdate;
  }
);
let feePayerFrom = Random.dependent(
  Random.feePayer,
  (publicKey: PublicKey, [feePayer]) => {
    feePayer.body.publicKey = publicKey;
    feePayer.authorization = Signature.toBase58(Signature.dummy());
    return feePayer;
  }
);

let size = Random.nat(20);
// reset: true makes call depth start from 0 again at the start of each sampled array
let accountUpdatesFrom = Random.array(accountUpdateFrom, size, { reset: true });

let zkappCommandFrom = Random.dependent(
  feePayerFrom,
  accountUpdatesFrom,
  Random.memo,
  (
    publicKey: PublicKey,
    [feePayerFrom, accountUpdatesFrom, memo]
  ): ZkappCommand => {
    let feePayer = feePayerFrom(publicKey);
    let accountUpdates = accountUpdatesFrom.map((from) => from(publicKey));
    return { feePayer, accountUpdates, memo };
  }
);

let zkappCommand: Random<ZkappCommand> = zkappCommandFrom(Random.publicKey);
let zkappCommandAndFeePayerKey = Random.map(
  Random.privateKey,
  zkappCommandFrom,
  (feePayerKey, zkappCommandFrom) => ({
    feePayerKey,
    zkappCommand: zkappCommandFrom(PrivateKey.toPublicKey(feePayerKey)),
  })
);

// json zkapp command, supporting invalid samples
let accountUpdateJson = Random.map.withInvalid(
  Random.json.accountUpdate,
  callDepth,
  (accountUpdate, callDepth) => {
    accountUpdate.body.callDepth = callDepth;
    return accountUpdate;
  }
);
let zkappCommandJson = Random.record({
  feePayer: Random.json.feePayer,
  accountUpdates: Random.array(accountUpdateJson, size),
  memo: Random.memo,
});

const RandomTransaction = {
  payment,
  delegation,
  signedPayment,
  signedDelegation,
  zkappCommand,
  zkappCommandAndFeePayerKey,
  zkappCommandJson,
  networkId: Random.oneOf<NetworkId[]>('testnet', 'mainnet'),
};
```

</file>

<file>

## path: /src/mina-signer/src/rosetta.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/src/rosetta.ts

```
import { Binable } from '../../bindings/lib/binable.js';
import { PublicKey, Scalar } from '../../provable/curve-bigint.js';
import { Field } from '../../provable/field-bigint.js';
import { Memo } from './memo.js';
import { Signature } from './signature.js';

export { publicKeyToHex, rosettaTransactionToSignedCommand };

function publicKeyToHex(publicKey: PublicKey) {
  return fieldToHex(Field, publicKey.x, !!publicKey.isOdd);
}

function signatureFromHex(signatureHex: string): Signature {
  let half = signatureHex.length / 2;
  let fieldHex = signatureHex.slice(0, half);
  let scalarHex = signatureHex.slice(half);
  return {
    r: fieldFromHex(Field, fieldHex)[0],
    s: fieldFromHex(Scalar, scalarHex)[0],
  };
}
```

```
function fieldToHex<T extends Field | Scalar>(
  binable: Binable<T>,
  x: T,
  paddingBit: boolean = false
) {
  let bytes = binable.toBytes(x);
  // set highest bit (which is empty)
  bytes[bytes.length - 1] &= Number(paddingBit) << 7;
  // map each byte to a hex string of length 2
  return bytes
    .map((byte) => byte.toString(16).split('').reverse().join(''))
    .join('');
}
function fieldFromHex<T extends Field | Scalar>(
  binable: Binable<T>,
  hex: string
): [T, boolean] {
  let bytes: number[] = [];
  for (let i = 0; i < hex.length; i += 2) {
    let byte = parseInt(hex[i + 1] + hex[i], 16);
    bytes.push(byte);
  }
  // read highest bit
  let paddingBit = !!(bytes[bytes.length - 1] >> 7);
  bytes[bytes.length - 1] &= 0x7f;
  return [binable.fromBytes(bytes), paddingBit];
}

// TODO: clean up this logic, was copied over from OCaml code
function rosettaTransactionToSignedCommand({
  signature,
  payment,
  stake_delegation,
}: RosettaTransactionJson) {
  let signatureDecoded = signatureFromHex(signature);
  let signatureBase58 = Signature.toBase58(signatureDecoded);
  let [t, nonce] = (() => {
    if (payment !== null && stake_delegation === null) {
      let r = payment;
      let command = {
        receiver: r.to,
        source: r.from,
        kind: 'Payment' as const,
        fee_payer: r.from,
        fee_token: r.token,
        fee: r.fee,
        amount: r.amount,
        valid_until: r.valid_until,
        memo: r.memo,
      };
      return [command, r.nonce];
    } else if (payment === null && stake_delegation !== null) {
      let r = stake_delegation;
      let command = {
        receiver: r.new_delegate,
        source: r.delegator,
        kind: 'Delegation' as const,
        fee_payer: r.delegator,
        fee_token: '1',
        fee: r.fee,
        amount: null,
        valid_until: r.valid_until,
        memo: r.memo,
      };
      return [command, r.nonce];
    } else {
      throw Error('rosettaTransactionToSignedCommand: Unsupported transaction');
    }
  })();
  let payload = (() => {
    let fee_payer_pk = t.fee_payer;
    let source_pk = t.source;
    let receiver_pk = t.receiver;
    let memo = Memo.toBase58(Memo.fromString(t.memo ?? ''));
    let common = {
      fee: t.fee,
      fee_payer_pk,
      nonce,
      valid_until: t.valid_until,
      memo,
    };
    if (t.kind === 'Payment') {
      return {
        common,
        body: ['Payment', { source_pk, receiver_pk, amount: t.amount }],
      };
    } else if (t.kind === 'Delegation') {
      return {
        common,
        body: [
          'Stake_delegation',
          ['Set_delegate', { delegator: source_pk, new_delegate: receiver_pk }],
        ],
      };
    } else throw Error('rosettaTransactionToSignedCommand has a bug');
  })();
  return {
    signature: signatureBase58,
    signer: payload.common.fee_payer_pk,
    payload,
  };
}

type RosettaTransactionJson = {
  signature: string;
  payment: {
    to: string;
    from: string;
    fee: string;
    token: string;
    nonce: string;
    memo: string | null;
    amount: string;
    valid_until: string | null;
  } | null;
  stake_delegation: {
    delegator: string;
    new_delegate: string;
    fee: string;
    nonce: string;
    memo: string | null;
    valid_until: string | null;
  } | null;
};
```

</file>

<file>

## path: /src/mina-signer/src/sign-legacy.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/src/sign-legacy.ts

```
import { UInt32, UInt64 } from '../../provable/field-bigint.js';
import { PrivateKey, PublicKey } from '../../provable/curve-bigint.js';
import { HashInputLegacy } from '../../provable/poseidon-bigint.js';
import { Memo } from './memo.js';
import {
  SignatureJson,
  NetworkId,
  Signature,
  signLegacy,
  verifyLegacy,
} from './signature.js';
import { Json } from '../../bindings/mina-transaction/gen/transaction-bigint.js';
import { bytesToBits, stringToBytes } from '../../bindings/lib/binable.js';

export {
  signPayment,
  signStakeDelegation,
  signString,
  verifyPayment,
  verifyStakeDelegation,
  verifyStringSignature,
  paymentFromJson,
  delegationFromJson,
  commonFromJson,
  PaymentJson,
  PaymentJsonV1,
  DelegationJson,
  DelegationJsonV1,
  CommonJson,
  Tag,
  UserCommand,
  UserCommandEnum,
  BodyEnum,
  Payment,
  Delegation,
  Common,
};

function signPayment(
  payment: PaymentJson,
  privateKeyBase58: string,
  networkId: NetworkId
) {
  let command = paymentFromJson(payment);
  return signUserCommand(command, privateKeyBase58, networkId);
}
function signStakeDelegation(
  delegation: DelegationJson,
  privateKeyBase58: string,
  networkId: NetworkId
) {
  let command = delegationFromJson(delegation);
  return signUserCommand(command, privateKeyBase58, networkId);
}

function signUserCommand(
  command: UserCommand,
  privateKeyBase58: string,
  networkId: NetworkId
) {
  let input = toInputLegacy(command);
  let privateKey = PrivateKey.fromBase58(privateKeyBase58);
  let signature = signLegacy(input, privateKey, networkId);
  return Signature.toJSON(signature);
}

function verifyPayment(
  payment: PaymentJson,
  signatureJson: SignatureJson,
  publicKeyBase58: string,
  networkId: NetworkId
) {
  try {
    return verifyUserCommand(
      paymentFromJson(payment),
      signatureJson,
      publicKeyBase58,
      networkId
    );
  } catch {
    return false;
  }
}
function verifyStakeDelegation(
  delegation: DelegationJson,
  signatureJson: SignatureJson,
  publicKeyBase58: string,
  networkId: NetworkId
) {
  try {
    return verifyUserCommand(
      delegationFromJson(delegation),
      signatureJson,
      publicKeyBase58,
      networkId
    );
  } catch {
    return false;
  }
}

function verifyUserCommand(
  command: UserCommand,
  signatureJson: SignatureJson,
  publicKeyBase58: string,
  networkId: NetworkId
) {
  let input = toInputLegacy(command);
  let signature = Signature.fromJSON(signatureJson);
  let publicKey = PublicKey.fromBase58(publicKeyBase58);
  return verifyLegacy(signature, input, publicKey, networkId);
}

function toInputLegacy({ common, body }: UserCommand) {
  return HashInputLegacy.append(
    commonToInputLegacy(common),
    bodyToInputLegacy(body)
  );
}

// Mina_base.Transaction_union_payload.Body.to_input_legacy
function bodyToInputLegacy({
  tag,
  source,
  receiver,
  amount,
}: UserCommand['body']) {
  return [
    tagToInput(tag),
    PublicKey.toInputLegacy(source),
    PublicKey.toInputLegacy(receiver),
    HashInputLegacy.bits(legacyTokenId),
    HashInputLegacy.bits(UInt64.toBits(amount)),
    HashInputLegacy.bits([false]), // token_locked
  ].reduce(HashInputLegacy.append);
}

// Mina_base.Signed_command_payload.Common.to_input_legacy
function commonToInputLegacy({
  fee,
  feePayer,
  nonce,
  validUntil,
  memo,
}: UserCommand['common']) {
  return [
    HashInputLegacy.bits(UInt64.toBits(fee)),
    HashInputLegacy.bits(legacyTokenId),
    PublicKey.toInputLegacy(feePayer),
    HashInputLegacy.bits(UInt32.toBits(nonce)),
    HashInputLegacy.bits(UInt32.toBits(validUntil.value)),
    HashInputLegacy.bits(Memo.toBits(memo)),
  ].reduce(HashInputLegacy.append);
}

function tagToInput(tag: Tag) {
  let int = { Payment: 0, StakeDelegation: 1 }[tag];
  let bits = [int & 4, int & 2, int & 1].map(Boolean);
  return HashInputLegacy.bits(bits);
}
const legacyTokenId = [true, ...Array<boolean>(63).fill(false)];

function paymentFromJson({
  common,
  body: { receiver, amount },
}: PaymentJson): UserCommand {
  return {
    common: commonFromJson(common),
    body: {
      tag: 'Payment',
      source: PublicKey.fromJSON(common.feePayer),
      receiver: PublicKey.fromJSON(receiver),
      amount: UInt64.fromJSON(amount),
    },
  };
}

function delegationFromJson({
  common,
  body: { newDelegate },
}: DelegationJson): UserCommand {
  return {
    common: commonFromJson(common),
    body: {
      tag: 'StakeDelegation',
      source: PublicKey.fromJSON(common.feePayer),
      receiver: PublicKey.fromJSON(newDelegate),
      amount: UInt64(0),
    },
  };
}

function commonFromJson(c: CommonJson): Common {
  return {
    fee: UInt64.fromJSON(c.fee),
    feePayer: PublicKey.fromJSON(c.feePayer),
    nonce: UInt32.fromJSON(c.nonce),
    validUntil: { type: 'SinceGenesis', value: UInt32.fromJSON(c.validUntil) },
    // TODO: this might need to be fromBase58
    memo: Memo.fromString(c.memo),
  };
}

function signString(
  string: string,
  privateKeyBase58: string,
  networkId: NetworkId
) {
  let input = stringToInput(string);
  let privateKey = PrivateKey.fromBase58(privateKeyBase58);
  let signature = signLegacy(input, privateKey, networkId);
  return Signature.toJSON(signature);
}
function verifyStringSignature(
  string: string,
  signatureJson: SignatureJson,
  publicKeyBase58: string,
  networkId: NetworkId
) {
  try {
    let input = stringToInput(string);
    let signature = Signature.fromJSON(signatureJson);
    let publicKey = PublicKey.fromBase58(publicKeyBase58);
    return verifyLegacy(signature, input, publicKey, networkId);
  } catch {
    return false;
  }
}

function stringToInput(string: string) {
  let bits = stringToBytes(string)
    .map((byte) => bytesToBits([byte]).reverse())
    .flat();
  return HashInputLegacy.bits(bits);
}

// types

type Tag = 'Payment' | 'StakeDelegation';

type UserCommand = {
  common: Common;
  body: {
    tag: Tag;
    source: PublicKey;
    receiver: PublicKey;
    amount: UInt64;
  };
};

type UserCommandEnum = {
  common: Common;
  body: BodyEnum;
};

type BodyEnum =
  | { type: 'Payment'; value: Payment }
  | {
      type: 'StakeDelegation';
      value: { type: 'SetDelegate'; value: Delegation };
    };

type Common = {
  fee: UInt64;
  feePayer: PublicKey;
  nonce: UInt32;
  validUntil: { type: 'SinceGenesis'; value: UInt32 };
  memo: string;
};

type Payment = {
  receiver: PublicKey;
  amount: UInt64;
};
type Delegation = {
  newDelegate: PublicKey;
};

type CommonJson = {
  fee: Json.UInt64;
  feePayer: Json.PublicKey;
  nonce: Json.UInt32;
  validUntil: Json.UInt32;
  memo: string;
};

type PaymentJson = {
  common: CommonJson;
  body: {
    receiver: Json.PublicKey;
    amount: Json.UInt64;
  };
};

type PaymentJsonV1 = {
  common: CommonJson;
  body: {
    source: Json.PublicKey;
    receiver: Json.PublicKey;
    amount: Json.UInt64;
  };
};

type DelegationJson = {
  common: CommonJson;
  body: {
    newDelegate: Json.PublicKey;
  };
};

type DelegationJsonV1 = {
  common: CommonJson;
  body: {
    delegator: Json.PublicKey;
    newDelegate: Json.PublicKey;
  };
};
```

</file>

<file>

## path: /src/mina-signer/src/sign-legacy.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/src/sign-legacy.unit-test.ts

```
import {
  payments,
  delegations,
  strings,
  keypair,
  signatures,
} from './test-vectors/legacySignatures.js';
```

```
import {
  PaymentJson,
  signPayment,
  signStakeDelegation,
  signString,
  verifyPayment,
  verifyStakeDelegation,
  verifyStringSignature,
} from './sign-legacy.js';
import { NetworkId, Signature, SignatureJson } from './signature.js';
import { expect } from 'expect';
import { PublicKey, Scalar } from '../../provable/curve-bigint.js';
import { Field } from '../../provable/field-bigint.js';
import { Random, test } from '../../lib/testing/property.js';
import { RandomTransaction } from './random-transaction.js';

let { privateKey, publicKey } = keypair;
let networks: NetworkId[] = ['testnet', 'mainnet'];

// test hard-coded cases against reference signature

for (let network of networks) {
  let i = 0;
  let reference = signatures[network];

  for (let payment of payments) {
    let signature = signPayment(payment, privateKey, network);
    let sig = Signature.fromJSON(signature);
    let ref = reference[i++];
    expect(sig.r).toEqual(BigInt(ref.field));
    expect(sig.s).toEqual(BigInt(ref.scalar));
    let ok = verifyPayment(payment, signature, publicKey, network);
    expect(ok).toEqual(true);
  }

  for (let delegation of delegations) {
    let signature = signStakeDelegation(delegation, privateKey, network);
    let sig = Signature.fromJSON(signature);
    let ref = reference[i++];
    expect(sig.r).toEqual(BigInt(ref.field));
    expect(sig.s).toEqual(BigInt(ref.scalar));
    let ok = verifyStakeDelegation(delegation, signature, publicKey, network);
    expect(ok).toEqual(true);
  }

  for (let string of strings) {
    let signature = signString(string, privateKey, network);
    let sig = Signature.fromJSON(signature);
    let ref = reference[i++];
    expect(sig.r).toEqual(BigInt(ref.field));
    expect(sig.s).toEqual(BigInt(ref.scalar));
    let ok = verifyStringSignature(string, signature, publicKey, network);
  expect(ok).toEqual(true);
 }
}

// sign & verify with randomly generated payments

test(
 RandomTransaction.payment,
 Random.json.keypair,
 Random.json.privateKey,
 (payment, { privateKey, publicKey }, otherKey, assert) => {
  let verify = (sig: SignatureJson, network: NetworkId) =>
   verifyPayment(payment, sig, publicKey, network);

  // valid signatures & verification matrix
  let testnet = signPayment(payment, privateKey, 'testnet');
  let mainnet = signPayment(payment, privateKey, 'mainnet');
  assert(verify(testnet, 'testnet') === true);
  assert(verify(testnet, 'mainnet') === false);
  assert(verify(mainnet, 'testnet') === false);
  assert(verify(mainnet, 'mainnet') === true);

  // fails when signing with wrong private key
  let testnetWrong = signPayment(payment, otherKey, 'testnet');
  let mainnetWrong = signPayment(payment, otherKey, 'mainnet');
  assert(verify(testnetWrong, 'testnet') === false);
  assert(verify(mainnetWrong, 'mainnet') === false);
 }
);

// generative negative tests - any invalid payment should fail

test.negative(
 RandomTransaction.payment.invalid!,
 Random.json.privateKey,
 RandomTransaction.networkId,
 (payment, privateKey, network) => signPayment(payment, privateKey, network)
);

// negative tests with invalid payments

let validPayment = payments[0];
let amountTooLarge = {
 common: validPayment.common,
 body: {
  ...validPayment.body,
  amount: (2n ** 64n).toString(),
 },
};
let signature = Signature.toJSON({ r: Field.random(), s: Scalar.random() });

expect(() => signPayment(amountTooLarge, privateKey, 'mainnet')).toThrow(
  &nbsp;inputs larger than ${2n ** 64n - 1n} are not allowed&nbsp;
);
expect(verifyPayment(amountTooLarge, signature, publicKey, 'mainnet')).toEqual(
  false
);

// negative tests with invalid signatures

let garbageSignature = { field: 'garbage', scalar: 'garbage' };
let signatureFieldTooLarge = Signature.toJSON({
  r: Field.modulus,
  s: Scalar.random(),
});
let signatureScalarTooLarge = Signature.toJSON({
  r: Field.random(),
  s: Scalar.modulus,
});

expect(
  verifyPayment(validPayment, garbageSignature, publicKey, 'mainnet')
).toEqual(false);
expect(
  verifyPayment(validPayment, signatureFieldTooLarge, publicKey, 'mainnet')
).toEqual(false);
expect(
  verifyPayment(validPayment, signatureScalarTooLarge, publicKey, 'mainnet')
).toEqual(false);

console.log(
  'legacy signatures match the test vectors and successfully verify!      '
);
process.exit(0);
```

</file>

<file>

## path: /src/mina-signer/src/sign-zkapp-command.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/src/sign-zkapp-command.ts

```
import { Bool, Field, Sign, UInt32 } from '../../provable/field-bigint.js';
import { PrivateKey, PublicKey } from '../../provable/curve-bigint.js';
import {
  Json,
  AccountUpdate,
  ZkappCommand,
} from '../../bindings/mina-transaction/gen/transaction-bigint.js';
import {
  hashWithPrefix,
  packToFields,
  prefixes,
} from '../../provable/poseidon-bigint.js';
import { Memo } from './memo.js';
import {
  NetworkId,
  Signature,
  signFieldElement,
  verifyFieldElement,
} from './signature.js';
import { mocks } from '../../bindings/crypto/constants.js';

// external API
export { signZkappCommand, verifyZkappCommandSignature };

// internal API
export {
  transactionCommitments,
  verifyAccountUpdateSignature,
  accountUpdatesToCallForest,
  callForestHash,
  accountUpdateHash,
  feePayerHash,
  createFeePayer,
  accountUpdateFromFeePayer,
  isCallDepthValid,
};

function signZkappCommand(
  zkappCommand_: Json.ZkappCommand,
  privateKeyBase58: string,
  networkId: NetworkId
): Json.ZkappCommand {
  let zkappCommand = ZkappCommand.fromJSON(zkappCommand_);

  let { commitment, fullCommitment } = transactionCommitments(zkappCommand);
  let privateKey = PrivateKey.fromBase58(privateKeyBase58);
  let publicKey = zkappCommand.feePayer.body.publicKey;

  // sign fee payer
  let signature = signFieldElement(fullCommitment, privateKey, networkId);
  zkappCommand.feePayer.authorization = Signature.toBase58(signature);

  // sign other updates with the same public key that require a signature
  for (let update of zkappCommand.accountUpdates) {
    if (update.body.authorizationKind.isSigned === 0n) continue;
    if (!PublicKey.equal(update.body.publicKey, publicKey)) continue;
    let { useFullCommitment } = update.body;
    let usedCommitment = useFullCommitment === 1n ? fullCommitment : commitment;
    let signature = signFieldElement(usedCommitment, privateKey, networkId);
    update.authorization = { signature: Signature.toBase58(signature) };
  }
  return ZkappCommand.toJSON(zkappCommand);
```

```
}

function verifyZkappCommandSignature(
  zkappCommand_: Json.ZkappCommand,
  publicKeyBase58: string,
  networkId: NetworkId
) {
  let zkappCommand = ZkappCommand.fromJSON(zkappCommand_);

  let { commitment, fullCommitment } = transactionCommitments(zkappCommand);
  let publicKey = PublicKey.fromBase58(publicKeyBase58);

  // verify fee payer signature
  let signature = Signature.fromBase58(zkappCommand.feePayer.authorization);
  let ok = verifyFieldElement(signature, fullCommitment, publicKey, networkId);
  if (!ok) return false;

  // verify other signatures for the same public key
  for (let update of zkappCommand.accountUpdates) {
    if (update.body.authorizationKind.isSigned === 0n) continue;
    if (!PublicKey.equal(update.body.publicKey, publicKey)) continue;
    let { useFullCommitment } = update.body;
    let usedCommitment = useFullCommitment === 1n ? fullCommitment : commitment;
    if (update.authorization.signature === undefined) return false;
    let signature = Signature.fromBase58(update.authorization.signature);
    ok = verifyFieldElement(signature, usedCommitment, publicKey, networkId);
    if (!ok) return false;
  }
  return ok;
}

function verifyAccountUpdateSignature(
  update: AccountUpdate,
  transactionCommitments: { commitment: bigint; fullCommitment: bigint },
  networkId: NetworkId
) {
  if (update.authorization.signature === undefined) return false;

  let { publicKey, useFullCommitment } = update.body;
  let { commitment, fullCommitment } = transactionCommitments;
  let usedCommitment = useFullCommitment === 1n ? fullCommitment : commitment;
  let signature = Signature.fromBase58(update.authorization.signature);

  return verifyFieldElement(signature, usedCommitment, publicKey, networkId);
}

function transactionCommitments(zkappCommand: ZkappCommand) {
  if (!isCallDepthValid(zkappCommand)) {
    throw Error('zkapp command: invalid call depth');
  }
  let callForest = accountUpdatesToCallForest(zkappCommand.accountUpdates);
  let commitment = callForestHash(callForest);
  let memoHash = Memo.hash(Memo.fromBase58(zkappCommand.memo));
  let feePayerDigest = feePayerHash(zkappCommand.feePayer);
  let fullCommitment = hashWithPrefix(prefixes.accountUpdateCons, [
    memoHash,
    feePayerDigest,
    commitment,
  ]);
  return { commitment, fullCommitment };
}

type CallTree = { accountUpdate: AccountUpdate; children: CallForest };
type CallForest = CallTree[];

/**
 * Turn flat list into a hierarchical structure (forest) by letting the callDepth
 * determine parent-child relationships
 */
function accountUpdatesToCallForest(updates: AccountUpdate[], callDepth = 0) {
  let remainingUpdates = callDepth > 0 ? updates : [...updates];
  let forest: CallForest = [];
  while (remainingUpdates.length > 0) {
    let accountUpdate = remainingUpdates[0];
    if (accountUpdate.body.callDepth < callDepth) return forest;
    remainingUpdates.shift();
    let children = accountUpdatesToCallForest(remainingUpdates, callDepth + 1);
    forest.push({ accountUpdate, children });
  }
  return forest;
}

function accountUpdateHash(update: AccountUpdate) {
  assertAuthorizationKindValid(update);
  let input = AccountUpdate.toInput(update);
  let fields = packToFields(input);
  return hashWithPrefix(prefixes.body, fields);
}

function callForestHash(forest: CallForest): Field {
  let stackHash = 0n;
  for (let callTree of [...forest].reverse()) {
    let calls = callForestHash(callTree.children);
    let treeHash = accountUpdateHash(callTree.accountUpdate);
    let nodeHash = hashWithPrefix(prefixes.accountUpdateNode, [
      treeHash,
      calls,
    ]);
    stackHash = hashWithPrefix(prefixes.accountUpdateCons, [
      nodeHash,
      stackHash,
    ]);
  }
  return stackHash;
```

```
}

type FeePayer = ZkappCommand['feePayer'];

function createFeePayer(feePayer: FeePayer['body']): FeePayer {
  return { authorization: '', body: feePayer };
}
function feePayerHash(feePayer: FeePayer) {
  let accountUpdate = accountUpdateFromFeePayer(feePayer);
  return accountUpdateHash(accountUpdate);
}

function accountUpdateFromFeePayer({
  body: { fee, nonce, publicKey, validUntil },
  authorization: signature,
}: FeePayer): AccountUpdate {
  let { body } = AccountUpdate.emptyValue();
  body.publicKey = publicKey;
  body.balanceChange = { magnitude: fee, sgn: Sign(-1) };
  body.incrementNonce = Bool(true);
  body.preconditions.network.globalSlotSinceGenesis = {
    isSome: Bool(true),
    value: { lower: UInt32(0), upper: validUntil ?? UInt32.maxValue },
  };
  body.preconditions.account.nonce = {
    isSome: Bool(true),
    value: { lower: nonce, upper: nonce },
  };
  body.useFullCommitment = Bool(true);
  body.implicitAccountCreationFee = Bool(true);
  body.authorizationKind = {
    isProved: Bool(false),
    isSigned: Bool(true),
    verificationKeyHash: Field(mocks.dummyVerificationKeyHash),
  };
  return { body, authorization: { signature } };
}

function isCallDepthValid(zkappCommand: ZkappCommand) {
  let callDepths = zkappCommand.accountUpdates.map((a) => a.body.callDepth);
  let current = callDepths.shift() ?? 0;
  if (current !== 0) return false;
  for (let callDepth of callDepths) {
    if (callDepth < 0) return false;
    if (callDepth - current > 1) return false;
    current = callDepth;
  }
  return true;
}

function assertAuthorizationKindValid(accountUpdate: AccountUpdate) {
  let { isSigned, isProved, verificationKeyHash } =
    accountUpdate.body.authorizationKind;
  if (isProved && isSigned)
    throw Error(
      'Invalid authorization kind: Only one of &nbsp;isProved&nbsp; and &nbsp;isSigned&nbsp; may be true.'
    );
  if (
    !isProved &&
    verificationKeyHash !== Field(mocks.dummyVerificationKeyHash)
  )
    throw Error(
      &nbsp;Invalid authorization kind: If \&nbsp;isProved\&nbsp; is false, verification key hash must be
${mocks.dummyVerificationKeyHash}, got ${verificationKeyHash}&nbsp;
    );
}
```

</file>

<file>

## path: /src/mina-signer/src/sign-zkapp-command.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/src/sign-zkapp-command.unit-test.ts

```
import { expect } from 'expect';
import { Ledger, Test, Pickles } from '../../snarky.js';
import {
  PrivateKey as PrivateKeySnarky,
  PublicKey as PublicKeySnarky,
} from '../../lib/signature.js';
import {
  AccountUpdate as AccountUpdateSnarky,
  ZkappCommand as ZkappCommandSnarky,
} from '../../lib/account_update.js';
import { PrivateKey, PublicKey } from '../../provable/curve-bigint.js';
import {
  AccountUpdate,
  Field,
  Json,
  ZkappCommand,
} from '../../bindings/mina-transaction/gen/transaction-bigint.js';
import * as TypesSnarky from '../../bindings/mina-transaction/gen/transaction.js';
import {
  accountUpdateFromFeePayer,
  accountUpdateHash,
  accountUpdatesToCallForest,
  callForestHash,
  feePayerHash,
  isCallDepthValid,
  signZkappCommand,
  verifyZkappCommandSignature,
} from './sign-zkapp-command.js';
```

```
import {
  hashWithPrefix,
  packToFields,
  prefixes,
} from '../../provable/poseidon-bigint.js';
import { packToFields as packToFieldsSnarky } from '../../lib/hash.js';
import { Memo } from './memo.js';
import {
  NetworkId,
  Signature,
  signFieldElement,
  verifyFieldElement,
} from './signature.js';
import { Random, test, withHardCoded } from '../../lib/testing/property.js';
import { RandomTransaction } from './random-transaction.js';
import { Ml, MlHashInput } from '../../lib/ml/conversion.js';
import { FieldConst } from '../../lib/field.js';
import { mocks } from '../../bindings/crypto/constants.js';

// monkey-patch bigint to json
(BigInt.prototype as any).toJSON = function () {
  return this.toString();
};
let { parse, stringify } = JSON;
const toJSON = (x: any) => parse(stringify(x));

// public key roundtrip & consistency w/ OCaml serialization
test(Random.json.publicKey, (publicKeyBase58) => {
  let pkSnarky = PublicKeySnarky.fromBase58(publicKeyBase58);
  let pk = PublicKey.fromJSON(publicKeyBase58);
  expect(pk.x).toEqual(pkSnarky.x.toBigInt());
  expect(pk.isOdd).toEqual(pkSnarky.isOdd.toField().toBigInt());
  expect(PublicKey.toJSON(pk)).toEqual(publicKeyBase58);
});

// empty account update
let dummy = AccountUpdate.emptyValue();
let dummySnarky = AccountUpdateSnarky.dummy();
expect(AccountUpdate.toJSON(dummy)).toEqual(
  AccountUpdateSnarky.toJSON(dummySnarky)
);

let dummyInput = AccountUpdate.toInput(dummy);
let dummyInputSnarky = MlHashInput.from(
  Test.hashInputFromJson.body(
    JSON.stringify(AccountUpdateSnarky.toJSON(dummySnarky).body)
  )
);
expect(stringify(dummyInput.fields)).toEqual(
  stringify(dummyInputSnarky.fields)
);
expect(stringify(dummyInput.packed)).toEqual(
  stringify(dummyInputSnarky.packed)
);

test(Random.accountUpdate, (accountUpdate) => {
  fixVerificationKey(accountUpdate);

  // example account update
  let accountUpdateJson: Json.AccountUpdate =
    AccountUpdate.toJSON(accountUpdate);

  // account update hash
  let accountUpdateSnarky = AccountUpdateSnarky.fromJSON(accountUpdateJson);
  let inputSnarky = TypesSnarky.AccountUpdate.toInput(accountUpdateSnarky);
  let input = AccountUpdate.toInput(accountUpdate);
  expect(toJSON(input.fields)).toEqual(toJSON(inputSnarky.fields));
  expect(toJSON(input.packed)).toEqual(toJSON(inputSnarky.packed));

  let packed = packToFields(input);
  let packedSnarky = packToFieldsSnarky(inputSnarky);
  expect(toJSON(packed)).toEqual(toJSON(packedSnarky));

  let hash = accountUpdateHash(accountUpdate);
  let hashSnarky = accountUpdateSnarky.hash();
  expect(hash).toEqual(hashSnarky.toBigInt());
});

// private key to/from base58
test(Random.json.privateKey, (feePayerKeyBase58) => {
  let feePayerKey = PrivateKey.fromBase58(feePayerKeyBase58);
  let feePayerKeySnarky = PrivateKeySnarky.fromBase58(feePayerKeyBase58);
  let feePayerCompressed = feePayerKeySnarky.s.toFieldsCompressed();
  expect(feePayerKey).toEqual(feePayerCompressed.field.toBigInt());
  expect(PrivateKey.toBase58(feePayerKey)).toEqual(feePayerKeyBase58);
});

// memo
let memoGenerator = withHardCoded(Random.json.memoString, 'hello world');
test(memoGenerator, (memoString) => {
  let memo = Memo.fromString(memoString);
  let memoBase58 = Memo.toBase58(memo);
  let memoBase581 = Test.encoding.memoToBase58(memoString);
  expect(memoBase58).toEqual(memoBase581);
  let memoRecovered = Memo.fromBase58(memoBase58);
  expect(memoRecovered).toEqual(memo);
});

// zkapp transaction - basic properties & commitment
test(RandomTransaction.zkappCommand, (zkappCommand, assert) => {
  zkappCommand.accountUpdates.forEach(fixVerificationKey);

  assert(isCallDepthValid(zkappCommand));
  let zkappCommandJson = ZkappCommand.toJSON(zkappCommand);
  let ocamlCommitments = Test.hashFromJson.transactionCommitments(
    JSON.stringify(zkappCommandJson)
  );
  let callForest = accountUpdatesToCallForest(zkappCommand.accountUpdates);
  let commitment = callForestHash(callForest);
  expect(commitment).toEqual(FieldConst.toBigint(ocamlCommitments.commitment));
});

// invalid zkapp transactions
test.negative(
  RandomTransaction.zkappCommandJson.invalid!,
  Random.json.privateKey,
  RandomTransaction.networkId,
  (zkappCommand, feePayerKey, networkId) => {
    signZkappCommand(zkappCommand, feePayerKey, networkId);
  }
);

// zkapp transaction
test(
  RandomTransaction.zkappCommandAndFeePayerKey,
  ({ feePayerKey, zkappCommand }) => {
    zkappCommand.accountUpdates.forEach(fixVerificationKey);

    let feePayerKeyBase58 = PrivateKey.toBase58(feePayerKey);
    let feePayerKeySnarky = PrivateKeySnarky.fromBase58(feePayerKeyBase58);
    let feePayerAddress = PrivateKey.toPublicKey(feePayerKey);

    let { feePayer, memo: memoBase58 } = zkappCommand;
    feePayer.authorization = Signature.toBase58(Signature.dummy());
    let zkappCommandJson = ZkappCommand.toJSON(zkappCommand);

    // o1js fromJSON -> toJSON roundtrip, + consistency with mina-signer
    let zkappCommandSnarky = ZkappCommandSnarky.fromJSON(zkappCommandJson);
    let zkappCommandJsonSnarky = ZkappCommandSnarky.toJSON(zkappCommandSnarky);
    expect(JSON.stringify(zkappCommandJson)).toEqual(
      JSON.stringify(zkappCommandJsonSnarky)
    );
    let recoveredZkappCommand = ZkappCommand.fromJSON(zkappCommandJson);
    expect(recoveredZkappCommand).toEqual(zkappCommand);

    // tx commitment
    let ocamlCommitments = Test.hashFromJson.transactionCommitments(
      JSON.stringify(zkappCommandJson)
    );
    let callForest = accountUpdatesToCallForest(zkappCommand.accountUpdates);
    let commitment = callForestHash(callForest);
    expect(commitment).toEqual(
      FieldConst.toBigint(ocamlCommitments.commitment)
    );

    let memo = Memo.fromBase58(memoBase58);
    let memoHash = Memo.hash(memo);
    let memoHashSnarky = Test.encoding.memoHashBase58(memoBase58);
    expect(memoHash).toEqual(FieldConst.toBigint(memoHashSnarky));

    let feePayerAccountUpdate = accountUpdateFromFeePayer(feePayer);
    let feePayerJson = AccountUpdate.toJSON(feePayerAccountUpdate);

    let feePayerInput = AccountUpdate.toInput(feePayerAccountUpdate);
    let feePayerInput1 = MlHashInput.from(
      Test.hashInputFromJson.body(JSON.stringify(feePayerJson.body))
    );
    expect(stringify(feePayerInput.fields)).toEqual(
      stringify(feePayerInput1.fields)
    );
    expect(stringify(feePayerInput.packed)).toEqual(
      stringify(feePayerInput1.packed)
    );

    let feePayerDigest = feePayerHash(feePayer);
    expect(feePayerDigest).toEqual(
      FieldConst.toBigint(ocamlCommitments.feePayerHash)
    );

    let fullCommitment = hashWithPrefix(prefixes.accountUpdateCons, [
      memoHash,
      feePayerDigest,
      commitment,
    ]);
    expect(fullCommitment).toEqual(
      FieldConst.toBigint(ocamlCommitments.fullCommitment)
    );

    // signature
    let sigTestnet = signFieldElement(fullCommitment, feePayerKey, 'testnet');
    let sigMainnet = signFieldElement(fullCommitment, feePayerKey, 'mainnet');
    let sigTestnetOcaml = Test.signature.signFieldElement(
      ocamlCommitments.fullCommitment,
      Ml.fromPrivateKey(feePayerKeySnarky),
      false
    );
    let sigMainnetOcaml = Test.signature.signFieldElement(
      ocamlCommitments.fullCommitment,
      Ml.fromPrivateKey(feePayerKeySnarky),
      true
    );
    expect(Signature.toBase58(sigTestnet)).toEqual(sigTestnetOcaml);
    expect(Signature.toBase58(sigMainnet)).toEqual(sigMainnetOcaml);

    let verify = (s: Signature, id: NetworkId) =>
      verifyFieldElement(s, fullCommitment, feePayerAddress, id);
    expect(verify(sigTestnet, 'testnet')).toEqual(true);
    expect(verify(sigTestnet, 'mainnet')).toEqual(false);
```

```
    expect(verify(sigMainnet, 'testnet')).toEqual(false);
    expect(verify(sigMainnet, 'mainnet')).toEqual(true);

    // full end-to-end test: sign a zkapp transaction
    let sTest = signZkappCommand(
      zkappCommandJson,
      feePayerKeyBase58,
      'testnet'
    );
    expect(sTest.feePayer.authorization).toEqual(sigTestnetOcaml);
    let sMain = signZkappCommand(
      zkappCommandJson,
      feePayerKeyBase58,
      'mainnet'
    );
    expect(sMain.feePayer.authorization).toEqual(sigMainnetOcaml);

    let feePayerAddressBase58 = PublicKey.toBase58(feePayerAddress);
    expect(
      verifyZkappCommandSignature(sTest, feePayerAddressBase58, 'testnet')
    ).toEqual(true);
    expect(
      verifyZkappCommandSignature(sTest, feePayerAddressBase58, 'mainnet')
    ).toEqual(false);
    expect(
      verifyZkappCommandSignature(sMain, feePayerAddressBase58, 'testnet')
    ).toEqual(false);
    expect(
      verifyZkappCommandSignature(sMain, feePayerAddressBase58, 'mainnet')
    ).toEqual(true);
  }
);

console.log('to/from json, hashes & signatures are consistent!      ');

function fixVerificationKey(a: AccountUpdate) {
  // ensure verification key is valid
  if (a.body.update.verificationKey.isSome === 1n) {
    let [, data, hash] = Pickles.dummyVerificationKey();
    a.body.update.verificationKey.value = {
      data,
      hash: FieldConst.toBigint(hash),
    };
  } else {
    a.body.update.verificationKey.value = {
      data: '',
      hash: Field(0),
    };
  }
  fixVerificationKeyHash(a);
}
```

```
function fixVerificationKeyHash(a: AccountUpdate) {
  a.body.authorizationKind.verificationKeyHash = Field(
    mocks.dummyVerificationKeyHash
  );
}
```

</file>

<file>

## path: /src/mina-signer/src/signature.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/src/signature.ts

```
import { blake2b } from 'blakejs';
import { Field } from '../../provable/field-bigint.js';
import {
  Group,
  Scalar,
  PrivateKey,
  versionNumbers,
  PublicKey,
} from '../../provable/curve-bigint.js';
import {
  HashInput,
  hashWithPrefix,
  packToFields,
  prefixes,
  Poseidon,
  HashInputLegacy,
  packToFieldsLegacy,
  inputToBitsLegacy,
  HashLegacy,
} from '../../provable/poseidon-bigint.js';
import {
  bitsToBytes,
  bytesToBits,
  record,
  withVersionNumber,
} from '../../bindings/lib/binable.js';
import { base58 } from '../../lib/base58.js';
import { versionBytes } from '../../bindings/crypto/constants.js';
import { Pallas } from '../../bindings/crypto/elliptic_curve.js';

export {
  sign,
  verify,
  signFieldElement,
  verifyFieldElement,
  Signature,
  SignatureJson,
  NetworkId,
  signLegacy,
  verifyLegacy,
  deriveNonce,
};

const networkIdMainnet = 0x01n;
const networkIdTestnet = 0x00n;
type NetworkId = 'mainnet' | 'testnet';
type Signature = { r: Field; s: Scalar };
type SignatureJson = { field: string; scalar: string };

const BinableSignature = withVersionNumber(
  record({ r: Field, s: Scalar }, ['r', 's']),
  versionNumbers.signature
);
const Signature = {
  ...BinableSignature,
  ...base58(BinableSignature, versionBytes.signature),
  toJSON({ r, s }: Signature): SignatureJson {
    return {
      field: Field.toJSON(r),
      scalar: Scalar.toJSON(s),
    };
  },
  fromJSON({ field, scalar }: SignatureJson) {
    let r = Field.fromJSON(field);
    let s = Scalar.fromJSON(scalar);
    return { r, s };
  },
  dummy() {
    return { r: Field(1), s: Scalar(1) };
  },
};

/**
 * Convenience wrapper around {@link sign} where the message is a single {@link Field} element
 */
function signFieldElement(
  message: Field,
  privateKey: PrivateKey,
  networkId: NetworkId
) {
  return sign({ fields: [message] }, privateKey, networkId);
}
/**
 * Convenience wrapper around {@link verify} where the message is a single {@link Field} element
 */
function verifyFieldElement(
  signature: Signature,
  message: Field,
  publicKey: PublicKey,
  networkId: NetworkId
) {
  return verify(signature, { fields: [message] }, publicKey, networkId);
}

/**
 * Schnorr signature algorithm consistent with the OCaml implementation in Schnorr.Chunked.sign, over
 * the Pallas curve with the original "Mina" generator.
 *
 * @see {@link https://github.com/MinaProtocol/mina/blob/develop/docs/specs/signatures/description.md
detailed spec of the algorithm}
 *
 * In contrast to the spec above, this uses the "chunked" style of hash input packing, implemented in {@link
packToFields}.
 *
 * @param message The &nbsp;message&nbsp; can be an arbitrary {@link HashInput}, that can be created
with
 * &nbsp;ProvableExtended<T>.toInput(t)&nbsp; for any provable type &nbsp;T&nbsp;, and by
concatenating multiple hash inputs
 * with {@link HashInput.append}.
 * Currently, we only use the variant {@link signFieldElement} where the message is a single field element,
 * which itself is the result of computing a hash.
 *
 * @param privateKey The &nbsp;privateKey&nbsp; represents an element of the Pallas scalar field, and
should be given as a native bigint.
 * It can be converted from the base58 string representation using {@link PrivateKey.fromBase58}.
 *
 * @param networkId The &nbsp;networkId&nbsp; is either "testnet" or "mainnet" and ensures that testnet
transactions can
 * never be used as valid mainnet transactions.
 *
 * @see {@link deriveNonce} and {@link hashMessage} for details on how the nonce and hash are
computed.
 */
function sign(
  message: HashInput,
  privateKey: PrivateKey,
  networkId: NetworkId
): Signature {
  let publicKey = Group.scale(Group.generatorMina, privateKey);
  let kPrime = deriveNonce(message, publicKey, privateKey, networkId);
  if (Scalar.equal(kPrime, 0n)) throw Error('sign: derived nonce is 0');
  let { x: rx, y: ry } = Group.scale(Group.generatorMina, kPrime);
  let k = Field.isEven(ry) ? kPrime : Scalar.negate(kPrime);
  let e = hashMessage(message, publicKey, rx, networkId);
  let s = Scalar.add(k, Scalar.mul(e, privateKey));
  return { r: rx, s };
}

/**
 * Deterministically derive the nonce for the Schnorr signature algorithm, by:
 * - packing all inputs into a byte array,
 * - applying the [blake2b](https://en.wikipedia.org/wiki/BLAKE_(hash_function)) hash function, and
 * - interpreting the resulting 32 bytes as an element of the Pallas curve scalar field (by dropping bits 254
and 255).
 *
 * @see {@link https://github.com/MinaProtocol/mina/blob/develop/docs/specs/signatures/description.md
detailed spec of the algorithm}
 *
 * In contrast to the spec above, this uses the "chunked" style of hash input packing, implemented in {@link
packToFields}.
 *
 * Input arguments are the same as for {@link sign}, with an additional &nbsp;publicKey&nbsp; (a non-zero,
affine point on the Pallas curve),
 * which &nbsp;sign&nbsp; re-derives by scaling the Pallas "Mina" generator by the
&nbsp;privateKey&nbsp;.
 */
function deriveNonce(
  message: HashInput,
  publicKey: Group,
  privateKey: Scalar,
  networkId: NetworkId
): Scalar {
  let { x, y } = publicKey;
  let d = Field(privateKey);
  let id = networkId === 'mainnet' ? networkIdMainnet : networkIdTestnet;
  let input = HashInput.append(message, {
    fields: [x, y, d],
    packed: [[id, 8]],
  });
  let packedInput = packToFields(input);
  let inputBits = packedInput.map(Field.toBits).flat();
  let inputBytes = bitsToBytes(inputBits);
  let bytes = blake2b(Uint8Array.from(inputBytes), undefined, 32);
  // drop the top two bits to convert into a scalar field element
  // (creates negligible bias because q = 2^254 + eps, eps << q)
  bytes[bytes.length - 1] &= 0x3f;
  return Scalar.fromBytes([...bytes]);
}

/**
 * Hash a message for use by the Schnorr signature algorithm, by:
 * - packing the inputs &nbsp;message&nbsp;, &nbsp;publicKey&nbsp;, &nbsp;r&nbsp; into an array of
Pallas base field elements,
 * - apply a salted hash with the {@link Poseidon} hash function,
 * - interpreting the resulting base field element as a scalar
 *   (which is always possible, and is a no-op, since the scalar field is larger and both fields are represented
with bigints).
 *
 * @see {@link https://github.com/MinaProtocol/mina/blob/develop/docs/specs/signatures/description.md
detailed spec of the algorithm}
 *
 * In contrast to the spec above, this uses the "chunked" style of hash input packing, implemented in {@link
packToFields}.
```

```
 *
 * @param message an arbitrary {@link HashInput}
 * @param publicKey an affine, non-zero point on the Pallas curve, derived by {@link sign} from the private
key
 * @param r an element of the Pallas base field, computed by {@link sign} as the x-coordinate of the
generator, scaled by the nonce.
 * @param networkId either "testnet" or "mainnet", determines the salt (initial state) in the Poseidon hash.
 */
function hashMessage(
  message: HashInput,
  publicKey: Group,
  r: Field,
  networkId: NetworkId
): Scalar {
  let { x, y } = publicKey;
  let input = HashInput.append(message, { fields: [x, y, r] });
  let prefix =
    networkId === 'mainnet'
      ? prefixes.signatureMainnet
      : prefixes.signatureTestnet;
  return hashWithPrefix(prefix, packToFields(input));
}

/**
 * Verifies a signature created by {@link sign}, returns &nbsp;true&nbsp; if (and only if) the signature is valid.
 *
 * @see {@link https://github.com/MinaProtocol/mina/blob/develop/docs/specs/signatures/description.md
detailed spec of the algorithm}
 *
 * In contrast to the spec above, this uses the "chunked" style of hash input packing, implemented in {@link
packToFields}.
 *
 * @param publicKey the public key has to be passed in as a compressed {@link PublicKey}.
 * It can be created from a base58 string with {@link PublicKey.fromBase58}.
 */
function verify(
  signature: Signature,
  message: HashInput,
  publicKey: PublicKey,
  networkId: NetworkId
) {
  let { r, s } = signature;
  let pk = PublicKey.toGroup(publicKey);
  let e = hashMessage(message, pk, r, networkId);
  let { scale, one, sub } = Pallas;
  let R = sub(scale(one, s), scale(Group.toProjective(pk), e));
  try {
    // if &nbsp;R&nbsp; is infinity, Group.fromProjective throws an error, so &nbsp;verify&nbsp; returns false
    let { x: rx, y: ry } = Group.fromProjective(R);
    return Field.isEven(ry) && Field.equal(rx, r);
  } catch {
    return false;
  }
}

// legacy signatures

/**
 * Same as {@link sign}, but using the "legacy" style of hash input packing.
 */
function signLegacy(
  message: HashInputLegacy,
  privateKey: PrivateKey,
  networkId: NetworkId
): Signature {
  let publicKey = Group.scale(Group.generatorMina, privateKey);
  let kPrime = deriveNonceLegacy(message, publicKey, privateKey, networkId);
  if (Scalar.equal(kPrime, 0n)) throw Error('sign: derived nonce is 0');
  let { x: rx, y: ry } = Group.scale(Group.generatorMina, kPrime);
  let k = Field.isEven(ry) ? kPrime : Scalar.negate(kPrime);
  let e = hashMessageLegacy(message, publicKey, rx, networkId);
  let s = Scalar.add(k, Scalar.mul(e, privateKey));
  return { r: rx, s };
}

/**
 * Same as {@link verify}, but using the "legacy" style of hash input packing.
 */
function verifyLegacy(
  signature: Signature,
  message: HashInputLegacy,
  publicKey: PublicKey,
  networkId: NetworkId
) {
  try {
    let { r, s } = signature;
    let pk = PublicKey.toGroup(publicKey);
    let e = hashMessageLegacy(message, pk, r, networkId);
    let { scale, one, sub } = Pallas;
    let R = sub(scale(one, s), scale(Group.toProjective(pk), e));
    // if &nbsp;R&nbsp; is infinity, Group.fromProjective throws an error, so &nbsp;verify&nbsp; returns false
    let { x: rx, y: ry } = Group.fromProjective(R);
    return Field.isEven(ry) && Field.equal(rx, r);
  } catch {
    return false;
  }
}

/**
 * Same as {@link deriveNonce}, but using the "legacy" style of hash input packing.
 */
function deriveNonceLegacy(
  message: HashInputLegacy,
  publicKey: Group,
  privateKey: Scalar,
  networkId: NetworkId
): Scalar {
  let { x, y } = publicKey;
  let scalarBits = Scalar.toBits(privateKey);
  let id = networkId === 'mainnet' ? networkIdMainnet : networkIdTestnet;
  let idBits = bytesToBits([Number(id)]);
  let input = HashInputLegacy.append(message, {
    fields: [x, y],
    bits: [...scalarBits, ...idBits],
  });
  let inputBits = inputToBitsLegacy(input);
  let inputBytes = bitsToBytes(inputBits);
  let bytes = blake2b(Uint8Array.from(inputBytes), undefined, 32);
  // drop the top two bits to convert into a scalar field element
  // (creates negligible bias because q = 2^254 + eps, eps << q)
  bytes[bytes.length - 1] &= 0x3f;
  return Scalar.fromBytes([...bytes]);
}

/**
 * Same as {@link hashMessage}, except for two differences:
 * - uses the "legacy" style of hash input packing.
 * - uses Poseidon with "legacy" parameters for hashing
 *
 * The method produces a hash in the Pallas base field ({@link Field}) and reinterprets it as a {@link Scalar}.
 * This is possible, and a no-op, since the scalar field is larger and both fields are represented with bigints.
 */
function hashMessageLegacy(
  message: HashInputLegacy,
  publicKey: Group,
  r: Field,
  networkId: NetworkId
): Scalar {
  let { x, y } = publicKey;
  let input = HashInputLegacy.append(message, { fields: [x, y, r], bits: [] });
  let prefix =
    networkId === 'mainnet'
      ? prefixes.signatureMainnet
      : prefixes.signatureTestnet;
  return HashLegacy.hashWithPrefix(prefix, packToFieldsLegacy(input));
}
```

</file>

<file>

## path: /src/mina-signer/src/signature.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/src/signature.unit-test.ts

```
// unit tests dedicated to testing consistency of the signature algorithm
import { expect } from 'expect';
import {
  sign,
  Signature,
  signFieldElement,
  verify,
  verifyFieldElement,
} from './signature.js';
import { Ledger, Test } from '../../snarky.js';
import { Field as FieldSnarky } from '../../lib/core.js';
import { Field } from '../../provable/field-bigint.js';
import { PrivateKey, PublicKey } from '../../provable/curve-bigint.js';
import { PrivateKey as PrivateKeySnarky } from '../../lib/signature.js';
import { p } from '../../bindings/crypto/finite_field.js';
import { AccountUpdate } from '../../bindings/mina-transaction/gen/transaction-bigint.js';
import { HashInput } from '../../bindings/lib/provable-bigint.js';
import { Ml } from '../../lib/ml/conversion.js';
import { FieldConst } from '../../lib/field.js';

// check consistency with OCaml, where we expose the function to sign 1 field element with "testnet"
function checkConsistentSingle(
  msg: Field,
  key: PrivateKey,
  keySnarky: PrivateKeySnarky,
  pk: PublicKey
) {
  let sigTest = signFieldElement(msg, key, 'testnet');
  let sigMain = signFieldElement(msg, key, 'mainnet');
  // verify
  let okTestnetTestnet = verifyFieldElement(sigTest, msg, pk, 'testnet');
  let okMainnetTestnet = verifyFieldElement(sigMain, msg, pk, 'testnet');
  let okTestnetMainnet = verifyFieldElement(sigTest, msg, pk, 'mainnet');
  let okMainnetMainnet = verifyFieldElement(sigMain, msg, pk, 'mainnet');
  expect(okTestnetTestnet).toEqual(true);
  expect(okMainnetTestnet).toEqual(false);
  expect(okTestnetMainnet).toEqual(false);
  expect(okMainnetMainnet).toEqual(true);
  // consistent with OCaml
  let msgMl = FieldConst.fromBigint(msg);
  let keyMl = Ml.fromPrivateKey(keySnarky);
  let actualTest = Test.signature.signFieldElement(msgMl, keyMl, false);
  let actualMain = Test.signature.signFieldElement(msgMl, keyMl, true);
  expect(Signature.toBase58(sigTest)).toEqual(actualTest);
  expect(Signature.toBase58(sigMain)).toEqual(actualMain);
}

// check that various multi-field hash inputs can be verified
function checkCanVerify(msg: HashInput, key: PrivateKey, pk: PublicKey) {
  let sigTest = sign(msg, key, 'testnet');
  let sigMain = sign(msg, key, 'mainnet');
  // verify
  let okTestnetTestnet = verify(sigTest, msg, pk, 'testnet');
  let okMainnetTestnet = verify(sigMain, msg, pk, 'testnet');
  let okTestnetMainnet = verify(sigTest, msg, pk, 'mainnet');
  let okMainnetMainnet = verify(sigMain, msg, pk, 'mainnet');
  expect(okTestnetTestnet).toEqual(true);
  expect(okMainnetTestnet).toEqual(false);
  expect(okTestnetMainnet).toEqual(false);
  expect(okMainnetMainnet).toEqual(true);
}

// created with PrivateKey.random(); hard-coded to make tests reproducible
let privateKeys = [
  5168137537350106646038172092273964439447230860125519060122927580020622270141n,
  14928086762226019830440257243808305601005302250478321528858262459356286366453n,
  11143322587625020388344946433171679426319801593192223014315467772089286592604n,
  18906421803290277088414121591868716174001000360632913534629968674606246464596n,
  5925568526516417064863153411603894796335750603303676942163464712022427664721n,
  20037223684845539907201171426637539137161967796359729867193035014844512841185n,
  20236646771171834616838432888269706094783740262871406794713329156173567092679n,
  13507121957360975407754936805517309719995432919287716894090189496158972691353n,
  19152183577529784585604205414612388832924448123778079866321952213819564835523n,
  20617898261860919949586898353988281719931035849127590497832022423723702051831n,
];
// created with Field.random(); hard-coded to make tests reproducible
let randomFields = [
  7612970850134051471932333398465838351839532419148061934628866168663280504034n,
  27972698397543914557828288754101805907710482736523928428688840423684487283686n,
  8703743676147621684744466323325623278489059672232298069375861211469834922688n,
  16192824873102077219220893637595957128993235636019312988127478669820406609509n,
  5123429362863871987559144871409804453562526127381116452870491690397753811094n,
  25971592550162911171172249177105774374832521772676232623383824977784210633515n,
  16430178824936223386420564202153148454359399753677109773596670870194786964396n,
  19157956145170718958732174379270969488704624188346232665215657585142941265622n,
  28486023928659548119979958179075614860483912203433036340088142691910674902557n,
  14367789165599062779555941943909225438752186664081151509427474521285809812186n,
];

// check with different random private keys
for (let i = 0; i < 10; i++) {
  let key = privateKeys[i];
  let keySnarky = PrivateKeySnarky.fromBase58(PrivateKey.toBase58(key));
  let publicKey = PrivateKey.toPublicKey(key);

  // hard coded single field elements
  let hardcoded = [0n, 1n, 2n, p - 1n];
  for (let x of hardcoded) {
    checkConsistentSingle(x, key, keySnarky, publicKey);
  }
  // random single field elements
  for (let i = 0; i < 10; i++) {
    let x = randomFields[i];
    checkConsistentSingle(x, key, keySnarky, publicKey);
  }
  // hard-coded multi-element hash inputs
  let messages: HashInput[] = [
    { fields: [0n, 0n, 0n] },
    { fields: [4n, 20n, 120398120n] },
    {
     fields: [1n, p - 1n],
     packed: [
      [0n, 0],
      [1n, 1],
     ],
    },
    {
     packed: [
      [0xffn, 8],
      [0xffffn, 16],
      [0xffff_ffffn, 32],
      [0xffff_ffff_ffff_ffffn, 64],
     ],
    },
    AccountUpdate.toInput(AccountUpdate.emptyValue()),
  ];
  for (let msg of messages) {
    checkCanVerify(msg, key, publicKey);
  }
}

console.log(
  "signatures are consistent or verify / don't verify as expected!     "
);
```

</file>

<file>

## path: /src/mina-signer/src/test-vectors/accountUpdate.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/src/test-vectors/accountUpdate.ts

```
import * as Json from '../../../bindings/mina-transaction/gen/transaction-json.js';
import { mocks } from '../../../bindings/crypto/constants.js';

export { accountUpdateExample };

// an example account update, to be used for tests
let accountUpdateExample: Json.AccountUpdate = {
 body: {
  publicKey: 'B62qmfmZrxjfRHfnx1QJLHUyStQxSkqao9civMXPaymkknX5PCiZT7J',
  tokenId: 'yEWUTZqtT8PmCFU32EXwCwudK4gtxCWkjcAC7eTwj2riWhCV8M',
  update: {
    appState: ['9', null, null, null, null, null, null, null],
    delegate: 'B62qrja1a2wu3ciKygrqNiNoDZUsHCcE1VfF4LZQtQkzszWhogpWN9i',
    verificationKey: null,
    permissions: {
      editState: 'Proof',
      access: 'None',
      send: 'Signature',
      receive: 'Proof',
      setDelegate: 'Signature',
      setPermissions: 'None',
      setVerificationKey: 'None',
      setZkappUri: 'Signature',
      editActionState: 'Proof',
      setTokenSymbol: 'Signature',
      incrementNonce: 'Signature',
      setVotingFor: 'Signature',
      setTiming: 'Signature',
    },
    zkappUri: null,
    tokenSymbol: 'BLABLA',
    timing: {
      initialMinimumBalance: '1',
      cliffTime: '0',
      cliffAmount: '0',
      vestingPeriod: '1',
      vestingIncrement: '2',
    },
    votingFor: null,
  },
  balanceChange: { magnitude: '14197832', sgn: 'Negative' },
  incrementNonce: true,
  events: [['0'], ['1']],
  actions: [['0'], ['1']],
  callData:
    '6743900749438632952963252074409706338210982229126682817949490928992849119219',
  callDepth: 0,
  preconditions: {
    network: {
      snarkedLedgerHash: null,
      blockchainLength: null,
      minWindowDensity: null,
      totalCurrency: null,
      globalSlotSinceGenesis: null,
      stakingEpochData: {
        ledger: {
          hash:
'4295928848099762379149452702606274128891023958431976727769309015818325653869',
          totalCurrency: null,
        },
        seed: null,
        startCheckpoint: null,
        lockCheckpoint: null,
        epochLength: null,
      },
      nextEpochData: {
        ledger: { hash: null, totalCurrency: null },
        seed: null,
        startCheckpoint: null,
        lockCheckpoint:
          '16957731668585847663441468154039306422576952181094510426739468515732343321592',
        epochLength: null,
      },
    },
    account: {
      balance: { lower: '1000000000', upper: '1000000000' },
      nonce: null,
      receiptChainHash: null,
      delegate: 'B62qrja1a2wu3ciKygrqNiNoDZUsHCcE1VfF4LZQtQkzszWhogpWN9i',
      state: ['9', null, null, null, null, null, null, null],
      actionState: null,
      provedState: null,
      isNew: true,
    },
    validWhile: null,
  },
  useFullCommitment: false,
  implicitAccountCreationFee: false,
  mayUseToken: { parentsOwnToken: false, inheritFromParent: false },
  authorizationKind: {
    isSigned: false,
    isProved: false,
    verificationKeyHash: mocks.dummyVerificationKeyHash,
  },
},
authorization: { proof: null, signature: null },
};
```

</file>

<file>

## path: /src/mina-signer/src/test-vectors/legacySignatures.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/src/test-vectors/legacySignatures.ts

```
import { DelegationJson, PaymentJson } from '../sign-legacy.js';

// inputs and generated signatures from client_sdk/tests/test_signatures.js
export { signatures, payments, delegations, strings, keypair };

let keypair = {
  privateKey: 'EKFKgDtU3rcuFTVSEpmpXSkukjmX4cKefYREi6Sdsk7E7wsT7KRw',
  publicKey: 'B62qiy32p8kAKnny8ZFwoMhYpBppM1DWVCqAPBYNcXnsAHhnfAAuXgg',
};

let receiver = 'B62qrcFstkpqXww1EkSGrqMCwCNho86kuqBd4FrAAUsPxNKdiPzAUsy';

let newDelegate = 'B62qkfHpLpELqpMK6ZvUTJ5wRqKDRF3UHyJ4Kv3FU79Sgs4qpBnx5RR';

let payments: PaymentJson[] = [
 {
   body: { receiver, amount: '42' },
   common: {
     fee: '3',
     feePayer: keypair.publicKey,
     nonce: '200',
     validUntil: '10000',
     memo: 'this is a memo',
   },
 },
 {
   body: { receiver, amount: '2048' },
   common: {
     fee: '15',
     feePayer: keypair.publicKey,
     nonce: '212',
     validUntil: '305',
     memo: 'this is not a pipe',
   },
 },
 {
   body: { receiver, amount: '109' },
   common: {
     fee: '2001',
     feePayer: keypair.publicKey,
     nonce: '3050',
     validUntil: '9000',
     memo: 'blessed be the geek',
   },
 },
];

let delegations: DelegationJson[] = [
 {
   body: { newDelegate },
   common: {
     fee: '3',
     feePayer: keypair.publicKey,
     nonce: '10',
     validUntil: '4000',
     memo: 'more delegates, more fun',
   },
 },
 {
   body: { newDelegate },
   common: {
     fee: '10',
     feePayer: keypair.publicKey,
     nonce: '1000',
     validUntil: '8192',
     memo: 'enough stake to kill a vampire',
   },
 },
 {
   body: { newDelegate },
   common: {
     fee: '8',
     feePayer: keypair.publicKey,
     nonce: '1010',
     validUntil: '100000',
     memo: 'another memo',
   },
 },
];

let strings = [
 'this is a test',
 'this is only a test',
 'if this had been an actual emergency...',
];

/**
 * for each network (testnet, mainnet), these are the signatures for
 * - the 3 payments,
 * - the 3 stake delegations,
 * - the 3 strings.
 */
let signatures = {
 testnet: [
   {
     field:
       '3925887987173883783388058255268083382298769764463609405200521482763932632383',
     scalar:
       '445615701481226398197189554290689546503290167815530435382795701939759548136',
   },
   {
     field:
       '11838925242791061185900891854974280922359055483441419242429642295065318643984',
     scalar:
       '5057044820006008308046028014628135487302791372585541488835641418654652928805',
   },
   {
     field:
       '13570419670106759824217358880396743605262660069048455950202130815805728575057',
     scalar:
       '2256128221267944805514947515637443480133552241968312777663034361688965989223',
   },
   {
     field:
       '18603328765572408555868399359399411973012220541556204196884026585115374044583',
     scalar:
       '17076342019359061119005549736934690084415105419939473687106079907606137611470',
   },
   {
     field:
       '1786373894608285187089973929748850875336413409295396991315429715474432640801',
     scalar:
       '10435258496141097615588833319454104720521911644724923418749752896069542389757',
   },
   {
     field:
       '11710586766419351067338319607483640291676872446372400739329190129174446858072',
     scalar:
       '21663533922934564101122062377096487451020504743791218020915919810997397884837',
   },
   {
     field:
       '11583775536286847540414661987230057163492736306749717851628536966882998258109',
     scalar:
       '14787360096063782022566783796923142259879388947509616216546009448340181956495',
   },
   {
     field:
       '24809097509137086694730479515383937245108109696879845335879579016397403384488',
     scalar:
       '23723859937408726087117568974923795978435877847592289069941156359435022279156',
   },
   {
     field:
       '23803497755408154859878117448681790665144834176143832235351783889976460433296',
     scalar:
       '21219917886278462345652813021708727397787183083051040637716760620250038837684',
   },
 ],
 mainnet: [
   {
     field:
       '2290465734865973481454975811990842289349447524565721011257265781466170720513',
     scalar:
       '174718295375042423373378066296864207343460524320417038741346483351503066865',
   },
   {
     field:
       '3338221378196321618737404652850173545830741260219426985985110494623248154796',
     scalar:
       '13582570889626737053936904045130069988029386067840542224501137534361543053466',
   },
   {
     field:
       '24977166875850415387591601609169744956874881328889802588427412550673368014171',
     scalar:
       '8818176737844714163963728742657256399283959917269715546724011366788373936767',
   },
   {
     field:
       '18549185720796945285997801022505868190780742636917696085321477383695464941808',
     scalar:
       '9968155560235917784839059154575307851833761552720670659405850314060739412758',
   },
   {
     field:
       '27435277901837444378602251759261698832749786010721792798570593506489878524054',
     scalar:
       '5303814070856978976450674139278204752713705309497875510553816988969674317908',
   },
   {
     field:
       '18337925798749632162999573213504280894403810378974021233452576035581180265108',
     scalar:
       '17033350386680878193188260707518516061312646961349757526930471244219909355133',
   },
   {
     field:
       '15321026181887258084717253351692625217563887132804118766475695975434200286072',
     scalar:
       '27693688834009297019754701709097142916828669707451033859732637861400085816575',
   },
   {
     field:
       '7389839717736616673468176670823346848621475008909123730960586617430930011362',
     scalar:
       '16812002169649926565884427604872242188288298244442130642661893463581998776079',
   },
   {
     field:
       '25237307917208237775896283358517786348974681409860182331969894401303358790178',
     scalar:
       '1498643894425942815773348600211341433686244249442354387056510209608647184582',
   },
 ],
};
```

</file>

<file>

## path: /src/mina-signer/src/transaction-hash.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/src/transaction-hash.ts

```
import { Bool, Field, UInt32, UInt64 } from '../../provable/field-bigint.js';
import {
  Binable,
  BinableString,
  BinableUint64,
  BinableUint32,
  defineBinable,
  enumWithArgument,
  record,
  stringToBytes,
  withVersionNumber,
} from '../../bindings/lib/binable.js';
import {
  Common,
  Delegation,
  Payment,
  UserCommand,
  UserCommandEnum,
  PaymentJson,
  DelegationJson,
  delegationFromJson,
  paymentFromJson,
} from './sign-legacy.js';
import { PublicKey, Scalar } from '../../provable/curve-bigint.js';
import { Signature, SignatureJson } from './signature.js';
import { blake2b } from 'blakejs';
import { base58, withBase58 } from '../../lib/base58.js';
import { versionBytes } from '../../bindings/crypto/constants.js';

export {
  hashPayment,
  hashStakeDelegation,
  SignedCommand,
  SignedCommandV1,
  Common,
  userCommandToEnum,
  userCommandToV1,
  Signed,
  SignedLegacy,
  HashBase58,
};

type Signed<T> = { data: T; signature: string };
type SignedLegacy<T> = { data: T; signature: SignatureJson };
const dummySignature: Signature = { r: Field(1), s: Scalar(1) };

function hashPayment(
  signed: SignedLegacy<PaymentJson>,
  { berkeley = false } = {}
) {
  if (!berkeley) return hashPaymentV1(signed);
  let payload = userCommandToEnum(paymentFromJson(signed.data));
  return hashSignedCommand({
    signer: PublicKey.fromBase58(signed.data.common.feePayer),
    signature: dummySignature,
    payload,
  });
}

function hashStakeDelegation(
  signed: SignedLegacy<DelegationJson>,
  { berkeley = false } = {}
) {
  if (!berkeley) return hashStakeDelegationV1(signed);
  let payload = userCommandToEnum(delegationFromJson(signed.data));
  return hashSignedCommand({
    signer: PublicKey.fromBase58(signed.data.common.feePayer),
    signature: dummySignature,
    payload,
  });
}

function hashSignedCommand(command: SignedCommand) {
  let inputBytes = SignedCommand.toBytes(command);
  let bytes = blake2b(Uint8Array.from(inputBytes), undefined, 32);
  return HashBase58.toBase58(bytes);
}

// helper

function userCommandToEnum({ common, body }: UserCommand): UserCommandEnum {
  let { tag: type, ...value } = body;
  switch (type) {
    case 'Payment':
      return { common, body: { type, value: { receiver: body.receiver, amount: body.amount } } };
    case 'StakeDelegation':
      let { receiver: newDelegate } = value;
      return {
        common,
        body: {
          type,
          value: { type: 'SetDelegate', value: { newDelegate } },
        },
      };
  }
}

// binable

let BinablePublicKey = record({ x: Field, isOdd: Bool }, ['x', 'isOdd']);
type GlobalSlotSinceGenesis = Common['validUntil'];
let GlobalSlotSinceGenesis = enumWithArgument<[GlobalSlotSinceGenesis]>([
  { type: 'SinceGenesis', value: BinableUint32 },
]);

const Common = record<Common>(
  {
    fee: BinableUint64,
    feePayer: BinablePublicKey,
    nonce: BinableUint32,
    validUntil: GlobalSlotSinceGenesis,
    memo: BinableString,
  },
  ['fee', 'feePayer', 'nonce', 'validUntil', 'memo']
);
const Payment = record<Payment>(
  {
    receiver: BinablePublicKey,
    amount: BinableUint64,
  },
  ['receiver', 'amount']
);
const Delegation = record<Delegation>(
  { newDelegate: BinablePublicKey },
  ['newDelegate']
);
type DelegationEnum = { type: 'SetDelegate'; value: Delegation };
const DelegationEnum = enumWithArgument<[DelegationEnum]>([
  { type: 'SetDelegate', value: Delegation },
]);

const Body = enumWithArgument<
  [
    { type: 'Payment'; value: Payment },
    { type: 'StakeDelegation'; value: DelegationEnum }
  ]
>([
  { type: 'Payment', value: Payment },
  { type: 'StakeDelegation', value: DelegationEnum },
]);

const UserCommand = record({ common: Common, body: Body }, ['common', 'body']);
const BinableSignature = record({ r: Field, s: Scalar }, ['r', 's']);

type SignedCommand = {
  payload: UserCommandEnum;
  signer: PublicKey;
  signature: Signature;
};
const SignedCommand = record<SignedCommand>(
  {
    payload: UserCommand,
    signer: BinablePublicKey,
    signature: BinableSignature,
  },
  ['payload', 'signer', 'signature']
);

const HashBase58 = base58(
  defineBinable<Uint8Array>({
    toBytes(t: Uint8Array) {
      return [t.length, ...t];
    },
    readBytes(bytes) {
      return [Uint8Array.from(bytes.slice(1)), bytes.length];
    },
  }),
  versionBytes.transactionHash
);

// legacy / v1 stuff

function hashPaymentV1({ data, signature }: SignedLegacy<PaymentJson>) {
  let paymentV1 = userCommandToV1(paymentFromJson(data));
  return hashSignedCommandV1({
    signer: PublicKey.fromBase58(data.common.feePayer),
    signature: Signature.fromJSON(signature),
    payload: paymentV1,
  });
}

function hashStakeDelegationV1({
  data,
  signature,
}: SignedLegacy<DelegationJson>) {
  let payload = userCommandToV1(delegationFromJson(data));
  return hashSignedCommandV1({
    signer: PublicKey.fromBase58(data.common.feePayer),
    signature: Signature.fromJSON(signature),
    payload,
  });
}

function hashSignedCommandV1(command: SignedCommandV1) {
  let base58 = SignedCommandV1.toBase58(command);
  let inputBytes = stringToBytes(base58);
  let bytes = blake2b(Uint8Array.from(inputBytes), undefined, 32);
  return HashBase58.toBase58(bytes);
}

function userCommandToV1({ common, body }: UserCommand): UserCommandV1 {
  let { tag: type, ...value } = body;
  let commonV1: CommonV1 = {
    ...common,
    validUntil: common.validUntil.value,
    feeToken: 1n,
  };
  switch (type) {
    case 'Payment':
      let paymentV1: PaymentV1 = { ...value, tokenId: 1n };
      return { common: commonV1, body: { type, value: paymentV1 } };
    case 'StakeDelegation':
      let { source: delegator, receiver: newDelegate } = value;
      return {
        common: commonV1,
        body: {
          type,
          value: { type: 'SetDelegate', value: { delegator, newDelegate } },
        },
      };
  }
}
// binables for v1 signed commands

// TODO: Version numbers (of 1) were placed somewhat arbitrarily until it worked / matched serializations
from OCaml.
// I couldn't precisely explain each of them from following the OCaml type annotations, which I find hard to
parse.
// You could get an equivalent serialization by moving, for example, one of the version numbers on
&nbsp;common&nbsp; one level down to become
// another version number on &nbsp;fee&nbsp;, and I'm not sure what the correct answer is. I think this
doesn't matter because
// the type layout here, including version numbers, is frozen, so if it works once it'll work forever.
const with1 = <T>(binable: Binable<T>) => withVersionNumber(binable, 1);
const Uint64V1 = with1(with1(BinableUint64));
const Uint32V1 = with1(with1(BinableUint32));
type CommonV1 = {
  fee: UInt64;
  feePayer: PublicKey;
  nonce: UInt32;
  validUntil: UInt32;
  memo: string;
  feeToken: UInt64;
};

const CommonV1 = with1(
  with1(
    record<CommonV1>(
      {
        fee: with1(Uint64V1),
        feeToken: with1(Uint64V1),
        feePayer: PublicKey,
        nonce: Uint32V1,
        validUntil: Uint32V1,
        memo: with1(BinableString),
      },
      ['fee', 'feeToken', 'feePayer', 'nonce', 'validUntil', 'memo']
    )
  )
```

```
);
type PaymentV1 = Payment & { source: PublicKey, tokenId: UInt64 };
const PaymentV1 = with1(
  with1(
    record<PaymentV1>(
      {
        source: PublicKey,
        receiver: PublicKey,
        tokenId: Uint64V1,
        amount: with1(Uint64V1),
      },
      ['source', 'receiver', 'tokenId', 'amount']
    )
  )
);
type DelegationV1 = Delegation & { delegator: PublicKey };
const DelegationV1 = record<DelegationV1>(
  { delegator: PublicKey, newDelegate: PublicKey },
  ['delegator', 'newDelegate']
);
type DelegationEnumV1 = { type: 'SetDelegate'; value: DelegationV1 };
const DelegationEnumV1 = with1(
  enumWithArgument<[DelegationEnumV1]>([
    { type: 'SetDelegate', value: DelegationV1 },
  ])
);
type BodyV1 =
  | { type: 'Payment'; value: PaymentV1 }
  | { type: 'StakeDelegation'; value: DelegationEnumV1 };
const BodyV1 = with1(
  enumWithArgument<
    [
      { type: 'Payment'; value: PaymentV1 },
      { type: 'StakeDelegation'; value: DelegationEnumV1 }
    ]
  >([
    { type: 'Payment', value: PaymentV1 },
    { type: 'StakeDelegation', value: DelegationEnumV1 },
  ])
);
type UserCommandV1 = { common: CommonV1; body: BodyV1 };
const UserCommandV1 = with1(
  record<UserCommandV1>({ common: CommonV1, body: BodyV1 }, ['common', 'body'])
);
type SignedCommandV1 = {
  payload: UserCommandV1;
  signer: PublicKey;
  signature: Signature;
};
const SignedCommandV1 = withBase58<SignedCommandV1>(
  with1(
    with1(
    record(
      {
        payload: UserCommandV1,
        signer: with1(PublicKey),
        signature: with1(record({ r: with1(Field), s: Scalar }, ['r', 's'])),
      },
      ['payload', 'signer', 'signature']
    )
  )
  ),
  versionBytes.signedCommandV1
);
```

</file>

<file>

## path: /src/mina-signer/src/transaction-hash.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/src/transaction-hash.unit-test.ts

```
import { Ledger, Test } from '../../snarky.js';
import {
  Common,
  hashPayment,
  hashStakeDelegation,
  SignedCommand,
  SignedCommandV1,
  SignedLegacy,
  userCommandToEnum,
  userCommandToV1,
} from './transaction-hash.js';
import {
  PaymentJson,
  PaymentJsonV1,
  commonFromJson,
  paymentFromJson,
  CommonJson,
  DelegationJson,
  DelegationJsonV1,
  delegationFromJson,
} from './sign-legacy.js';
import { Signature, SignatureJson } from './signature.js';
import { PublicKey } from '../../provable/curve-bigint.js';
import { Memo } from './memo.js';
import { expect } from 'expect';
import { versionBytes } from '../../bindings/crypto/constants.js';
import { test } from '../../lib/testing/property.js';
import { RandomTransaction } from './random-transaction.js';

test(
RandomTransaction.signedPayment,
RandomTransaction.signedDelegation,
(payment, delegation) => {
  // common serialization
  let result = Test.transactionHash.serializeCommon(
    JSON.stringify(commonToOcaml(payment.data.common))
  );
  let bytes0 = [...result.data];
  let common = commonFromJson(payment.data.common);
  let bytes1 = Common.toBytes(common);
  expect(JSON.stringify(bytes1)).toEqual(JSON.stringify(bytes0));

  // payment serialization
  let ocamlPayment = JSON.stringify(paymentToOcaml(payment));
  result = Test.transactionHash.serializePayment(ocamlPayment);
  let paymentBytes0 = [...result.data];
  let payload = userCommandToEnum(paymentFromJson(payment.data));
  let command = {
    signer: PublicKey.fromBase58(payment.data.common.feePayer),
    signature: Signature.fromJSON(payment.signature),
    payload,
  };
  let paymentBytes1 = SignedCommand.toBytes(command);
  expect(JSON.stringify(paymentBytes1)).toEqual(
    JSON.stringify(paymentBytes0)
  );

  // payment roundtrip
  let commandRecovered = SignedCommand.fromBytes(paymentBytes1);
  expect(commandRecovered).toEqual(command);

  // payment hash
  let digest0 = Test.transactionHash.hashPayment(ocamlPayment);
  let digest1 = hashPayment(payment, { berkeley: true });
  expect(digest1).toEqual(digest0);

  // delegation serialization
  let ocamlDelegation = JSON.stringify(delegationToOcaml(delegation));
  result = Test.transactionHash.serializePayment(ocamlDelegation);
  let delegationBytes0 = [...result.data];
  payload = userCommandToEnum(delegationFromJson(delegation.data));
  command = {
    signer: PublicKey.fromBase58(delegation.data.common.feePayer),
    signature: Signature.fromJSON(delegation.signature),
    payload,
  };
  let delegationBytes1 = SignedCommand.toBytes(command);
  expect(JSON.stringify(delegationBytes1)).toEqual(
    JSON.stringify(delegationBytes0)
  );

  // delegation roundtrip
```

```
    commandRecovered = SignedCommand.fromBytes(delegationBytes1);
    expect(commandRecovered).toEqual(command);

    // delegation hash
    digest0 = Test.transactionHash.hashPayment(ocamlDelegation);
    digest1 = hashStakeDelegation(delegation, { berkeley: true });
    expect(digest1).toEqual(digest0);

    // payment v1 serialization
    let ocamlPaymentV1 = JSON.stringify(paymentToOcamlV1(payment));
    let ocamlBase58V1 = Test.transactionHash.serializePaymentV1(ocamlPaymentV1);
    let v1Bytes0 = stringToBytesOcaml(
      Test.encoding.ofBase58(ocamlBase58V1, versionBytes.signedCommandV1).c
    );
    let paymentV1Body = userCommandToV1(paymentFromJson(payment.data));
    let paymentV1 = {
      signer: PublicKey.fromBase58(payment.data.common.feePayer),
      signature: Signature.fromJSON(payment.signature),
      payload: paymentV1Body,
    };
    let v1Bytes1 = SignedCommandV1.toBytes(paymentV1);
    expect(JSON.stringify(v1Bytes1)).toEqual(JSON.stringify(v1Bytes0));

    // payment v1 hash
    digest0 = Test.transactionHash.hashPaymentV1(ocamlPaymentV1);
    digest1 = hashPayment(payment);
    expect(digest1).toEqual(digest0);

    // delegation v1 serialization
    let ocamlDelegationV1 = JSON.stringify(delegationToOcamlV1(delegation));
    ocamlBase58V1 = Test.transactionHash.serializePaymentV1(ocamlDelegationV1);
    v1Bytes0 = stringToBytesOcaml(
      Test.encoding.ofBase58(ocamlBase58V1, versionBytes.signedCommandV1).c
    );
    let delegationV1Body = userCommandToV1(delegationFromJson(delegation.data));
    let delegationV1 = {
      signer: PublicKey.fromBase58(delegation.data.common.feePayer),
      signature: Signature.fromJSON(delegation.signature),
      payload: delegationV1Body,
    };
    v1Bytes1 = SignedCommandV1.toBytes(delegationV1);
    expect(JSON.stringify(v1Bytes1)).toEqual(JSON.stringify(v1Bytes0));

    // delegation v1 hash
    digest0 = Test.transactionHash.hashPaymentV1(ocamlDelegationV1);
    digest1 = hashStakeDelegation(delegation);
    expect(digest1).toEqual(digest0);
  }
);

// negative tests
```

```
test.negative(RandomTransaction.signedPayment.invalid!, (payment) =>
  hashPayment(payment)
);
test.negative(RandomTransaction.signedPayment.invalid!, (payment) => {
  hashPayment(payment, { berkeley: true });
  // for "berkeley" hashing, it's fine if the signature is invalid because it's not part of the hash
  // => make invalid signatures fail independently
  Signature.fromJSON(payment.signature);
});
test.negative(RandomTransaction.signedDelegation.invalid!, (delegation) =>
  hashStakeDelegation(delegation)
);
test.negative(RandomTransaction.signedDelegation.invalid!, (delegation) => {
  hashStakeDelegation(delegation, { berkeley: true });
  // for "berkeley" hashing, it's fine if the signature is invalid because it's not part of the hash
  // => make invalid signatures fail independently
  Signature.fromJSON(delegation.signature);
});

function paymentToOcaml({
  data: {
    common,
    body: { receiver, amount },
  },
  signature,
}: SignedLegacy<PaymentJson>) {
  return {
    payload: {
      common: commonToOcaml(common),
      body: ['Payment', { receiver_pk: receiver, amount }],
    },
    signer: common.feePayer,
    signature: signatureToOCaml(signature),
  };
}

function paymentToOcamlV1({
  data: {
    common,
    body: { receiver, amount },
  },
  signature,
}: SignedLegacy<PaymentJson>) {
  return {
    payload: {
      common: commonToOcamlV1(common),
      body: [
        'Payment',
        { source_pk: common.feePayer, receiver_pk: receiver, amount, token_id: '1' },
      ],
    },
    signer: common.feePayer,
    signature: signatureToOCaml(signature),
  };
}

function delegationToOcaml({
  data: {
    common,
    body: { newDelegate },
  },
  signature,
}: SignedLegacy<DelegationJson>) {
  return {
    payload: {
      common: commonToOcaml(common),
      body: [
        'Stake_delegation',
        ['Set_delegate', { new_delegate: newDelegate }],
      ],
    },
    signer: common.feePayer,
    signature: signatureToOCaml(signature),
  };
}

function delegationToOcamlV1({
  data: {
    common,
    body: { newDelegate },
  },
  signature,
}: SignedLegacy<DelegationJson>) {
  return {
    payload: {
      common: commonToOcamlV1(common),
      body: [
        'Stake_delegation',
        ['Set_delegate', { delegator: common.feePayer, new_delegate: newDelegate }],
      ],
    },
    signer: common.feePayer,
    signature: signatureToOCaml(signature),
  };
}

function commonToOcaml({ fee, feePayer, nonce, validUntil, memo }: CommonJson) {
  memo = Memo.toBase58(Memo.fromString(memo));
  return {
    fee: fee === '0' ? fee : fee.slice(0, -9),
    fee_payer_pk: feePayer,
    nonce,
    valid_until: ['Since_genesis', validUntil],
    memo,
  };
}
function commonToOcamlV1({
  fee,
  feePayer,
  nonce,
  validUntil,
  memo,
}: CommonJson) {
  memo = Memo.toBase58(Memo.fromString(memo));
  return {
    fee: fee.slice(0, -9),
    fee_payer_pk: feePayer,
    nonce,
    valid_until: validUntil,
    memo,
    fee_token: '1',
  };
}

function signatureToOCaml(signature: SignatureJson) {
  return Signature.toBase58(Signature.fromJSON(signature));
}

function stringToBytesOcaml(string: string) {
  return [...string].map((_, i) => string.charCodeAt(i));
}
```

</file>

<file>

## path: /src/mina-signer/tests/client.test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/tests/client.test.ts

```
import Client from '../dist/node/mina-signer/MinaSigner.js';

describe('Client Class Initialization', () => {
  let client;

  it('should accept &nbsp;mainnet&nbsp; as a valid network parameter', () => {
    client = new Client({ network: 'mainnet' });
    expect(client).toBeDefined();
  });

  it('should accept &nbsp;testnet&nbsp; as a valid network parameter', () => {
    client = new Client({ network: 'testnet' });
    expect(client).toBeDefined();
  });

  it('should throw an error if a value that is not &nbsp;mainnet&nbsp; or &nbsp;testnet&nbsp; is specified', ()
=> {
    try {
      //@ts-ignore
      client = new Client({ network: 'new-network' });
    } catch (error) {
      expect(error).toBeDefined();
    }
  });
});
```

</file>

<file>

## path: /src/mina-signer/tests/keypair.test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/tests/keypair.test.ts

```
import Client from '../dist/node/mina-signer/MinaSigner.js';

describe('Keypair', () => {
  let client: Client;

  beforeAll(async () => {
    client = new Client({ network: 'mainnet' });
  });

  it('generates a valid key pair', () => {
    const keypair = client.genKeys();
    expect(keypair.publicKey).toBeDefined();
    expect(keypair.privateKey).toBeDefined();
  });

  it('can verify a valid key pair', () => {
    const keypair = client.genKeys();
    expect(client.verifyKeypair(keypair)).toBeTruthy();
  });

  it('fails to derive an invalid key pair', () => {
    try {
      client.verifyKeypair({ publicKey: 'invalid', privateKey: 'invalid' });
    } catch (error) {
      expect(error).toBeDefined();
    }
  });

  it('derives an equivalent public key from a private key', () => {
    const keypair = client.genKeys();
    const publicKey = client.derivePublicKey(keypair.privateKey);
    expect(keypair.publicKey).toEqual(publicKey);
  });

  it('can derive a hex-encoded public key from a public key', () => {
    const keypair = client.genKeys();
    const rawPublicKey = client.publicKeyToRaw(keypair.publicKey);
    expect(rawPublicKey).toBeDefined();
  });
});
```

</file>

<file>

## path: /src/mina-signer/tests/message.test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/tests/message.test.ts

```
import Client from '../dist/node/mina-signer/MinaSigner.js';
import type { PrivateKey } from '../dist/node/mina-signer/src/TSTypes.js';
```

```
describe('Message', () => {
  describe('Mainnet network', () => {
    let client: Client;
    let privateKey: PrivateKey;

    beforeAll(async () => {
      client = new Client({ network: 'mainnet' });
      ({ privateKey } = client.genKeys());
    });

    it('generates a signed message', () => {
      const message = client.signMessage('hello', privateKey);
      expect(message.data).toBeDefined();
      expect(message.signature).toBeDefined();
    });

    it('generates a signed message by using signTransaction', () => {
      const message = client.signTransaction('hello', privateKey);
      expect(message.data).toBeDefined();
      expect(message.signature).toBeDefined();
    });

    it('verifies a signed message', () => {
      const message = client.signMessage('hello', privateKey);
      const verifiedMessage = client.verifyMessage(message);
      expect(verifiedMessage).toBeTruthy();
      expect(client.verifyTransaction(message)).toEqual(true);
    });

    it('verifies a signed message generated by signTransaction', () => {
      const message = client.signTransaction('hello', privateKey);
      const verifiedMessage = client.verifyMessage(message);
      expect(verifiedMessage).toBeTruthy();
      expect(client.verifyTransaction(message)).toEqual(true);
    });

    it('does not verify a signed message from &nbsp;testnet&nbsp;', () => {
      const message = client.signMessage('hello', privateKey);
      const testnetClient = new Client({ network: 'testnet' });
      const invalidMessage = testnetClient.verifyMessage(message);
      expect(invalidMessage).toBeFalsy();
      expect(testnetClient.verifyTransaction(message)).toEqual(false);
    });
  });

  describe('Testnet network', () => {
    let client: Client;
    let privateKey: PrivateKey;

    beforeAll(async () => {
      client = new Client({ network: 'testnet' });
      ({ privateKey } = client.genKeys());
    });

    it('generates a signed message', () => {
      const message = client.signMessage('hello', privateKey);
      expect(message.data).toBeDefined();
      expect(message.signature).toBeDefined();
    });

    it('generates a signed message by using signTransaction', () => {
      const message = client.signTransaction('hello', privateKey);
      expect(message.data).toBeDefined();
      expect(message.signature).toBeDefined();
    });

    it('verifies a signed message', () => {
      const message = client.signMessage('hello', privateKey);
      const verifiedMessage = client.verifyMessage(message);
      expect(verifiedMessage).toBeTruthy();
      expect(client.verifyTransaction(message)).toEqual(true);
    });

    it('verifies a signed message generated by signTransaction', () => {
      const message = client.signTransaction('hello', privateKey);
      const verifiedMessage = client.verifyMessage(message);
      expect(verifiedMessage).toBeTruthy();
      expect(client.verifyTransaction(message)).toEqual(true);
    });

    it('does not verify a signed message from &nbsp;mainnet&nbsp;', () => {
      const message = client.signMessage('hello', privateKey);
      const mainnetClient = new Client({ network: 'mainnet' });
      const invalidMessage = mainnetClient.verifyMessage(message);
      expect(invalidMessage).toBeFalsy();
      expect(mainnetClient.verifyTransaction(message)).toEqual(false);
    });
  });
 });
```

</file>

<file>

## path: /src/mina-signer/tests/payment.test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/tests/payment.test.ts

```
import Client from '../dist/node/mina-signer/MinaSigner.js';
import type { Keypair } from '../dist/node/mina-signer/src/TSTypes.js';

describe('Payment', () => {
```

```
describe('Mainnet network', () => {
  let client: Client;
  let keypair: Keypair;

  beforeAll(async () => {
    client = new Client({ network: 'mainnet' });
    keypair = client.genKeys();
  });

  it('generates a signed payment', () => {
    const payment = client.signPayment(
      {
        to: keypair.publicKey,
        from: keypair.publicKey,
        amount: '1',
        fee: '1',
        nonce: '0',
      },
      keypair.privateKey
    );
    expect(payment.data).toBeDefined();
    expect(payment.signature).toBeDefined();
  });

  it('generates a signed transaction by using signTransaction', () => {
    const payment = client.signTransaction(
      {
        to: keypair.publicKey,
        from: keypair.publicKey,
        amount: '1',
        fee: '1',
        nonce: '0',
      },
      keypair.privateKey
    );
    expect(payment.data).toBeDefined();
    expect(payment.signature).toBeDefined();
  });

  it('verifies a signed payment', () => {
    const payment = client.signPayment(
      {
        to: keypair.publicKey,
        from: keypair.publicKey,
        amount: '1',
        fee: '1',
        nonce: '0',
      },
      keypair.privateKey
    );
    const verifiedPayment = client.verifyPayment(payment);
    expect(verifiedPayment).toBeTruthy();
  expect(client.verifyTransaction(payment)).toEqual(true);
});

it('verifies a signed payment generated by signTransaction', () => {
  const payment = client.signTransaction(
    {
      to: keypair.publicKey,
      from: keypair.publicKey,
      amount: '1',
      fee: '1',
      nonce: '0',
    },
    keypair.privateKey
  );
  const verifiedPayment = client.verifyPayment(payment);
  expect(verifiedPayment).toBeTruthy();
  expect(client.verifyTransaction(payment)).toEqual(true);
});

it('hashes a signed payment', () => {
  const payment = client.signPayment(
    {
      to: keypair.publicKey,
      from: keypair.publicKey,
      amount: '1',
      fee: '1',
      nonce: '0',
    },
    keypair.privateKey
  );
  const hashedPayment = client.hashPayment(payment);
  expect(hashedPayment).toBeDefined();
});

it('hashes a signed payment generated by signTransaction', () => {
  const payment = client.signTransaction(
    {
      to: keypair.publicKey,
      from: keypair.publicKey,
      amount: '1',
      fee: '1',
      nonce: '0',
    },
    keypair.privateKey
  );
  const hashedPayment = client.hashPayment(payment);
  expect(hashedPayment).toBeDefined();
});

it('does not verify a signed payment from &nbsp;testnet&nbsp;', () => {
  const payment = client.signPayment(
    {
        to: keypair.publicKey,
        from: keypair.publicKey,
        amount: '1',
        fee: '1',
        nonce: '0',
      },
      keypair.privateKey
    );
    const testnetClient = new Client({ network: 'testnet' });
    const invalidPayment = testnetClient.verifyPayment(payment);
    expect(invalidPayment).toBeFalsy();
    expect(testnetClient.verifyTransaction(payment)).toEqual(false);
  });
});

describe('Testnet network', () => {
  let client: Client;
  let keypair: Keypair;

  beforeAll(async () => {
    client = new Client({ network: 'testnet' });
    keypair = client.genKeys();
  });

  it('generates a signed payment', () => {
    const payment = client.signPayment(
      {
        to: keypair.publicKey,
        from: keypair.publicKey,
        amount: '1',
        fee: '1',
        nonce: '0',
      },
      keypair.privateKey
    );
    expect(payment.data).toBeDefined();
    expect(payment.signature).toBeDefined();
  });

  it('generates a signed transaction by using signTransaction', () => {
    const payment = client.signTransaction(
      {
        to: keypair.publicKey,
        from: keypair.publicKey,
        amount: '1',
        fee: '1',
        nonce: '0',
      },
      keypair.privateKey
    );
    expect(payment.data).toBeDefined();
    expect(payment.signature).toBeDefined();
```

```
});

it('verifies a signed payment', () => {
  const payment = client.signPayment(
    {
      to: keypair.publicKey,
      from: keypair.publicKey,
      amount: '1',
      fee: '1',
      nonce: '0',
    },
    keypair.privateKey
  );
  const verifiedPayment = client.verifyPayment(payment);
  expect(verifiedPayment).toBeTruthy();
  expect(client.verifyTransaction(payment)).toEqual(true);
});

it('verifies a signed payment generated by signTransaction', () => {
  const payment = client.signTransaction(
    {
      to: keypair.publicKey,
      from: keypair.publicKey,
      amount: '1',
      fee: '1',
      nonce: '0',
    },
    keypair.privateKey
  );
  const verifiedPayment = client.verifyPayment(payment);
  expect(verifiedPayment).toBeTruthy();
  expect(client.verifyTransaction(payment)).toEqual(true);
});

it('hashes a signed payment', () => {
  const payment = client.signPayment(
    {
      to: keypair.publicKey,
      from: keypair.publicKey,
      amount: '1',
      fee: '1',
      nonce: '0',
    },
    keypair.privateKey
  );
  const hashedPayment = client.hashPayment(payment);
  expect(hashedPayment).toBeDefined();
});

it('hashes a signed payment generated by signTransaction', () => {
  const payment = client.signTransaction(
    {
          to: keypair.publicKey,
          from: keypair.publicKey,
          amount: '1',
          fee: '1',
          nonce: '0',
        },
        keypair.privateKey
      );
      const hashedPayment = client.hashPayment(payment);
      expect(hashedPayment).toBeDefined();
    });

    it('does not verify a signed payment from &nbsp;mainnet&nbsp;', () => {
      const payment = client.signPayment(
        {
          to: keypair.publicKey,
          from: keypair.publicKey,
          amount: '1',
          fee: '1',
          nonce: '0',
        },
        keypair.privateKey
      );
      const mainnetClient = new Client({ network: 'mainnet' });
      const invalidPayment = mainnetClient.verifyPayment(payment);
      expect(invalidPayment).toBeFalsy();
      expect(mainnetClient.verifyTransaction(payment)).toEqual(false);
    });
  });
});
```

</file>

<file>

## path: /src/mina-signer/tests/rosetta.test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/tests/rosetta.test.ts

```
import Client from '../dist/node/mina-signer/MinaSigner.js';

describe('Rosetta', () => {
  let client: Client;

  const signedRosettaTnxMock = &nbsp;
  {
    "signature":
"389ac7d4077f3d485c1494782870979faa222cd906b25b2687333a92f41e40b925adb08705eddf2a7098e5ac
9938498e8a0ce7c70b25ea392f4846b854086d43",
    "payment": {
      "to": "B62qnzbXmRNo9q32n4SNu2mpB8e7FYYLH8NmaX6oFCBYjjQ8SbD7uzV",
      "from": "B62qnzbXmRNo9q32n4SNu2mpB8e7FYYLH8NmaX6oFCBYjjQ8SbD7uzV",
      "fee": "10000000",
      "token": "1",
      "nonce": "0",
      "memo": null,
      "amount": "1000000000",
      "valid_until": "4294967295"
    },
    "stake_delegation": null
  }&nbsp;;

  beforeAll(async () => {
    client = new Client({ network: 'mainnet' });
  });

  it('generates a valid rosetta transaction', () => {
    const signedGraphQLCommand =
      client.signedRosettaTransactionToSignedCommand(signedRosettaTnxMock);
    const signedRosettaTnxMockJson = JSON.parse(signedRosettaTnxMock);
    const signedGraphQLCommandJson = JSON.parse(signedGraphQLCommand);

    expect(signedRosettaTnxMockJson.payment.to).toEqual(
      signedGraphQLCommandJson.data.payload.body[1].receiver_pk
    );

    expect(signedRosettaTnxMockJson.payment.from).toEqual(
      signedGraphQLCommandJson.data.payload.common.fee_payer_pk
    );

    expect(signedRosettaTnxMockJson.payment.amount).toEqual(
      signedGraphQLCommandJson.data.payload.body[1].amount
    );
  });
});
```

</file>

<file>

path: /src/mina-signer/tests/stake-delegation.test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/tests/stake-delegation.test.ts

```
import Client from '../dist/node/mina-signer/MinaSigner.js';
import type { Keypair } from '../dist/node/mina-signer/src/TSTypes.js';

describe('Stake Delegation', () => {
  describe('Mainnet network', () => {
    let client: Client;
    let keypair: Keypair;

    beforeAll(async () => {
      client = new Client({ network: 'mainnet' });
      keypair = client.genKeys();
    });

    it('generates a signed staked delegation', () => {
      const delegation = client.signStakeDelegation(
        {
          to: keypair.publicKey,
          from: keypair.publicKey,
          fee: '1',
          nonce: '0',
        },
        keypair.privateKey
      );
      expect(delegation.data).toBeDefined();
      expect(delegation.signature).toBeDefined();
    });

    it('generates a signed staked delegation using signTransaction', () => {
      const delegation = client.signTransaction(
        {
          to: keypair.publicKey,
          from: keypair.publicKey,
          fee: '1',
          nonce: '0',
        },
        keypair.privateKey
      );
      expect(delegation.data).toBeDefined();
      expect(delegation.signature).toBeDefined();
    });

    it('verifies a signed delegation', () => {
      const delegation = client.signStakeDelegation(
        {
          to: keypair.publicKey,
          from: keypair.publicKey,
          fee: '1',
          nonce: '0',
        },
        keypair.privateKey
      );
      const verifiedDelegation = client.verifyStakeDelegation(delegation);
      expect(verifiedDelegation).toBeTruthy();
      expect(client.verifyTransaction(delegation)).toEqual(true);
    });

    it('verifies a signed delegation generated by signTransaction', () => {
      const delegation = client.signTransaction(
        {
          to: keypair.publicKey,
          from: keypair.publicKey,
          fee: '1',
          nonce: '0',
        },
        keypair.privateKey
      );
      const verifiedDelegation = client.verifyStakeDelegation(delegation);
      expect(verifiedDelegation).toBeTruthy();
      expect(client.verifyTransaction(delegation)).toEqual(true);
    });

    it('hashes a signed stake delegation', () => {
      const delegation = client.signStakeDelegation(
        {
          to: keypair.publicKey,
          from: keypair.publicKey,
          fee: '1',
          nonce: '0',
        },
        keypair.privateKey
      );
      const hashedDelegation = client.hashStakeDelegation(delegation);
      expect(hashedDelegation).toBeDefined();
    });

    it('hashes a signed stake delegation generated by signTransaction', () => {
      const delegation = client.signTransaction(
        {
          to: keypair.publicKey,
          from: keypair.publicKey,
          fee: '1',
          nonce: '0',
        },
        keypair.privateKey
      );
      const hashedDelegation = client.hashStakeDelegation(delegation);
      expect(hashedDelegation).toBeDefined();
    });
```

```
  it('does not verify a signed message from &nbsp;testnet&nbsp;', () => {
    const delegation = client.signStakeDelegation(
      {
        to: keypair.publicKey,
        from: keypair.publicKey,
        fee: '1',
        nonce: '0',
      },
      keypair.privateKey
    );
    const testnetClient = new Client({ network: 'testnet' });
    const invalidMessage = testnetClient.verifyStakeDelegation(delegation);
    expect(invalidMessage).toBeFalsy();
    expect(testnetClient.verifyTransaction(delegation)).toEqual(false);
  });
});

describe('Testnet network', () => {
  let client: Client;
  let keypair: Keypair;

  beforeAll(async () => {
    client = new Client({ network: 'testnet' });
    keypair = client.genKeys();
  });

  it('generates a signed staked delegation', () => {
    const delegation = client.signStakeDelegation(
      {
        to: keypair.publicKey,
        from: keypair.publicKey,
        fee: '1',
        nonce: '0',
      },
      keypair.privateKey
    );
    expect(delegation.data).toBeDefined();
    expect(delegation.signature).toBeDefined();
  });

  it('generates a signed staked delegation using signTransaction', () => {
    const delegation = client.signTransaction(
      {
        to: keypair.publicKey,
        from: keypair.publicKey,
        fee: '1',
        nonce: '0',
      },
      keypair.privateKey
    );
    expect(delegation.data).toBeDefined();
    expect(delegation.signature).toBeDefined();
```

```
});

it('verifies a signed delegation', () => {
  const delegation = client.signStakeDelegation(
    {
      to: keypair.publicKey,
      from: keypair.publicKey,
      fee: '1',
      nonce: '0',
    },
    keypair.privateKey
  );
  const verifiedDelegation = client.verifyStakeDelegation(delegation);
  expect(verifiedDelegation).toBeTruthy();
  expect(client.verifyTransaction(delegation)).toEqual(true);
});

it('verifies a signed delegation generated by signTransaction', () => {
  const delegation = client.signTransaction(
    {
      to: keypair.publicKey,
      from: keypair.publicKey,
      fee: '1',
      nonce: '0',
    },
    keypair.privateKey
  );
  const verifiedDelegation = client.verifyStakeDelegation(delegation);
  expect(verifiedDelegation).toBeTruthy();
  expect(client.verifyTransaction(delegation)).toEqual(true);
});

it('hashes a signed stake delegation', () => {
  const delegation = client.signStakeDelegation(
    {
      to: keypair.publicKey,
      from: keypair.publicKey,
      fee: '1',
      nonce: '0',
    },
    keypair.privateKey
  );
  const hashedDelegation = client.hashStakeDelegation(delegation);
  expect(hashedDelegation).toBeDefined();
});

it('does not verify a signed message from &nbsp;mainnet&nbsp;', () => {
  const delegation = client.signStakeDelegation(
    {
      to: keypair.publicKey,
      from: keypair.publicKey,
      fee: '1',
      nonce: '0',
    },
    keypair.privateKey
  );
  const mainnetClient = new Client({ network: 'mainnet' });
  const invalidMessage = mainnetClient.verifyStakeDelegation(delegation);
  expect(invalidMessage).toBeFalsy();
  expect(mainnetClient.verifyTransaction(delegation)).toEqual(false);
  });
 });
});
```

</file>

<file>

## path: /src/mina-signer/tests/verify-in-snark.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/tests/verify-in-snark.unit-test.ts

```
import { Field } from '../../lib/core.js';
import { ZkProgram } from '../../lib/proof_system.js';
import Client from '../MinaSigner.js';
import { PrivateKey, Signature } from '../../lib/signature.js';
import { expect } from 'expect';
import { Provable } from '../../lib/provable.js';

let fields = [10n, 20n, 30n, 340817401n, 2091283n, 1n, 0n];
let privateKey = 'EKENaWFuAiqktsnWmxq8zaoR8bSgVdscsghJE5tV6hPoNm8qBKWM';

// sign with mina-signer
let client = new Client({ network: 'mainnet' });
let signed = client.signFields(fields, privateKey);

// verify with mina-signer
let ok = client.verifyFields(signed);
expect(ok).toEqual(true);

// sign with o1js and check that we get the same signature
let fieldsSnarky = fields.map(Field);
let privateKeySnarky = PrivateKey.fromBase58(privateKey);
let signatureSnarky = Signature.create(privateKeySnarky, fieldsSnarky);
expect(signatureSnarky.toBase58()).toEqual(signed.signature);

// verify out-of-snark with o1js
let publicKey = privateKeySnarky.toPublicKey();
let signature = Signature.fromBase58(signed.signature);
Provable.assertEqual(Signature, signature, signatureSnarky);
signature.verify(publicKey, fieldsSnarky).assertTrue();

// verify in-snark with o1js
```

```
const Message = Provable.Array(Field, fields.length);

const MyProgram = ZkProgram({
  name: 'verify-signature',
  methods: {
    verifySignature: {
      privateInputs: [Signature, Message],
      method(signature: Signature, message: Field[]) {
        signature.verify(publicKey, message).assertTrue();
      },
    },
  },
});

await MyProgram.compile();
let proof = await MyProgram.verifySignature(signature, fieldsSnarky);
ok = await MyProgram.verify(proof);
expect(ok).toEqual(true);

// negative test - sign with the wrong private key

let { privateKey: wrongKey } = client.genKeys();
let invalidSigned = client.signFields(fields, wrongKey);
let invalidSignature = Signature.fromBase58(invalidSigned.signature);

// can't verify out of snark
invalidSignature.verify(publicKey, fieldsSnarky).assertFalse();

// can't verify in snark
await expect(() =>
  MyProgram.verifySignature(invalidSignature, fieldsSnarky)
).rejects.toThrow('Constraint unsatisfied');

// negative test - try to verify a different message

let wrongFields = [...fieldsSnarky];
wrongFields[0] = wrongFields[0].add(1);

// can't verify out of snark
signature.verify(publicKey, wrongFields).assertFalse();

// can't verify in snark
await expect(() =>
  MyProgram.verifySignature(signature, wrongFields)
).rejects.toThrow('Constraint unsatisfied');
```

</file>

<file>

## path: /src/mina-signer/tests/zkapp.unit-test.ts

url: https://github.com/o1-labs/o1js/blob/main/src/mina-signer/tests/zkapp.unit-test.ts

```
import { ZkappCommand } from '../../bindings/mina-transaction/gen/transaction-bigint.js';
import * as TransactionJson from '../../bindings/mina-transaction/gen/transaction-json.js';
import Client from '../MinaSigner.js';
import { accountUpdateExample } from '../src/test-vectors/accountUpdate.js';
import { expect } from 'expect';
import { Transaction } from '../../lib/mina.js';
import { PrivateKey } from '../../lib/signature.js';
import { Signature } from '../src/signature.js';
import { mocks } from '../../bindings/crypto/constants.js';

const client = new Client({ network: 'testnet' });
let { publicKey, privateKey } = client.genKeys();

let dummy = ZkappCommand.toJSON(ZkappCommand.emptyValue());
let dummySignature = Signature.toBase58(Signature.dummy());

// we construct a transaction which needs signing of the fee payer and another account update
let accountUpdateExample2: TransactionJson.AccountUpdate = {
  ...accountUpdateExample,
  body: {
    ...accountUpdateExample.body,
    publicKey,
    authorizationKind: {
      isSigned: true,
      isProved: false,
      verificationKeyHash: mocks.dummyVerificationKeyHash,
    },
  },
  authorization: { proof: null, signature: dummySignature },
};

let exampleZkappCommand: TransactionJson.ZkappCommand = {
  ...dummy,
  accountUpdates: [accountUpdateExample, accountUpdateExample2],
  memo: 'E4YM2vTHhWEg66xpj52JErHUBU4pZ1yageL4TVDDpTTSsv8mK6YaH',
};

let exampleFeePayer = {
  feePayer: publicKey,
  fee: '100000000',
  nonce: '1',
  memo: 'test memo',
};

// generates and verifies a signed zkapp command
let zkappCommand = client.signZkappCommand(
  { zkappCommand: exampleZkappCommand, feePayer: exampleFeePayer },
  privateKey
);
expect(zkappCommand.data).toBeDefined();
expect(zkappCommand.signature).toBeDefined();
expect(client.verifyZkappCommand(zkappCommand)).toEqual(true);
expect(client.verifyTransaction(zkappCommand)).toEqual(true);

// generates and verifies a signed zkapp command by using signTransaction
zkappCommand = client.signTransaction(
  { zkappCommand: exampleZkappCommand, feePayer: exampleFeePayer },
  privateKey
);
expect(zkappCommand.data).toBeDefined();
expect(zkappCommand.signature).toBeDefined();
expect(client.verifyZkappCommand(zkappCommand)).toEqual(true);
expect(client.verifyTransaction(zkappCommand)).toEqual(true);

// does not verify a signed zkapp command from &nbsp;mainnet&nbsp;
const mainnetClient = new Client({ network: 'mainnet' });
expect(mainnetClient.verifyZkappCommand(zkappCommand)).toEqual(false);
expect(mainnetClient.verifyTransaction(zkappCommand)).toEqual(false);

// should throw an error if no fee is passed to the fee payer
expect(() => {
  client.signZkappCommand(
    {
      zkappCommand: exampleZkappCommand,
      // @ts-ignore - fee is not defined
      feePayer: { feePayer: publicKey, nonce: '0', memo: 'test memo' },
    },
    privateKey
  );
}).toThrow('Missing fee in fee payer');

// should calculate a correct minimum fee
expect(
  client.getAccountUpdateMinimumFee(exampleZkappCommand.accountUpdates)
).toBe(0.002);

// same transaction signed with o1js (OCaml implementation) gives the same result

let transactionJson = {
  ...exampleZkappCommand,
  feePayer: {
    body: {
      publicKey: exampleFeePayer.feePayer,
      fee: exampleFeePayer.fee,
      nonce: exampleFeePayer.nonce,
      validUntil: null,
    },
    authorization: dummySignature,
  },
  memo: zkappCommand.data.zkappCommand.memo,
};
```

```
let tx = Transaction.fromJSON(transactionJson);
tx.transaction.feePayer.lazyAuthorization = { kind: 'lazy-signature' };
tx.transaction.accountUpdates[1].lazyAuthorization = { kind: 'lazy-signature' };
tx.sign([PrivateKey.fromBase58(privateKey)]);

expect(zkappCommand.data.zkappCommand.feePayer.authorization).toEqual(
  tx.transaction.feePayer.authorization
);
expect(
  zkappCommand.data.zkappCommand.accountUpdates[1].authorization.signature
).toEqual(tx.transaction.accountUpdates[1].authorization.signature);
expect(JSON.stringify(zkappCommand.data.zkappCommand)).toEqual(tx.toJSON());
```

</file>

<file>

## path: /src/provable/curve-bigint.ts

url: https://github.com/o1-labs/o1js/blob/main/src/provable/curve-bigint.ts

```
import { Fq, mod } from '../bindings/crypto/finite_field.js';
import { GroupProjective, Pallas } from '../bindings/crypto/elliptic_curve.js';
import { versionBytes } from '../bindings/crypto/constants.js';
import {
  record,
  withCheck,
  withVersionNumber,
} from '../bindings/lib/binable.js';
import { base58, withBase58 } from '../lib/base58.js';
import { Bool, checkRange, Field, pseudoClass } from './field-bigint.js';
import {
  BinableBigint,
  ProvableBigint,
  provable,
} from '../bindings/lib/provable-bigint.js';
import { HashInputLegacy } from './poseidon-bigint.js';

export { Group, PublicKey, Scalar, PrivateKey, versionNumbers };

// TODO generate
const versionNumbers = {
  field: 1,
  scalar: 1,
  publicKey: 1,
  signature: 1,
};

type Group = { x: Field; y: Field };
type PublicKey = { x: Field; isOdd: Bool };
type Scalar = bigint;
type PrivateKey = bigint;

/**
 * A non-zero point on the Pallas curve in affine form { x, y }
 */
const Group = {
  toProjective({ x, y }: Group): GroupProjective {
    return Pallas.fromAffine({ x, y, infinity: false });
  },
  /**
   * Convert a projective point to a non-zero affine point.
   * Throws an error if the point is zero / infinity, i.e. if z === 0
   */
  fromProjective(point: GroupProjective): Group {
    let { x, y, infinity } = Pallas.toAffine(point);
    if (infinity) throw Error('Group.fromProjective: point is infinity');
    return { x, y };
  },
  get generatorMina(): Group {
    return Group.fromProjective(Pallas.one);
  },
  scale(point: Group, scalar: Scalar): Group {
    return Group.fromProjective(
      Pallas.scale(Group.toProjective(point), scalar)
    );
  },
  b: Pallas.b,
  toFields({ x, y }: Group) {
    return [x, y];
  },
};

let FieldWithVersion = withVersionNumber(Field, versionNumbers.field);
let BinablePublicKey = withVersionNumber(
  withCheck(
    record({ x: FieldWithVersion, isOdd: Bool }, ['x', 'isOdd']),
    ({ x }) => {
      let { mul, add } = Field;
      let ySquared = add(mul(x, mul(x, x)), Pallas.b);
      if (!Field.isSquare(ySquared)) {
        throw Error('PublicKey: not a valid group element');
      }
    }
  ),
  versionNumbers.publicKey
);

/**
 * A public key, represented by a non-zero point on the Pallas curve, in compressed form { x, isOdd }
 */
const PublicKey = {
  ...provable({ x: Field, isOdd: Bool }),
  ...withBase58(BinablePublicKey, versionBytes.publicKey),

  toJSON(publicKey: PublicKey) {
    return PublicKey.toBase58(publicKey);
  },
  fromJSON(json: string): PublicKey {
    return PublicKey.fromBase58(json);
  },

  toGroup({ x, isOdd }: PublicKey): Group {
    let { mul, add } = Field;
    let ySquared = add(mul(x, mul(x, x)), Pallas.b);
    let y = Field.sqrt(ySquared);
    if (y === undefined) {
      throw Error('PublicKey.toGroup: not a valid group element');
    }
    if (isOdd !== (y & 1n)) y = Field.negate(y);
    return { x, y };
  },
  fromGroup({ x, y }: Group): PublicKey {
    let isOdd = (y & 1n) as Bool;
    return { x, isOdd };
  },

  equal(pk1: PublicKey, pk2: PublicKey) {
    return pk1.x === pk2.x && pk1.isOdd === pk2.isOdd;
  },

  toInputLegacy({ x, isOdd }: PublicKey): HashInputLegacy {
    return { fields: [x], bits: [!!isOdd] };
  },
};

const checkScalar = checkRange(0n, Fq.modulus, 'Scalar');

/**
 * The scalar field of the Pallas curve
 */
const Scalar = pseudoClass(
  function Scalar(value: bigint | number | string): Scalar {
    return mod(BigInt(value), Fq.modulus);
  },
  {
    ...ProvableBigint(checkScalar),
    ...BinableBigint(Fq.sizeInBits, checkScalar),
    ...Fq,
  }
);

let BinablePrivateKey = withVersionNumber(Scalar, versionNumbers.scalar);
let Base58PrivateKey = base58(BinablePrivateKey, versionBytes.privateKey);
```

```
/**
 * A private key, represented by a scalar of the Pallas curve
 */
const PrivateKey = {
  ...Scalar,
  ...provable(Scalar),
  ...Base58PrivateKey,
  ...BinablePrivateKey,
  toPublicKey(key: PrivateKey) {
    return PublicKey.fromGroup(Group.scale(Group.generatorMina, key));
  },
};
```

</file>

<file>

## path: /src/provable/field-bigint.ts

url: https://github.com/o1-labs/o1js/blob/main/src/provable/field-bigint.ts

```
import { randomBytes } from '../bindings/crypto/random.js';
import { Fp, mod } from '../bindings/crypto/finite_field.js';
import {
  BinableBigint,
  HashInput,
  ProvableBigint,
} from '../bindings/lib/provable-bigint.js';

export { Field, Bool, UInt32, UInt64, Sign };
export { pseudoClass, sizeInBits, checkRange, checkField };

type Field = bigint;
type Bool = 0n | 1n;
type UInt32 = bigint;
type UInt64 = bigint;

const sizeInBits = Fp.sizeInBits;

type minusOne =
  0x40000000000000000000000000000000224698fc094cf91b992d30ed00000000n;
const minusOne: minusOne =
  0x40000000000000000000000000000000224698fc094cf91b992d30ed00000000n;
type Sign = 1n | minusOne;

const checkField = checkRange(0n, Fp.modulus, 'Field');
const checkBool = checkAllowList(new Set([0n, 1n]), 'Bool');
const checkSign = checkAllowList(new Set([1n, minusOne]), 'Sign');

/**
 * The base field of the Pallas curve
 */
const Field = pseudoClass(
  function Field(value: bigint | number | string): Field {
    return mod(BigInt(value), Fp.modulus);
  },
  {
    ...ProvableBigint(checkField),
    ...BinableBigint(Fp.sizeInBits, checkField),
    ...Fp,
  }
);

/**
 * A field element which is either 0 or 1
 */
const Bool = pseudoClass(
  function Bool(value: boolean): Bool {
    return BigInt(value) as Bool;
  },
  {
    ...ProvableBigint<Bool>(checkBool),
    ...BinableBigint<Bool>(1, checkBool),
    toInput(x: Bool): HashInput {
      return { fields: [], packed: [[x, 1]] };
    },
    toBoolean(x: Bool) {
      return !!x;
    },
    toJSON(x: Bool) {
      return !!x;
    },
    fromJSON(b: boolean) {
      let x = BigInt(b) as Bool;
      checkBool(x);
      return x;
    },
    sizeInBytes() {
      return 1;
    },
    fromField(x: Field) {
      checkBool(x);
      return x as 0n | 1n;
    },
  }
);

function Unsigned(bits: number) {
  let maxValue = (1n << BigInt(bits)) - 1n;
  let checkUnsigned = checkRange(0n, 1n << BigInt(bits), &nbsp;UInt${bits}&nbsp;);
  let binable = BinableBigint(bits, checkUnsigned);
  let bytes = Math.ceil(bits / 8);
  return pseudoClass(
    function Unsigned(value: bigint | number | string) {
      let x = BigInt(value);
      checkUnsigned(x);
      return x;
    },
    {
      ...ProvableBigint(checkUnsigned),
      ...binable,
      toInput(x: bigint): HashInput {
        return { fields: [], packed: [[x, bits]] };
      },
      maxValue,
      random() {
        return binable.fromBytes([...randomBytes(bytes)]);
      },
    }
  );
}
const UInt32 = Unsigned(32);
const UInt64 = Unsigned(64);

const Sign = pseudoClass(
  function Sign(value: 1 | -1): Sign {
    if (value !== 1 && value !== -1)
      throw Error('Sign: input must be 1 or -1.');
    return mod(BigInt(value), Fp.modulus) as Sign;
  },
  {
    ...ProvableBigint<Sign, 'Positive' | 'Negative'>(checkSign),
    ...BinableBigint<Sign>(1, checkSign),
    emptyValue() {
      return 1n;
    },
    toInput(x: Sign): HashInput {
      return { fields: [], packed: [[x === 1n ? 1n : 0n, 1]] };
    },
    fromFields([x]: Field[]): Sign {
      if (x === 0n) return 1n;
      checkSign(x);
      return x as Sign;
    },
    toJSON(x: Sign) {
      return x === 1n ? 'Positive' : 'Negative';
    },
    fromJSON(x: 'Positive' | 'Negative'): Sign {
      if (x !== 'Positive' && x !== 'Negative')
        throw Error('Sign: invalid input');
      return x === 'Positive' ? 1n : minusOne;
    },
  }
);
```

```
// helper

function pseudoClass<
  F extends (...args: any) => any,
  M
  // M extends Provable<ReturnType<F>>
>(constructor: F, module: M) {
  return Object.assign<F, M>(constructor, module);
}

// validity checks

function checkRange(lower: bigint, upper: bigint, name: string) {
  return (x: bigint) => {
    if (x < lower)
      throw Error(
        &nbsp;${name}: inputs smaller than ${lower} are not allowed, got ${x}&nbsp;
      );
    if (x >= upper)
      throw Error(
        &nbsp;${name}: inputs larger than ${upper - 1n} are not allowed, got ${x}&nbsp;
      );
  };
}

function checkAllowList(valid: Set<bigint>, name: string) {
  return (x: bigint) => {
    if (!valid.has(x)) {
      throw Error(
        &nbsp;${name}: input must be one of ${[...valid].join(', ')}, got ${x}&nbsp;
      );
    }
  };
}
```

</file>

<file>

## path: /src/provable/poseidon-bigint.ts

url: https://github.com/o1-labs/o1js/blob/main/src/provable/poseidon-bigint.ts

```
import { Field, sizeInBits } from './field-bigint.js';
import { Poseidon, PoseidonLegacy } from '../bindings/crypto/poseidon.js';
import { prefixes } from '../bindings/crypto/constants.js';
import { createHashInput } from '../bindings/lib/provable-generic.js';
import { GenericHashInput } from '../bindings/lib/generic.js';
import { createHashHelpers } from '../lib/hash-generic.js';
```

```
export {
  Poseidon,
  Hash,
  HashInput,
  prefixes,
  packToFields,
  hashWithPrefix,
  packToFieldsLegacy,
  HashInputLegacy,
  inputToBitsLegacy,
  HashLegacy,
};

type HashInput = GenericHashInput<Field>;
const HashInput = createHashInput<Field>();
const Hash = createHashHelpers(Field, Poseidon);
let { hashWithPrefix } = Hash;

const HashLegacy = createHashHelpers(Field, PoseidonLegacy);

/**
 * Convert the {fields, packed} hash input representation to a list of field elements
 * Random_oracle_input.Chunked.pack_to_fields
 */
function packToFields({ fields = [], packed = [] }: HashInput) {
  if (packed.length === 0) return fields;
  let packedBits = [];
  let currentPackedField = 0n;
  let currentSize = 0;
  for (let [field, size] of packed) {
    currentSize += size;
    if (currentSize < 255) {
      currentPackedField = currentPackedField * (1n << BigInt(size)) + field;
    } else {
      packedBits.push(currentPackedField);
      currentSize = size;
      currentPackedField = field;
    }
  }
  packedBits.push(currentPackedField);
  return fields.concat(packedBits);
}

/**
 * Random_oracle_input.Legacy.pack_to_fields
 */
function packToFieldsLegacy({ fields, bits }: HashInputLegacy) {
  let packedFields = [];
  while (bits.length > 0) {
    let fieldBits = bits.splice(0, sizeInBits - 1);
    let field = Field.fromBits(fieldBits);
    packedFields.push(field);
  }
  return fields.concat(packedFields);
}
function inputToBitsLegacy({ fields, bits }: HashInputLegacy) {
  let fieldBits = fields.map(Field.toBits).flat();
  return fieldBits.concat(bits);
}

type HashInputLegacy = { fields: Field[]; bits: boolean[] };

const HashInputLegacy = {
  empty(): HashInputLegacy {
    return { fields: [], bits: [] };
  },
  bits(bits: boolean[]): HashInputLegacy {
    return { fields: [], bits };
  },
  append(input1: HashInputLegacy, input2: HashInputLegacy): HashInputLegacy {
    return {
      fields: (input1.fields ?? []).concat(input2.fields ?? []),
      bits: (input1.bits ?? []).concat(input2.bits ?? []),
    };
  },
};
```

</file>

<file>

## path: /src/snarky.d.ts

url: https://github.com/o1-labs/o1js/blob/main/src/snarky.d.ts

```
import type { Account as JsonAccount } from './bindings/mina-transaction/gen/transaction-json.js';
import type { Field, FieldConst, FieldVar } from './lib/field.js';
import type { BoolVar, Bool } from './lib/bool.js';
import type { ScalarConst } from './lib/scalar.js';
import type {
  MlArray,
  MlPair,
  MlList,
  MlOption,
  MlBool,
  MlBytes,
  MlResult,
  MlUnit,
  MlString,
  MlTuple,
} from './lib/ml/base.js';
import type { MlHashInput } from './lib/ml/conversion.js';
import type {
  SnarkKey,
  SnarkKeyHeader,
  MlWrapVerificationKey,
} from './lib/proof-system/prover-keys.js';
import { getWasm } from './bindings/js/wrapper.js';
import type {
  WasmFpSrs,
  WasmFqSrs,
} from './bindings/compiled/node_bindings/plonk_wasm.cjs';

export { ProvablePure, Provable, Ledger, Pickles, Gate, GateType, getWasm };

// internal
export {
  Snarky,
  Test,
  JsonGate,
  MlPublicKey,
  MlPublicKeyVar,
  FeatureFlags,
  MlFeatureFlags,
};

/**
 * &nbsp;Provable<T>&nbsp; is the general circuit type interface in o1js. &nbsp;Provable<T>&nbsp;
interface describes how a type &nbsp;T&nbsp; is made up of {@link Field} elements and "auxiliary" (non-
provable) data.
 *
 * &nbsp;Provable<T>&nbsp; is the required input type in a few places in o1js. One convenient way to create
a &nbsp;Provable<T>&nbsp; is using &nbsp;Struct&nbsp;.
 *
 * The properties and methods on the provable type exist in all base o1js types as well (aka. {@link Field},
{@link Bool}, etc.). In most cases, a zkApp developer does not need these functions to create zkApps.
 */
declare interface Provable<T> {
  /**
   * A function that takes &nbsp;value&nbsp;, an element of type &nbsp;T&nbsp;, as argument and returns
an array of {@link Field} elements that make up the provable data of &nbsp;value&nbsp;.
   *
   * @param value - the element of type &nbsp;T&nbsp; to generate the {@link Field} array from.
   *
   * @return A {@link Field} array describing how this &nbsp;T&nbsp; element is made up of {@link Field}
elements.
   */
  toFields: (value: T) => Field[];

  /**
   * A function that takes &nbsp;value&nbsp; (optional), an element of type &nbsp;T&nbsp;, as argument and
returns an array of any type that make up the "auxiliary" (non-provable) data of &nbsp;value&nbsp;.
   *
   * @param value - the element of type &nbsp;T&nbsp; to generate the auxiliary data array from, optional. If
not provided, a default value for auxiliary data is returned.
```

```
   *
   * @return An array of any type describing how this &nbsp;T&nbsp; element is made up of "auxiliary" (non-
provable) data.
   */
  toAuxiliary: (value?: T) => any[];

  /**
   * A function that returns an element of type &nbsp;T&nbsp; from the given provable and "auxiliary" data.
   *
   * **Important**: For any element of type &nbsp;T&nbsp;, this function is the reverse operation of calling
{@link toFields} and {@link toAuxilary} methods on an element of type &nbsp;T&nbsp;.
   *
   * @param fields - an array of {@link Field} elements describing the provable data of the new
&nbsp;T&nbsp; element.
   * @param aux - an array of any type describing the "auxiliary" data of the new &nbsp;T&nbsp; element,
optional.
   *
   * @return An element of type &nbsp;T&nbsp; generated from the given provable and "auxiliary" data.
   */
  fromFields: (fields: Field[], aux: any[]) => T;

  /**
   * Return the size of the &nbsp;T&nbsp; type in terms of {@link Field} type, as {@link Field} is the primitive
type.
   *
   * **Warning**: This function returns a &nbsp;number&nbsp;, so you cannot use it to prove something on
chain. You can use it during debugging or to understand the memory complexity of some type.
   *
   * @return A &nbsp;number&nbsp; representing the size of the &nbsp;T&nbsp; type in terms of {@link
Field} type.
   */
  sizeInFields(): number;

  /**
   * Add assertions to the proof to check if &nbsp;value&nbsp; is a valid member of type &nbsp;T&nbsp;.
   * This function does not return anything, instead it creates any number of assertions to prove that
&nbsp;value&nbsp; is a valid member of the type &nbsp;T&nbsp;.
   *
   * For instance, calling check function on the type {@link Bool} asserts that the value of the element is
either 1 or 0.
   *
   * @param value - the element of type &nbsp;T&nbsp; to put assertions on.
   */
  check: (value: T) => void;
}

/**
 * &nbsp;ProvablePure<T>&nbsp; is a special kind of {@link Provable} interface, where the "auxiliary" (non-
provable) data is empty. This means the type consists only of field elements, in that sense it is "pure".
 * Any element on the interface &nbsp;ProvablePure<T>&nbsp; is also an element of the interface
&nbsp;Provable<T>&nbsp; where the "auxiliary" data is empty.
 *
 * Examples where &nbsp;ProvablePure<T>&nbsp; is required are types of on-chain state, events and actions.
 *
 * It includes the same properties and methods as the {@link Provable} interface.
 */
declare interface ProvablePure<T> extends Provable<T> {
  /**
   * A function that takes &nbsp;value&nbsp;, an element of type &nbsp;T&nbsp;, as argument and returns an array of {@link Field} elements that make up the provable data of &nbsp;value&nbsp;.
   *
   * @param value - the element of type &nbsp;T&nbsp; to generate the {@link Field} array from.
   *
   * @return A {@link Field} array describing how this &nbsp;T&nbsp; element is made up of {@link Field} elements.
   */
  toFields: (value: T) => Field[];

  /**
   * A function that takes &nbsp;value&nbsp; (optional), an element of type &nbsp;T&nbsp;, as argument and returns an array of any type that make up the "auxiliary" (non-provable) data of &nbsp;value&nbsp;.
   * As any element of the interface &nbsp;ProvablePure<T>&nbsp; includes no "auxiliary" data by definition, this function always returns a default value.
   *
   * @param value - the element of type &nbsp;T&nbsp; to generate the auxiliary data array from, optional. If not provided, a default value for auxiliary data is returned.
   *
   * @return An empty array, as any element of the interface &nbsp;ProvablePure<T>&nbsp; includes no "auxiliary" data by definition.
   */
  toAuxiliary: (value?: T) => any[];

  /**
   * A function that returns an element of type &nbsp;T&nbsp; from the given provable data.
   *
   * **Important**: For any element of type &nbsp;T&nbsp;, this function is the reverse operation of calling {@link toFields} method on an element of type &nbsp;T&nbsp;.
   *
   * @param fields - an array of {@link Field} elements describing the provable data of the new &nbsp;T&nbsp; element.
   *
   * @return An element of type &nbsp;T&nbsp; generated from the given provable data.
   */
  fromFields: (fields: Field[]) => T;

  /**
   * Return the size of the &nbsp;T&nbsp; type in terms of {@link Field} type, as {@link Field} is the primitive type.
   *
   * **Warning**: This function returns a &nbsp;number&nbsp;, so you cannot use it to prove something on chain. You can use it during debugging or to understand the memory complexity of some type.
   *
   * @return A &nbsp;number&nbsp; representing the size of the &nbsp;T&nbsp; type in terms of {@link
```

```
Field} type.
   */
  sizeInFields(): number;

  /**
   * Add assertions to the proof to check if &nbsp;value&nbsp; is a valid member of type &nbsp;T&nbsp;.
   * This function does not return anything, rather creates any number of assertions on the proof to prove
&nbsp;value&nbsp; is a valid member of the type &nbsp;T&nbsp;.
   *
   * For instance, calling check function on the type {@link Bool} asserts that the value of the element is
either 1 or 0.
   *
   * @param value - the element of type &nbsp;T&nbsp; to put assertions on.
   */
  check: (value: T) => void;
}

type MlGroup = MlPair<FieldVar, FieldVar>;

declare namespace Snarky {
  type Main = (publicInput: MlArray<FieldVar>) => void;
  type Keypair = unknown;
  type VerificationKey = unknown;
  type Proof = unknown;
}

/**
 * Internal interface to snarky-ml
 *
 * Note for devs: This module is intended to closely mirror snarky-ml's core, low-level APIs.
 */
declare const Snarky: {
  /**
   * witness &nbsp;sizeInFields&nbsp; field element variables
   *
   * Note: this is called "exists" because in a proof, you use it like this:
   * > "I prove that there exists x, such that (some statement)"
   */
  exists(
    sizeInFields: number,
    compute: () => MlArray<FieldConst>
  ): MlArray<FieldVar>;
  /**
   * witness a single field element variable
   */
  existsVar(compute: () => FieldConst): FieldVar;

  /**
   * APIs that have to do with running provable code
   */
  run: {
    /**
   * Runs code as a prover.
   */
  asProver(f: () => void): void;
  /**
   * Check whether we are inside an asProver or exists block
   */
  inProverBlock(): boolean;
  /**
   * Runs code and checks its correctness.
   */
  runAndCheck(f: () => void): void;
  /**
   * Runs code in prover mode, without checking correctness.
   */
  runUnchecked(f: () => void): void;
  /**
   * Returns information about the constraint system in the callback function.
   */
  constraintSystem(f: () => void): {
    rows: number;
    digest: string;
    json: JsonConstraintSystem;
  };
};

/**
 * APIs to add constraints on field variables
 */
field: {
  /**
   * add x, y to get a new AST node Add(x, y); handles if x, y are constants
   */
  add(x: FieldVar, y: FieldVar): FieldVar;
  /**
   * scale x by a constant to get a new AST node Scale(c, x); handles if x is a constant
   */
  scale(c: FieldConst, x: FieldVar): FieldVar;
  /**
   * witnesses z = x*y and constrains it with [assert_r1cs]; handles constants
   */
  mul(x: FieldVar, y: FieldVar): FieldVar;
  /**
   * evaluates a CVar by walking the AST and reading Vars from a list of public input + aux values
   */
  readVar(x: FieldVar): FieldConst;
  /**
   * x === y without handling of constants
   */
  assertEqual(x: FieldVar, y: FieldVar): void;
  /**
   * x*y === z without handling of constants
   */
  assertMul(x: FieldVar, y: FieldVar, z: FieldVar): void;
  /**
   * x*x === y without handling of constants
   */
  assertSquare(x: FieldVar, y: FieldVar): void;
  /**
   * x*x === x without handling of constants
   */
  assertBoolean(x: FieldVar): void;
  /**
   * check x < y and x <= y
   */
  compare(
    bitLength: number,
    x: FieldVar,
    y: FieldVar
  ): [_: 0, less: BoolVar, lessOrEqual: BoolVar];
  /**
   *
   */
  toBits(length: number, x: FieldVar): MlArray<BoolVar>;
  /**
   *
   */
  fromBits(bits: MlArray<BoolVar>): FieldVar;
  /**
   * returns x truncated to the lowest &nbsp;16 * lengthDiv16&nbsp; bits
   * => can be used to assert that x fits in &nbsp;16 * lengthDiv16&nbsp; bits.
   *
   * more efficient than &nbsp;toBits()&nbsp; because it uses the EC_endoscalar gate;
   * does 16 bits per row (vs 1 bits per row that you can do with generic gates).
   */
  truncateToBits16(lengthDiv16: number, x: FieldVar): FieldVar;
  /**
   * returns a new witness from an AST
   * (implemented with toConstantAndTerms)
   */
  seal(x: FieldVar): FieldVar;
  /**
   * Unfolds AST to get &nbsp;x = c + c0*Var(i0) + ... + cn*Var(in)&nbsp;,
   * returns &nbsp;(c, [(c0, i0), ..., (cn, in)])&nbsp;;
   * c is optional
   */
  toConstantAndTerms(
    x: FieldVar
  ): [
    _: 0,
    constant: MlOption<FieldConst>,
    terms: MlList<MlPair<FieldConst, number>>
  ];
};
```

```
gates: {
  zero(in1: FieldVar, in2: FieldVar, out: FieldVar): void;

  generic(
    sl: FieldConst,
    l: FieldVar,
    sr: FieldConst,
    r: FieldVar,
    so: FieldConst,
    o: FieldVar,
    sm: FieldConst,
    sc: FieldConst
  ): void;

  poseidon(state: MlArray<MlTuple<Field, 3>>): void;

  /**
   * Low-level Elliptic Curve Addition gate.
   */
  ecAdd(
    p1: MlGroup,
    p2: MlGroup,
    p3: MlGroup,
    inf: FieldVar,
    same_x: FieldVar,
    slope: FieldVar,
    inf_z: FieldVar,
    x21_inv: FieldVar
  ): MlGroup;

  ecScale(
    state: MlArray<
      [
        _: 0,
        accs: MlArray<MlTuple<FieldVar, 2>>,
        bits: MlArray<FieldVar>,
        ss: MlArray<FieldVar>,
        base: MlGroup,
        nPrev: Field,
        nNext: Field
      ]
    >
  ): void;

  ecEndoscale(
    state: MlArray<
      [
        _: 0,
        xt: FieldVar,
        yt: FieldVar,
        xp: FieldVar,
        yp: FieldVar,
        nAcc: FieldVar,
        xr: FieldVar,
        yr: FieldVar,
        s1: FieldVar,
        s3: FieldVar,
        b1: FieldVar,
        b2: FieldVar,
        b3: FieldVar,
        b4: FieldVar
      ]
    >,
    xs: FieldVar,
    ys: FieldVar,
    nAcc: FieldVar
  ): void;

  ecEndoscalar(
    state: MlArray<
      [
        _: 0,
        n0: FieldVar,
        n8: FieldVar,
        a0: FieldVar,
        b0: FieldVar,
        a8: FieldVar,
        b8: FieldVar,
        x0: FieldVar,
        x1: FieldVar,
        x2: FieldVar,
        x3: FieldVar,
        x4: FieldVar,
        x5: FieldVar,
        x6: FieldVar,
        x7: FieldVar
      ]
    >
  ): void;

  lookup(input: MlTuple<FieldVar, 7>): void;

  /**
   * Range check gate
   *
   * @param v0 field var to be range checked
   * @param v0p bits 16 to 88 as 6 12-bit limbs
   * @param v0c bits 0 to 16 as 8 2-bit limbs
   * @param compact boolean field elements -- whether to use "compact mode"
   */
  rangeCheck0(
    v0: FieldVar,
    v0p: MlTuple<FieldVar, 6>,
    v0c: MlTuple<FieldVar, 8>,
    compact: FieldConst
  ): void;

  rangeCheck1(
    v2: FieldVar,
    v12: FieldVar,
    vCurr: MlTuple<FieldVar, 13>,
    vNext: MlTuple<FieldVar, 15>
  ): void;

  xor(
    in1: FieldVar,
    in2: FieldVar,
    out: FieldVar,
    in1_0: FieldVar,
    in1_1: FieldVar,
    in1_2: FieldVar,
    in1_3: FieldVar,
    in2_0: FieldVar,
    in2_1: FieldVar,
    in2_2: FieldVar,
    in2_3: FieldVar,
    out_0: FieldVar,
    out_1: FieldVar,
    out_2: FieldVar,
    out_3: FieldVar
  ): void;

  foreignFieldAdd(
    left: MlTuple<FieldVar, 3>,
    right: MlTuple<FieldVar, 3>,
    fieldOverflow: FieldVar,
    carry: FieldVar,
    foreignFieldModulus: MlTuple<FieldConst, 3>,
    sign: FieldConst
  ): void;

  foreignFieldMul(
    left: MlTuple<FieldVar, 3>,
    right: MlTuple<FieldVar, 3>,
    remainder: MlTuple<FieldVar, 2>,
    quotient: MlTuple<FieldVar, 3>,
    quotientHiBound: FieldVar,
    product1: MlTuple<FieldVar, 3>,
    carry0: FieldVar,
    carry1p: MlTuple<FieldVar, 7>,
    carry1c: MlTuple<FieldVar, 4>,
    foreignFieldModulus2: FieldConst,
    negForeignFieldModulus: MlTuple<FieldConst, 3>
  ): void;

  rotate(
      field: FieldVar,
      rotated: FieldVar,
      excess: FieldVar,
      limbs: MlArray<FieldVar>,
      crumbs: MlArray<FieldVar>,
      two_to_rot: FieldConst
    ): void;

    addFixedLookupTable(id: number, data: MlArray<MlArray<FieldConst>>): void;

    addRuntimeTableConfig(id: number, firstColumn: MlArray<FieldConst>): void;

    raw(
      kind: KimchiGateType,
      values: MlArray<FieldVar>,
      coefficients: MlArray<FieldConst>
    ): void;
  };

  bool: {
    not(x: BoolVar): BoolVar;

    and(x: BoolVar, y: BoolVar): BoolVar;

    or(x: BoolVar, y: BoolVar): BoolVar;

    equals(x: BoolVar, y: BoolVar): BoolVar;

    assertEqual(x: BoolVar, y: BoolVar): void;
  };

  group: {
    scale(p: MlGroup, s: MlArray<BoolVar>): MlGroup;
  };

  /**
   * The circuit API is a low level interface to create zero-knowledge proofs
   */
  circuit: {
    /**
     * Generates a proving key and a verification key for the provable function &nbsp;main&nbsp;
     */
    compile(main: Snarky.Main, publicInputSize: number): Snarky.Keypair;

    /**
     * Proves a statement using the private input, public input and the keypair of the circuit.
     */
    prove(
      main: Snarky.Main,
      publicInputSize: number,
      publicInput: MlArray<FieldConst>,
      keypair: Snarky.Keypair
    ): Snarky.Proof;

    /**
     * Verifies a proof using the public input, the proof and the verification key of the circuit.
     */
    verify(
      publicInput: MlArray<FieldConst>,
      proof: Snarky.Proof,
      verificationKey: Snarky.VerificationKey
    ): boolean;

    keypair: {
      getVerificationKey(keypair: Snarky.Keypair): Snarky.VerificationKey;
      /**
       * Returns a low-level JSON representation of the circuit:
       * a list of gates, each of which represents a row in a table, with certain coefficients and wires to other
(row, column) pairs
       */
      getConstraintSystemJSON(keypair: Snarky.Keypair): JsonConstraintSystem;
    };
  };

  poseidon: {
    update(
      state: MlArray<FieldVar>,
      input: MlArray<FieldVar>
    ): [0, FieldVar, FieldVar, FieldVar];

    hashToGroup(input: MlArray<FieldVar>): MlPair<FieldVar, FieldVar>;

    sponge: {
      create(isChecked: boolean): unknown;
      absorb(sponge: unknown, x: FieldVar): void;
      squeeze(sponge: unknown): FieldVar;
    };
  };
};

declare enum KimchiGateType {
  Zero,
  Generic,
  Poseidon,
  CompleteAdd,
  VarBaseMul,
  EndoMul,
  EndoMulScalar,
  Lookup,
  CairoClaim,
  CairoInstruction,
  CairoFlags,
  CairoTransition,
  RangeCheck0,
  RangeCheck1,
  ForeignFieldAdd,
  ForeignFieldMul,
  Xor16,
  Rot64,
}

type GateType =
  | 'Zero'
  | 'Generic'
  | 'Poseidon'
  | 'CompleteAdd'
  | 'VarbaseMul'
  | 'EndoMul'
  | 'EndoMulScalar'
  | 'Lookup'
  | 'RangeCheck0'
  | 'RangeCheck1'
  | 'ForeignFieldAdd'
  | 'ForeignFieldMul'
  | 'Xor16'
  | 'Rot64';

type JsonGate = {
  typ: GateType;
  wires: { row: number; col: number }[];
  coeffs: string[];
};
type JsonConstraintSystem = { gates: JsonGate[]; public_input_size: number };

type Gate = {
  type: GateType;
  wires: { row: number; col: number }[];
  coeffs: string[];
};

// TODO: Add this when OCaml bindings are implemented:
// declare class EndoScalar {
//   static toFields(x: Scalar): Field[];
//   static fromFields(fields: Field[]): Scalar;
//   static sizeInFields(): number;
// }

type MlPublicKey = [_: 0, x: FieldConst, isOdd: MlBool];
type MlPublicKeyVar = [_: 0, x: FieldVar, isOdd: BoolVar];

/**
 * Represents the Mina ledger.
 */
declare class Ledger {
  /**
   * Creates a fresh ledger.
   */
  static create(): Ledger;

  /**
   * Adds an account and its balance to the ledger.
   */
  addAccount(publicKey: MlPublicKey, balance: string): void;

  /**
   * Applies a JSON transaction to the ledger.
   */
  applyJsonTransaction(
    txJson: string,
    accountCreationFee: string,
    networkState: string
  ): void;

  /**
   * Returns an account.
   */
  getAccount(
    publicKey: MlPublicKey,
    tokenId: FieldConst
  ): JsonAccount | undefined;
}

declare const Test: {
  encoding: {
    // arbitrary base58Check encoding
    toBase58(s: MlBytes, versionByte: number): string;
    ofBase58(base58: string, versionByte: number): MlBytes;

    // base58 encoding of some transaction types
    publicKeyToBase58(publicKey: MlPublicKey): string;
    publicKeyOfBase58(publicKeyBase58: string): MlPublicKey;
    privateKeyToBase58(privateKey: ScalarConst): string;
    privateKeyOfBase58(privateKeyBase58: string): ScalarConst;
    tokenIdToBase58(field: FieldConst): string;
    tokenIdOfBase58(fieldBase58: string): FieldConst;
    memoToBase58(memoString: string): string;
    memoHashBase58(memoBase58: string): FieldConst;
  };

  tokenId: {
    // derive custom token ids
    derive(publicKey: MlPublicKey, tokenId: FieldConst): FieldConst;
    deriveChecked(publicKey: MlPublicKeyVar, tokenId: FieldVar): FieldVar;
  };

  poseidon: {
    hashToGroup(input: MlArray<FieldConst>): MlPair<FieldConst, FieldConst>;
  };
```

```
signature: {
  /**
   * Signs a {@link Field} element.
   */
  signFieldElement(
    messageHash: FieldConst,
    privateKey: ScalarConst,
    isMainnet: boolean
  ): string;
  /**
   * Returns a dummy signature.
   */
  dummySignature(): string;
};

fieldsFromJson: {
  accountUpdate(json: string): MlArray<FieldConst>;
};
hashFromJson: {
  accountUpdate(json: string): FieldConst;
  /**
   * Returns the commitment of a JSON transaction.
   */
  transactionCommitments(txJson: string): {
    commitment: FieldConst;
    fullCommitment: FieldConst;
    feePayerHash: FieldConst;
  };
  /**
   * Returns the public input of a zkApp transaction.
   */
  zkappPublicInput(
    txJson: string,
    accountUpdateIndex: number
  ): { accountUpdate: FieldConst; calls: FieldConst };
};
hashInputFromJson: {
  packInput(input: MlHashInput): MlArray<FieldConst>;
  timing(json: String): MlHashInput;
  permissions(json: String): MlHashInput;
  update(json: String): MlHashInput;
  accountPrecondition(json: String): MlHashInput;
  networkPrecondition(json: String): MlHashInput;
  body(json: String): MlHashInput;
};

transactionHash: {
  examplePayment(): string;
  serializePayment(payment: string): { data: Uint8Array };
  serializePaymentV1(payment: string): string;
  serializeCommon(common: string): { data: Uint8Array };
```

```
    hashPayment(payment: string): string;
    hashPaymentV1(payment: string): string;
  };
};

type FeatureFlags = {
  rangeCheck0: boolean;
  rangeCheck1: boolean;
  foreignFieldAdd: boolean;
  foreignFieldMul: boolean;
  xor: boolean;
  rot: boolean;
  lookup: boolean;
  runtimeTables: boolean;
};

type MlFeatureFlags = [
  _: 0,
  rangeCheck0: MlBool,
  rangeCheck1: MlBool,
  foreignFieldAdd: MlBool,
  foreignFieldMul: MlBool,
  xor: MlBool,
  rot: MlBool,
  lookup: MlBool,
  runtimeTables: MlBool
];

declare namespace Pickles {
  type Proof = unknown; // opaque to js
  type Statement<F> = [_: 0, publicInput: MlArray<F>, publicOutput: MlArray<F>];

  /**
   * A "rule" is a circuit plus some metadata for &nbsp;Pickles.compile&nbsp;
   */
  type Rule = {
    identifier: string;
    /**
     * The main circuit functions
     */
    main: (publicInput: MlArray<FieldVar>) => {
      publicOutput: MlArray<FieldVar>;
      previousStatements: MlArray<Statement<FieldVar>>;
      shouldVerify: MlArray<BoolVar>;
    };
    /**
     * Feature flags which enable certain custom gates
     */
    featureFlags: MlFeatureFlags;
    /**
     * Description of previous proofs to verify in this rule
     */
    proofsToVerify: MlArray<{ isSelf: true } | { isSelf: false; tag: unknown }>;
  };

  /**
   * Type to configure how Pickles should cache prover keys
   */
  type Cache = [
    _: 0,
    read: (header: SnarkKeyHeader, path: string) => MlResult<SnarkKey, MlUnit>,
    write: (
      header: SnarkKeyHeader,
      value: SnarkKey,
      path: string
    ) => MlResult<undefined, MlUnit>,
    canWrite: MlBool
  ];

  type Prover = (
    publicInput: MlArray<FieldConst>,
    previousProofs: MlArray<Proof>
  ) => Promise<[_: 0, publicOutput: MlArray<FieldConst>, proof: Proof]>;
}

declare const Pickles: {
  /**
   * This is the core API of the &nbsp;Pickles&nbsp; library, exposed from OCaml to JS. It takes a list of
circuits --
   * each in the form of a function which takes a public input &nbsp;{ accountUpdate: Field; calls: Field
}&nbsp; as argument --,
   * and augments them to add the necessary circuit logic to recursively merge in earlier proofs.
   *
   * After forming those augmented circuits in the finite field represented by &nbsp;Field&nbsp;, they gets
wrapped in a
   * single recursive circuit in the field represented by &nbsp;Scalar&nbsp;. Any SmartContract proof will go
through both of these steps,
   * so that the final proof ends up back in &nbsp;Field&nbsp;.
   *
   * The function returns the building blocks needed for SmartContract proving:
   * * &nbsp;provers&nbsp; - a list of prover functions, on for each input &nbsp;rule&nbsp;
   * * &nbsp;verify&nbsp; - a function which can verify proofs from any of the provers
   * * &nbsp;getVerificationKeyArtifact&nbsp; - a function which returns the verification key used in
&nbsp;verify&nbsp;, in base58 format, usable to deploy a zkapp
   *
   * Internal details:
   * &nbsp;compile&nbsp; calls each of the input rules four times, inside pickles.ml / compile:
   * 1) let step_data = ...    -> Pickles.Step_branch_data.create -> Pickles.Fix_domains.domains ->
Impl.constraint_system
   * 2) let step_keypair = ... -> log_step -> Snarky_log.Constraints.log -> constraint_count
   * 3) let (wrap_pk, wrap_vk) -> log_wrap -> Snarky_log.Constraints.log -> constraint_count
   * 4) let (wrap_pk, wrap_vk) -> log_wrap -> Snarky_log.Constraints.log -> constraint_count (yes, a second
time)
   */
```

```
compile: (
  rules: MlArray<Pickles.Rule>,
  config: {
    publicInputSize: number;
    publicOutputSize: number;
    storable?: Pickles.Cache;
    overrideWrapDomain?: 0 | 1 | 2;
  }
) => {
  provers: MlArray<Pickles.Prover>;
  verify: (
    statement: Pickles.Statement<FieldConst>,
    proof: Pickles.Proof
  ) => Promise<boolean>;
  tag: unknown;
  /**
   * @returns (base64 vk, hash)
   */
  getVerificationKey: () => [_: 0, data: string, hash: FieldConst];
};

verify(
  statement: Pickles.Statement<FieldConst>,
  proof: Pickles.Proof,
  verificationKey: string
): Promise<boolean>;

loadSrsFp(): WasmFpSrs;
loadSrsFq(): WasmFqSrs;

dummyProof: <N extends 0 | 1 | 2>(
  maxProofsVerified: N,
  domainLog2: number
) => [N, Pickles.Proof];

/**
 * @returns (base64 vk, hash)
 */
dummyVerificationKey: () => [_: 0, data: string, hash: FieldConst];

encodeVerificationKey: (vk: MlWrapVerificationKey) => string;
decodeVerificationKey: (vk: string) => MlWrapVerificationKey;

proofToBase64: (proof: [0 | 1 | 2, Pickles.Proof]) => string;
proofOfBase64: <N extends 0 | 1 | 2>(
  base64: string,
  maxProofsVerified: N
) => [N, Pickles.Proof];

proofToBase64Transaction: (proof: Pickles.Proof) => string;

util: {
    toMlString(s: string): MlString;
    fromMlString(s: MlString): string;
  };
};
```

</file>

<file>

## path: /src/snarky.js

url: https://github.com/o1-labs/o1js/blob/main/src/snarky.js

```
import './bindings/crypto/bindings.js';
import { getSnarky, getWasm, withThreadPool } from './bindings/js/wrapper.js';
import snarkySpec from './bindings/js/snarky-class-spec.js';
import { proxyClasses } from './bindings/js/proxy.js';

export { Snarky, Ledger, Pickles, Test, withThreadPool, getWasm };
let isReadyBoolean = true;
let isItReady = () => isReadyBoolean;

let { Snarky, Ledger, Pickles, Test } = proxyClasses(
  getSnarky,
  isItReady,
  snarkySpec
);
```

</file>

<file>

## path: /src/tests/inductive-proofs-small.ts

url: https://github.com/o1-labs/o1js/blob/main/src/tests/inductive-proofs-small.ts

```
import {
  SelfProof,
  Field,
  ZkProgram,
  isReady,
  shutdown,
  Proof,
} from '../index.js';
import { tic, toc } from '../examples/utils/tic-toc.node.js';

await isReady;

let MaxProofsVerifiedOne = ZkProgram({
  name: 'recursive-1',
  publicInput: Field,

  methods: {
    baseCase: {
      privateInputs: [],

      method(publicInput: Field) {
        publicInput.assertEquals(Field(0));
      },
    },

    mergeOne: {
      privateInputs: [SelfProof],

      method(publicInput: Field, earlierProof: SelfProof<Field, undefined>) {
        earlierProof.verify();
        earlierProof.publicInput.add(1).assertEquals(publicInput);
      },
    },
  },
});

tic('compiling program');
await MaxProofsVerifiedOne.compile();
toc();

await testRecursion(MaxProofsVerifiedOne, 1);

async function testRecursion(
  Program: typeof MaxProofsVerifiedOne,
  maxProofsVerified: number
) {
  console.log(&nbsp;testing maxProofsVerified = ${maxProofsVerified}&nbsp;);

  let ProofClass = ZkProgram.Proof(Program);

  tic('executing base case');
  let initialProof = await Program.baseCase(Field(0));
  toc();
  initialProof = testJsonRoundtrip(ProofClass, initialProof);
  initialProof.verify();
  initialProof.publicInput.assertEquals(Field(0));

  if (initialProof.maxProofsVerified != maxProofsVerified) {
    throw Error(
      &nbsp;Expected initialProof to have maxProofsVerified = ${maxProofsVerified} but has
${initialProof.maxProofsVerified}&nbsp;
    );
  }

  let p1;
  if (initialProof.maxProofsVerified === 0) return;
  tic('executing mergeOne');
  p1 = await Program.mergeOne(Field(1), initialProof);
  toc();
  p1 = testJsonRoundtrip(ProofClass, p1);
  p1.verify();
  p1.publicInput.assertEquals(Field(1));
  if (p1.maxProofsVerified != maxProofsVerified) {
    throw Error(
      &nbsp;Expected p1 to have maxProofsVerified = ${maxProofsVerified} but has
${p1.maxProofsVerified}&nbsp;
    );
  }
}

function testJsonRoundtrip(ProofClass: any, proof: Proof<Field, void>) {
  let jsonProof = proof.toJSON();
  console.log(
    'json roundtrip',
    JSON.stringify({ ...jsonProof, proof: jsonProof.proof.slice(0, 10) + '..' })
  );
  return ProofClass.fromJSON(jsonProof);
}

shutdown();
```

</file>

<file>

## path: /src/tests/inductive-proofs.ts

url: https://github.com/o1-labs/o1js/blob/main/src/tests/inductive-proofs.ts

```
import {
  SelfProof,
  Field,
  ZkProgram,
  isReady,
  shutdown,
  Proof,
} from '../index.js';
import { tic, toc } from '../examples/utils/tic-toc.node.js';

await isReady;

let MaxProofsVerifiedZero = ZkProgram({
  name: 'no-recursion',
  publicInput: Field,

  methods: {
    baseCase: {
      privateInputs: [],

      method(publicInput: Field) {
        publicInput.assertEquals(Field(0));
      },
    },
  },
});

let MaxProofsVerifiedOne = ZkProgram({
  name: 'recursive-1',
  publicInput: Field,

  methods: {
    baseCase: {
      privateInputs: [],

      method(publicInput: Field) {
        publicInput.assertEquals(Field(0));
      },
    },

    mergeOne: {
      privateInputs: [SelfProof],

      method(publicInput: Field, earlierProof: SelfProof<Field, undefined>) {
        earlierProof.verify();
        earlierProof.publicInput.add(1).assertEquals(publicInput);
      },
    },
  },
});

let MaxProofsVerifiedTwo = ZkProgram({
  name: 'recursive-2',
  publicInput: Field,

  methods: {
    baseCase: {
      privateInputs: [],

      method(publicInput: Field) {
        publicInput.assertEquals(Field(0));
      },
    },

    mergeOne: {
      privateInputs: [SelfProof],

      method(publicInput: Field, earlierProof: SelfProof<Field, undefined>) {
        earlierProof.verify();
        earlierProof.publicInput.add(1).assertEquals(publicInput);
      },
    },

    mergeTwo: {
      privateInputs: [SelfProof, SelfProof],

      method(
        publicInput: Field,
        p1: SelfProof<Field, undefined>,
        p2: SelfProof<Field, undefined>
      ) {
        p1.verify();
        p1.publicInput.add(1).assertEquals(p2.publicInput);
        p2.verify();
        p2.publicInput.add(1).assertEquals(publicInput);
      },
    },
  },
});
tic('compiling three programs');
await MaxProofsVerifiedZero.compile();
await MaxProofsVerifiedOne.compile();
await MaxProofsVerifiedTwo.compile();
toc();

await testRecursion(MaxProofsVerifiedZero as any, 0);
await testRecursion(MaxProofsVerifiedOne as any, 1);
await testRecursion(MaxProofsVerifiedTwo, 2);

async function testRecursion(
  Program: typeof MaxProofsVerifiedTwo,
  maxProofsVerified: number
) {
  console.log(&nbsp;testing maxProofsVerified = ${maxProofsVerified}&nbsp;);

  let ProofClass = ZkProgram.Proof(Program);

  tic('executing base case');
  let initialProof = await Program.baseCase(Field(0));
  toc();
  initialProof = testJsonRoundtrip(ProofClass, initialProof);
  initialProof.verify();
  initialProof.publicInput.assertEquals(Field(0));

  if (initialProof.maxProofsVerified != maxProofsVerified) {
    throw Error(
      &nbsp;Expected initialProof to have maxProofsVerified = ${maxProofsVerified} but has
${initialProof.maxProofsVerified}&nbsp;
    );
  }
  let p1, p2;
  if (initialProof.maxProofsVerified === 0) return;

  tic('executing mergeOne');
  p1 = await Program.mergeOne(Field(1), initialProof);
  toc();
  p1 = testJsonRoundtrip(ProofClass, p1);
  p1.verify();
  p1.publicInput.assertEquals(Field(1));
  if (p1.maxProofsVerified != maxProofsVerified) {
    throw Error(
      &nbsp;Expected p1 to have maxProofsVerified = ${maxProofsVerified} but has
${p1.maxProofsVerified}&nbsp;
    );
  }

  if (initialProof.maxProofsVerified === 1) return;
  tic('executing mergeTwo');
  p2 = await Program.mergeTwo(Field(2), initialProof, p1);
  toc();
  p2 = testJsonRoundtrip(ProofClass, p2);
  p2.verify();
  p2.publicInput.assertEquals(Field(2));
  if (p2.maxProofsVerified != maxProofsVerified) {
    throw Error(
      &nbsp;Expected p2 to have maxProofsVerified = ${maxProofsVerified} but has
${p2.maxProofsVerified}&nbsp;
    );
  }
}

function testJsonRoundtrip(ProofClass: any, proof: Proof<Field, void>) {
  let jsonProof = proof.toJSON();
  console.log(
    'json roundtrip',
    JSON.stringify({ ...jsonProof, proof: jsonProof.proof.slice(0, 10) + '..' })
  );
  return ProofClass.fromJSON(jsonProof);
}

shutdown();
```

</file>

<file>

## path: /tests/artifacts/config/storageState.json

url: https://github.com/o1-labs/o1js/blob/main/tests/artifacts/config/storageState.json

```
{
  "cookies": [],
  "origins": []
}
```

</file>

<file>

## path: /tests/artifacts/html/on-chain-state-mgmt-zkapp-ui.html

url: https://github.com/o1-labs/o1js/blob/main/tests/artifacts/html/on-chain-state-mgmt-zkapp-ui.html

```
<!DOCTYPE html>
<html lang="en">

<head>
  <meta charset="utf-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1, shrink-to-fit=no" />
  <meta name="description" content="" />
  <meta name="author" content="" />
  <title>UI for the On-Chain State Management zkApp</title>
  <style>
    table,
    form,
    th,
    td {
      border: 1px solid black;
      border-collapse: collapse
    }
  </style>
</head>

<body>
  <div>
    <div>
      <h3>UI for the On-Chain State Management zkApp</h3>
    </div>
    <hr />
    <div>
      <h4>zkApp Management</h4>
      <div>
        <button type="button" id="deployButton">Deploy zkApp</button>
        <hr />
        <form style="width: 50%" action="" method="POST" id="zkAppUpdateForm">
          <div>
            <label for="zkAppStateValue"><u>zkApp State</u>:</label>
            <input type="number" id="zkAppStateValue" name="zkAppStateValue" placeholder="Enter new
zkApp state"
              value="" required />
          </div>
          <br />
          <button type="submit" id="updateButton">Update zkApp State</button>
        </form>
      </div>
    </div>
    <hr />
    <div>
      <h4>Application Data</h4>
      <table>
        <caption></caption>
        <tr>
          <th scope="col"></th>
          <th scope="col"></th>
        </tr>
        <tbody>
          <tr>
            <td style="width: 20%">
              <span>zkAppState[0]</span>:
            </td>
            <td>
              <span id="zkAppStateContainer">No data available yet.</span>
            </td>
          </tr>
          <tr>
            <td style="width: 20%">
              <span>Events</span>:
            </td>
            <td>
              <span id="eventsContainer">No data available yet.</span>
            </td>
          </tr>
        </tbody>
      </table>
      <br />
      <button type="button" id="clearEventsButton">Clear Events</button>
    </div>
  </div>

  <script type="module" src="./on-chain-state-mgmt-zkapp-ui.js"></script>
</body>

</html>
```

</file>

<file>

## path: /tests/artifacts/javascript/e2eTestsHelpers.js

url: https://github.com/o1-labs/o1js/blob/main/tests/artifacts/javascript/e2eTestsHelpers.js

```
const logEvents = (message, eventsContainer) => {
  const dateNow = new Date().toLocaleString();
  const previousMessage = eventsContainer.innerHTML.replace(
    /No data available yet./g,
    ''
  );

  eventsContainer.innerHTML = &nbsp;${dateNow} - ${message}<br>${previousMessage}&nbsp;;
};

export { logEvents };
```

</file>

<file>

## path: /tests/artifacts/javascript/on-chain-state-mgmt-zkapp-ui.js

url: https://github.com/o1-labs/o1js/blob/main/tests/artifacts/javascript/on-chain-state-mgmt-zkapp-ui.js

```
import { logEvents } from './e2eTestsHelpers.js';
import {
  adminPrivateKey,
  HelloWorld,
} from './examples/zkapps/hello_world/hello_world.js';
import {
  AccountUpdate,
  Field,
  isReady,
  Mina,
  PrivateKey,
  verify,
} from './index.js';

await isReady;

const deployButton = document.querySelector('#deployButton');
const updateButton = document.querySelector('#updateButton');
const clearEventsButton = document.querySelector('#clearEventsButton');
const eventsContainer = document.querySelector('#eventsContainer');
const zkAppStateContainer = document.querySelector('#zkAppStateContainer');

logEvents(
  &nbsp;o1js initialized after ${performance.now().toFixed(2)}ms&nbsp;,
  eventsContainer
);

// Setup local ledger
let Local = Mina.LocalBlockchain();
Mina.setActiveInstance(Local);
// Test account that pays all the fees
```

```
const feePayerKey = Local.testAccounts[0].privateKey;
const feePayer = Local.testAccounts[0].publicKey;
// zkApp account
const zkAppPrivateKey = PrivateKey.random();
const zkAppAddress = zkAppPrivateKey.toPublicKey();
const zkAppInstance = new HelloWorld(zkAppAddress);
let verificationKey = null;

deployButton.addEventListener('click', async () => {
  deployButton.disabled = true;

  logEvents('Deploying zkApp...', eventsContainer);

  try {
    await HelloWorld.compile();
    const deploymentTransaction = await Mina.transaction(feePayer, () => {
      if (!eventsContainer.innerHTML.includes('zkApp Deployed successfully')) {
        AccountUpdate.fundNewAccount(feePayer);
      }
      zkAppInstance.deploy();
    });

    await deploymentTransaction.sign([feePayerKey, zkAppPrivateKey]).send();
    const initialState =
      Mina.getAccount(zkAppAddress).zkapp?.appState?.[0].toString();
    zkAppStateContainer.innerHTML = initialState;
    logEvents(&nbsp;Initial zkApp State: ${initialState}&nbsp;, eventsContainer);
    logEvents('zkApp Deployed successfully!', eventsContainer);
  } catch (exception) {
    logEvents(
      &nbsp;zkApp Deployment failure: ${exception.message}&nbsp;,
      eventsContainer
    );
    console.log(exception);
  }

  deployButton.disabled = false;
});

updateButton.addEventListener('click', async (event) => {
  event.preventDefault();
  updateButton.disabled = true;

  const formData = JSON.stringify(
    Object.fromEntries(new FormData(document.querySelector('#zkAppUpdateForm')))
  );
  const zkAppStateValue = document.querySelector('#zkAppStateValue');

  try {
    const currentState =
      Mina.getAccount(zkAppAddress).zkapp?.appState?.[0].toString();
    logEvents(
      &nbsp;Updating zkApp State from ${currentState} to ${zkAppStateValue.value} with Admin Private Key
and using form data: ${formData}...&nbsp;,
      eventsContainer
    );
    const transaction = await Mina.transaction(feePayer, () => {
      zkAppInstance.update(
        Field(parseInt(zkAppStateValue.value)),
        adminPrivateKey
      );
    });

    const [proof] = await transaction.prove();

    if (verificationKey) {
      let isVerified = await verify(proof, verificationKey.data);
      if (!isVerified) throw Error('Proof verification failed');
    }

    await transaction.sign([feePayerKey]).send();

    const newState =
      Mina.getAccount(zkAppAddress).zkapp?.appState?.[0].toString();
    zkAppStateContainer.innerHTML = newState;
    logEvents(
      &nbsp;zkApp State successfully updated to: ${newState}!&nbsp;,
      eventsContainer
    );
  } catch (exception) {
    logEvents(
      &nbsp;zkApp State Update failure: ${exception.message}&nbsp;,
      eventsContainer
    );
    console.log(exception);
  }

  updateButton.disabled = false;
  return false;
});

clearEventsButton.addEventListener('click', async () => {
  eventsContainer.innerHTML = 'No data available yet.';
});
```

</file>

<file>

## path: /tests/fixtures/on-chain-state-mgmt-zkapp.ts

url: https://github.com/o1-labs/o1js/blob/main/tests/fixtures/on-chain-state-mgmt-zkapp.ts

```
import { test as base } from '@playwright/test';
import { OnChainStateMgmtZkAppPage } from '../pages/on-chain-state-mgmt-zkapp.js';

type OnChainStateMgmtZkAppFixture = {
  onChainStateMgmtZkAppPage: OnChainStateMgmtZkAppPage;
};

export const test = base.extend<OnChainStateMgmtZkAppFixture>({
  onChainStateMgmtZkAppPage: async ({ page }, use) => {
    await use(new OnChainStateMgmtZkAppPage(page));
  },
});
```

</file>

<file>

## path: /tests/integration/inductive-proofs.js

url: https://github.com/o1-labs/o1js/blob/main/tests/integration/inductive-proofs.js

```
import {
  SelfProof,
  Field,
  ZkProgram,
  isReady,
  shutdown,
} from '../../dist/node/index.js';
import { tic, toc } from './tictoc.js';

await isReady;

let MaxProofsVerifiedZero = ZkProgram({
  name: 'no-recursion',
  publicInput: Field,

  methods: {
    baseCase: {
      privateInputs: [],

      method(publicInput) {
        publicInput.assertEquals(Field(0));
      },
    },
  },
});

let MaxProofsVerifiedOne = ZkProgram({
  name: 'recursive-1',
  publicInput: Field,

  methods: {
    baseCase: {
      privateInputs: [],

      method(publicInput) {
        publicInput.assertEquals(Field(0));
      },
    },

    mergeOne: {
      privateInputs: [SelfProof],

      method(publicInput, earlierProof) {
        earlierProof.verify();
        earlierProof.publicInput.add(1).assertEquals(publicInput);
      },
    },
  },
});

let MaxProofsVerifiedTwo = ZkProgram({
  name: 'recursive-2',
  publicInput: Field,

  methods: {
    baseCase: {
      privateInputs: [],

      method(publicInput) {
        publicInput.assertEquals(Field(0));
      },
    },

    mergeOne: {
      privateInputs: [SelfProof],

      method(publicInput, earlierProof) {
        earlierProof.verify();
        earlierProof.publicInput.add(1).assertEquals(publicInput);
      },
    },

    mergeTwo: {
      privateInputs: [SelfProof, SelfProof],

      method(publicInput, p1, p2) {
        p1.verify();
        p1.publicInput.add(1).assertEquals(p2.publicInput);
        p2.verify();
        p2.publicInput.add(1).assertEquals(publicInput);
      },
    },
  },
});
tic('compiling three programs..');
await MaxProofsVerifiedZero.compile();
await MaxProofsVerifiedOne.compile();
await MaxProofsVerifiedTwo.compile();
toc();

await testRecursion(MaxProofsVerifiedZero, 0);
await testRecursion(MaxProofsVerifiedOne, 1);
await testRecursion(MaxProofsVerifiedTwo, 2);

async function testRecursion(Program, maxProofsVerified) {
  console.log(&nbsp;testing maxProofsVerified = ${maxProofsVerified}&nbsp;);

  let ProofClass = ZkProgram.Proof(Program);

  tic('executing base case..');
  let initialProof = await Program.baseCase(Field(0));
  toc();
  initialProof = testJsonRoundtrip(ProofClass, initialProof);
  initialProof.verify();
  initialProof.publicInput.assertEquals(Field(0));

  if (initialProof.maxProofsVerified != maxProofsVerified) {
    throw Error(
      &nbsp;Expected initialProof to have maxProofsVerified = ${maxProofsVerified} but has
${initialProof.maxProofsVerified}&nbsp;
    );
  }

  let p1, p2;
  if (initialProof.maxProofsVerified == 0) return;

  tic('executing mergeOne..');
  p1 = await Program.mergeOne(Field(1), initialProof);
  toc();
  p1 = testJsonRoundtrip(ProofClass, p1);
  p1.verify();
  p1.publicInput.assertEquals(Field(1));
  if (p1.maxProofsVerified != maxProofsVerified) {
    throw Error(
      &nbsp;Expected p1 to have maxProofsVerified = ${maxProofsVerified} but has
${p1.maxProofsVerified}&nbsp;
    );
  }

  if (initialProof.maxProofsVerified == 1) return;
  tic('executing mergeTwo..');
  p2 = await Program.mergeTwo(Field(2), initialProof, p1);
  toc();
  p2 = testJsonRoundtrip(ProofClass, p2);
  p2.verify();
  p2.publicInput.assertEquals(Field(2));
  if (p2.maxProofsVerified != maxProofsVerified) {
    throw Error(
      &nbsp;Expected p2 to have maxProofsVerified = ${maxProofsVerified} but has
${p2.maxProofsVerified}&nbsp;
    );
  }
}

function testJsonRoundtrip(ProofClass, proof) {
  let jsonProof = proof.toJSON();
  console.log(
    'json roundtrip',
    JSON.stringify({ ...jsonProof, proof: jsonProof.proof.slice(0, 10) + '..' })
  );
  return ProofClass.fromJSON(jsonProof);
}

shutdown();
```

</file>

<file>

## path: /tests/integration/package.json

url: https://github.com/o1-labs/o1js/blob/main/tests/integration/package.json

```
{
  "name": "o1js-integration-tests",
  "version": "0.0.0",
  "type": "module",
  "dependencies": {}
}
```

</file>

<file>

## path: /tests/integration/simple-zkapp-mock-apply.js

url: https://github.com/o1-labs/o1js/blob/main/tests/integration/simple-zkapp-mock-apply.js

```
import {
  Field,
  declareState,
  declareMethods,
  State,
  PrivateKey,
  SmartContract,
  isReady,
  shutdown,
  Mina,
  Permissions,
  verify,
  AccountUpdate,
} from '../../dist/node/index.js';
import { tic, toc } from './tictoc.js';

await isReady;

// declare the zkapp
const initialState = Field(1);
class SimpleZkapp extends SmartContract {
  constructor(address) {
    super(address);
    this.x = State();
  }

  deploy(args) {
    super.deploy(args);
    this.account.permissions.set({
      ...Permissions.default(),
      editState: Permissions.proofOrSignature(),
    });
  }

  init() {
    super.init();
    this.x.set(initialState);
  }

  update(y) {
    let x = this.x.get();
    this.x.assertEquals(x);
    y.assertGreaterThan(0);
    this.x.set(x.add(y));
  }
}
// note: this is our non-typescript way of doing what our decorators do
declareState(SimpleZkapp, { x: Field });
declareMethods(SimpleZkapp, { init: [], update: [Field] });

// setup mock mina
let Local = Mina.LocalBlockchain();
Mina.setActiveInstance(Local);
let { publicKey: sender, privateKey: senderKey } = Local.testAccounts[0];

let zkappKey = PrivateKey.random();
let zkappAddress = zkappKey.toPublicKey();
let zkapp = new SimpleZkapp(zkappAddress);
```

```
tic('compute circuit digest');
SimpleZkapp.digest();
toc();

tic('compile smart contract');
let { verificationKey } = await SimpleZkapp.compile();
toc();

tic('create deploy transaction (with proof)');
let deployTx = await Mina.transaction(sender, () => {
  AccountUpdate.fundNewAccount(sender);
  zkapp.deploy();
});
let [, , proof] = await deployTx.prove();
deployTx.sign([zkappKey, senderKey]);
toc();

tic('verify transaction proof');
let ok = await verify(proof, verificationKey.data);
toc();
console.log('did proof verify?', ok);
if (!ok) throw Error("proof didn't verify");

tic('apply deploy transaction');
await deployTx.send();
toc();

// check that deploy and initialize txns were applied
let zkappState = zkapp.x.get();
zkappState.assertEquals(1);
console.log('got initial state: ' + zkappState);

tic('create update transaction (no proof)');
let tx = await Mina.transaction(sender, () => {
  zkapp.update(Field(2));
  zkapp.requireSignature();
});
tx.sign([senderKey, zkappKey]);
toc();

tic('apply update transaction (no proof)');
await tx.send();
toc();

// check that first update txn was applied
zkappState = zkapp.x.get();
zkappState.assertEquals(3);
console.log('got updated state: ' + zkappState);

tic('create update transaction (with proof)');
tx = await Mina.transaction(sender, () => {
  zkapp.update(Field(2));
});
[proof] = await tx.prove();
tx.sign([senderKey]);
toc();

tic('verify transaction proof');
ok = await verify(proof, verificationKey.data);
toc();
console.log('did proof verify?', ok);
if (!ok) throw Error("proof didn't verify");

tic('apply update transaction (with proof)');
await tx.send();
toc();

// check that second update txn was applied
zkappState = zkapp.x.get();
zkappState.assertEquals(5);
console.log('got updated state: ' + zkappState);

shutdown();
```

</file>

<file>

## path: /tests/integration/simple-zkapp.js

url: https://github.com/o1-labs/o1js/blob/main/tests/integration/simple-zkapp.js

```
import {
  Field,
  declareState,
  declareMethods,
  State,
  PrivateKey,
  SmartContract,
  isReady,
  Mina,
  PublicKey,
  UInt64,
  AccountUpdate,
  Bool,
  shutdown,
  Permissions,
  fetchAccount,
} from 'o1js';

await isReady;
```

```
class NotSoSimpleZkapp extends SmartContract {
  events = { update: Field, payout: UInt64, payoutReceiver: PublicKey };

  constructor(address) {
    super(address);
    this.x = State();
  }

  init() {
    super.init();
    this.x.set(initialState);
    this.account.permissions.set({
      ...Permissions.default(),
      send: Permissions.proof(),
      editState: Permissions.proof(),
    });
  }

  update(y) {
    let x = this.x.get();
    this.x.assertEquals(x);
    y.assertGreaterThan(0);
    this.x.set(x.add(y));
  }

  payout(caller) {
    let callerAddress = caller.toPublicKey();
    callerAddress.assertEquals(privilegedAddress);

    let callerAccountUpdate = AccountUpdate.create(callerAddress);
    callerAccountUpdate.account.isNew.assertEquals(Bool(true));

    let balance = this.account.balance.get();
    this.account.balance.assertEquals(balance);
    let halfBalance = balance.div(2);
    this.send({ to: callerAccountUpdate, amount: halfBalance });

    // emit some events
    this.emitEvent('payoutReceiver', callerAddress);
    this.emitEvent('payout', halfBalance);
  }

  deposit(amount) {
    let senderUpdate = AccountUpdate.createSigned(this.sender);
    senderUpdate.send({ to: this, amount });
  }
}
// note: this is our non-typescript way of doing what our decorators do
declareState(NotSoSimpleZkapp, { x: Field });
declareMethods(NotSoSimpleZkapp, {
  update: [Field],
  payout: [PrivateKey],
  deposit: [UInt64],
});

// slightly adjusted polling parameters for tx.wait()
const waitParams = {
  maxAttempts: 30,
  interval: 45000,
};

// parse command line; for local testing, use random keys as fallback
let [feePayerKeyBase58, graphql_uri] = process.argv.slice(2);

let isLocal = false;

if (feePayerKeyBase58 === 'local') {
  isLocal = true;
  let LocalNetwork = Mina.LocalBlockchain(graphql_uri);
  Mina.setActiveInstance(LocalNetwork);
  let { privateKey } = LocalNetwork.testAccounts[0];
  feePayerKeyBase58 = privateKey.toBase58();
} else {
  if (!graphql_uri) throw Error('Graphql uri is undefined, aborting');
  if (!feePayerKeyBase58) throw Error('Fee payer key is undefined, aborting');
  let LocalNetwork = Mina.Network(graphql_uri);
  Mina.setActiveInstance(LocalNetwork);
}

let zkappKey = PrivateKey.random();
let zkappAddress = zkappKey.toPublicKey();

let feePayerKey = PrivateKey.fromBase58(feePayerKeyBase58);
let feePayerAddress = feePayerKey.toPublicKey();

if (!isLocal) {
  let res = await fetchAccount({
    publicKey: feePayerAddress,
  });
  if (res.error) {
    throw Error(
      &nbsp;The fee payer account needs to be funded in order for the script to succeed! Please provide the
private key of an already funded account. ${feePayerAddress.toBase58()}, ${feePayerKeyBase58}\n\n${
        res.error.message
      }&nbsp;
    );
  }
}

// a special account that is allowed to pull out half of the zkapp balance, once
let privilegedKey = PrivateKey.random();
let privilegedAddress = privilegedKey.toPublicKey();

let zkappTargetBalance = 10_000_000_000;
let initialBalance = zkappTargetBalance;
let initialState = Field(1);

console.log(
  &nbsp;simple-zkapp.js: Running with zkapp address ${zkappKey
    .toPublicKey()
    .toBase58()}, fee payer address ${feePayerAddress.toBase58()} and graphql uri ${graphql_uri}\n\n&nbsp;
);

console.log(&nbsp;simple-zkapp.js: Starting integration test\n&nbsp;);

let zkapp = new NotSoSimpleZkapp(zkappAddress);
await NotSoSimpleZkapp.compile();

console.log('deploying contract\n');
let tx = await Mina.transaction(
  { sender: feePayerAddress, fee: 100_000_000 },
  () => {
    AccountUpdate.fundNewAccount(feePayerAddress);

    zkapp.deploy();
  }
);
await tx.prove();
await (await tx.sign([feePayerKey, zkappKey]).send()).wait(waitParams);

if (!isLocal) await fetchAccount({ publicKey: zkappAddress });
let zkappAccount = Mina.getAccount(zkappAddress);

// we deployed the contract with an initial state of 1
expectAssertEquals(zkappAccount.zkapp.appState[0], Field(1));

// the fresh zkapp account shouldn't have any funds
expectAssertEquals(zkappAccount.balance, UInt64.from(0));

console.log('deposit funds\n');
tx = await Mina.transaction(
  { sender: feePayerAddress, fee: 100_000_000 },
  () => {
    zkapp.deposit(UInt64.from(initialBalance));
  }
);
await tx.prove();
await (await tx.sign([feePayerKey]).send()).wait(waitParams);

if (!isLocal) await fetchAccount({ publicKey: zkappAddress });
zkappAccount = Mina.getAccount(zkappAddress);

// we deposit 10_000_000_000 funds into the zkapp account
expectAssertEquals(zkappAccount.balance, UInt64.from(initialBalance));

console.log('update 1\n');
```

```
tx = await Mina.transaction(
  { sender: feePayerAddress, fee: 100_000_000 },
  () => {
    zkapp.update(Field(30));
  }
);
await tx.prove();
await (await tx.sign([feePayerKey]).send()).wait(waitParams);

console.log('update 2\n');
tx = await Mina.transaction(
  { sender: feePayerAddress, fee: 100_000_000 },
  () => {
    zkapp.update(Field(100));
  }
);
await tx.prove();
await (await tx.sign([feePayerKey]).send()).wait(waitParams);

if (!isLocal) await fetchAccount({ publicKey: zkappAddress });
zkappAccount = Mina.getAccount(zkappAddress);

// no balance change expected
expectAssertEquals(zkappAccount.balance, UInt64.from(initialBalance));

// we updated the zkapp state to 131
expectAssertEquals(zkappAccount.zkapp.appState[0], Field(131));

console.log('payout 1\n');
tx = await Mina.transaction(
  { sender: feePayerAddress, fee: 100_000_000 },
  () => {
    AccountUpdate.fundNewAccount(feePayerAddress);
    zkapp.payout(privilegedKey);
  }
);
await tx.prove();
await (await tx.sign([feePayerKey]).send()).wait(waitParams);

if (!isLocal) await fetchAccount({ publicKey: zkappAddress });
zkappAccount = Mina.getAccount(zkappAddress);

// we withdraw (payout) half of the initial balance
expectAssertEquals(zkappAccount.balance, UInt64.from(initialBalance / 2));

console.log('payout 2 (expected to fail)\n');
tx = await Mina.transaction(
  { sender: feePayerAddress, fee: 100_000_000 },
  () => {
    zkapp.payout(privilegedKey);
  }
);
```

```
await tx.prove();

// this tx should fail
try {
  let txId = await tx.sign([feePayerKey]).send();
  await txId.wait(waitParams);
} catch (err) {
  // throw if this is not the expected error
  if (!err.message.includes('Account_is_new_precondition_unsatisfied')) {
    throw err;
  }
}

// although we just checked above that the tx failed, I just would like to double-check that anyway (cross
checking logic)
if (!isLocal) await fetchAccount({ publicKey: zkappAddress });
zkappAccount = Mina.getAccount(zkappAddress);

// checking that state hasn't changed - we expect the tx to fail so the state should equal previous state
expectAssertEquals(zkappAccount.balance, UInt64.from(initialBalance / 2));

function expectAssertEquals(actual, expected) {
  try {
    actual.assertEquals(expected);
  } catch (error) {
    throw Error(
      &nbsp;Expected value ${expected.toString()}, but got ${actual.toString()}&nbsp;
    );
  }
}

shutdown();
```

</file>

<file>

## path: /tests/integration/tictoc.js

url: https://github.com/o1-labs/o1js/blob/main/tests/integration/tictoc.js

```
// helper for printing timings

export { tic, toc };

let timingStack = [];
let i = 0;

function tic(label = &nbsp;Run command ${i++}&nbsp;) {
  process.stdout.write(&nbsp;${label}... &nbsp;);
  timingStack.push([label, Date.now()]);
}

function toc() {
  let [label, start] = timingStack.pop();
  let time = (Date.now() - start) / 1000;
  process.stdout.write(&nbsp;\r${label}... ${time.toFixed(3)} sec\n&nbsp;);
}
```

</file>

<file>

## path: /tests/on-chain-state-mgmt-zkapp-ui.spec.ts

url: https://github.com/o1-labs/o1js/blob/main/tests/on-chain-state-mgmt-zkapp-ui.spec.ts

```
import { test } from './fixtures/on-chain-state-mgmt-zkapp.js';

test.describe('On-Chain State Management zkApp UI', () => {
  test('should load page and initialize o1js', async ({
    onChainStateMgmtZkAppPage,
  }) => {
    await onChainStateMgmtZkAppPage.goto();
    await onChainStateMgmtZkAppPage.checkO1jsInitialization();
  });

  test('should fail to update account state since zkApp was not yet deployed', async ({
    onChainStateMgmtZkAppPage,
  }) => {
    test.skip(process.env.CI === 'true', 'Skipping test in CI');

    await onChainStateMgmtZkAppPage.goto();
    await onChainStateMgmtZkAppPage.checkO1jsInitialization();
    await onChainStateMgmtZkAppPage.updateZkAppState('3');
    await onChainStateMgmtZkAppPage.checkZkAppStateUpdateFailureByUnknownAccount();
  });

  test('should compile and deploy zkApp', async ({
    onChainStateMgmtZkAppPage,
  }) => {
    await onChainStateMgmtZkAppPage.goto();
    await onChainStateMgmtZkAppPage.checkO1jsInitialization();
    await onChainStateMgmtZkAppPage.compileAndDeployZkApp();
    await onChainStateMgmtZkAppPage.checkDeployedZkApp();
  });

  test('should prove transaction and update zkApp account state', async ({
    onChainStateMgmtZkAppPage,
  }) => {
    const currentAccountState = '2';
    const newAccountState = '4';

    await onChainStateMgmtZkAppPage.goto();
    await onChainStateMgmtZkAppPage.checkO1jsInitialization();
    await onChainStateMgmtZkAppPage.compileAndDeployZkApp();
    await onChainStateMgmtZkAppPage.checkDeployedZkApp();
    await onChainStateMgmtZkAppPage.updateZkAppState(newAccountState);
    await onChainStateMgmtZkAppPage.checkUpdatedZkAppState(
      currentAccountState,
      newAccountState
    );
  });

  test.skip('should re-deploy zkApp', async ({ onChainStateMgmtZkAppPage }) => {
    test.skip(process.env.CI === 'true', 'Skipping test in CI');

    const currentAccountState = '2';
    const newAccountState = '4';

    await onChainStateMgmtZkAppPage.goto();
    await onChainStateMgmtZkAppPage.checkO1jsInitialization();
    await onChainStateMgmtZkAppPage.compileAndDeployZkApp();
    await onChainStateMgmtZkAppPage.checkDeployedZkApp();
    await onChainStateMgmtZkAppPage.updateZkAppState(newAccountState);
    await onChainStateMgmtZkAppPage.checkUpdatedZkAppState(
      currentAccountState,
      newAccountState
    );
    await onChainStateMgmtZkAppPage.compileAndDeployZkApp();
    await onChainStateMgmtZkAppPage.checkDeployedZkApp();
  });

  test.skip('should fail to re-deploy zkApp by fee excess', async ({
    onChainStateMgmtZkAppPage,
  }) => {
    test.skip(process.env.CI === 'true', 'Skipping test in CI');

    await onChainStateMgmtZkAppPage.goto();
    await onChainStateMgmtZkAppPage.checkO1jsInitialization();
    await onChainStateMgmtZkAppPage.compileAndDeployZkApp();
    await onChainStateMgmtZkAppPage.checkDeployedZkApp();
    await onChainStateMgmtZkAppPage.clearEvents();
    await onChainStateMgmtZkAppPage.compileAndDeployZkApp();
```

```
    await onChainStateMgmtZkAppPage.checkZkAppDeploymentFailureByFeeExcess();
  });

  test.skip('should fail to update account state by zkApp constraint', async ({
    onChainStateMgmtZkAppPage,
  }) => {
    test.skip(process.env.CI === 'true', 'Skipping test in CI');

    let currentAccountState = '2';
    let newAccountState = '4';
    const nextAccountState = '16';

    await onChainStateMgmtZkAppPage.goto();
    await onChainStateMgmtZkAppPage.checkO1jsInitialization();
    await onChainStateMgmtZkAppPage.compileAndDeployZkApp();
    await onChainStateMgmtZkAppPage.checkDeployedZkApp();
    await onChainStateMgmtZkAppPage.updateZkAppState(newAccountState);
    await onChainStateMgmtZkAppPage.checkUpdatedZkAppState(
      currentAccountState,
      newAccountState
    );
    currentAccountState = newAccountState;
    newAccountState = '1';
    await onChainStateMgmtZkAppPage.updateZkAppState(newAccountState);
    await onChainStateMgmtZkAppPage.checkZkAppStateUpdateFailureByStateConstraint(
      currentAccountState,
      nextAccountState,
      newAccountState
    );
  });
});
```

</file>

<file>

## path: /tests/pages/on-chain-state-mgmt-zkapp.ts

url: https://github.com/o1-labs/o1js/blob/main/tests/pages/on-chain-state-mgmt-zkapp.ts

```
import { expect, type Locator, type Page } from '@playwright/test';

export class OnChainStateMgmtZkAppPage {
  readonly page: Page;
  readonly deployButton: Locator;
  readonly updateButton: Locator;
  readonly clearEventsButton: Locator;
  readonly zkAppStateValue: Locator;
  readonly eventsContainer: Locator;
  readonly zkAppStateContainer: Locator;
```

```
constructor(page: Page) {
  this.page = page;
  this.deployButton = page.locator('button[id="deployButton"]');
  this.updateButton = page.locator('button[id="updateButton"]');
  this.clearEventsButton = page.locator('button[id="clearEventsButton"]');
  this.zkAppStateValue = page.locator('input[id="zkAppStateValue"]');
  this.eventsContainer = page.locator('span[id="eventsContainer"]');
  this.zkAppStateContainer = page.locator('span[id="zkAppStateContainer"]');
}

async goto() {
  await this.page.goto('/on-chain-state-mgmt-zkapp-ui.html');
}

async compileAndDeployZkApp() {
  await this.deployButton.click();
}

async updateZkAppState(value: string) {
  await this.zkAppStateValue.fill(value);
  await this.updateButton.click();
}

async clearEvents() {
  await this.clearEventsButton.click();
}

async checkO1jsInitialization() {
  await expect(this.eventsContainer).toContainText('o1js initialized after');
}

async checkDeployedZkApp() {
  await expect(this.eventsContainer).toContainText('Deploying zkApp');
  await expect(this.eventsContainer).toContainText('Initial zkApp State: 2');
  await expect(this.eventsContainer).toContainText(
    'zkApp Deployed successfully!'
  );
  await expect(this.zkAppStateContainer).toHaveText('2');
}

async checkUpdatedZkAppState(currentValue: string, nextValue: string) {
  await expect(this.eventsContainer).toContainText(
    &nbsp;Updating zkApp State from ${currentValue} to ${nextValue}&nbsp;
  );
  await expect(this.eventsContainer).toContainText(
    &nbsp;zkApp State successfully updated to: ${nextValue}!&nbsp;
  );
  await expect(this.zkAppStateContainer).toHaveText(nextValue);
}

async checkZkAppDeploymentFailureByFeeExcess() {
  await expect(this.eventsContainer).toContainText('Deploying zkApp');
  await expect(this.eventsContainer).toContainText(
   'zkApp Deployment failure'
  );
  await expect(this.eventsContainer).toContainText('Invalid_fee_excess');
 }

 async checkZkAppStateUpdateFailureByUnknownAccount() {
  await expect(this.eventsContainer).toContainText(
   'zkApp State Update failure'
  );
  await expect(this.eventsContainer).toContainText(
   'Could not find account for public key'
  );
  await expect(this.zkAppStateContainer).toHaveText('No data available yet.');
 }

 async checkZkAppStateUpdateFailureByStateConstraint(
  actualValue: string,
  nextValue: string,
  expectedValue: string
 ) {
  await expect(this.eventsContainer).toContainText(
   &nbsp;Updating zkApp State from ${actualValue} to ${expectedValue}&nbsp;
  );
  await expect(this.eventsContainer).toContainText(
   &nbsp;zkApp State Update failure: Field.assertEquals(): ${nextValue} != ${expectedValue}&nbsp;
  );
  await expect(this.zkAppStateContainer).toHaveText(actualValue);
 }
}
```

</file>

<file>

## path: /tests/vk-regression/plain-constraint-system.ts

url: https://github.com/o1-labs/o1js/blob/main/tests/vk-regression/plain-constraint-system.ts

```
import { Field, Group, Gadgets, Provable, Scalar } from 'o1js';

export { GroupCS, BitwiseCS };

const GroupCS = constraintSystem('Group Primitive', {
 add() {
  let g1 = Provable.witness(Group, () => Group.generator);
  let g2 = Provable.witness(Group, () => Group.generator);
  g1.add(g2);
 },
 sub() {
  let g1 = Provable.witness(Group, () => Group.generator);
```

```
    let g2 = Provable.witness(Group, () => Group.generator);
    g1.sub(g2);
  },
  scale() {
    let g1 = Provable.witness(Group, () => Group.generator);
    let s = Provable.witness(Scalar, () => Scalar.from(5n));
    g1.scale(s);
  },
  equals() {
    let g1 = Provable.witness(Group, () => Group.generator);
    let g2 = Provable.witness(Group, () => Group.generator);
    g1.equals(g2).assertTrue();
    g1.equals(g2).assertFalse();
    g1.equals(g2).assertEquals(true);
    g1.equals(g2).assertEquals(false);
  },
  assertions() {
    let g1 = Provable.witness(Group, () => Group.generator);
    let g2 = Provable.witness(Group, () => Group.generator);
    g1.assertEquals(g2);
  },
});

const BitwiseCS = constraintSystem('Bitwise Primitive', {
  rot() {
    let a = Provable.witness(Field, () => new Field(12));
    Gadgets.rotate(a, 2, 'left');
    Gadgets.rotate(a, 2, 'right');
    Gadgets.rotate(a, 4, 'left');
    Gadgets.rotate(a, 4, 'right');
  },
  xor() {
    let a = Provable.witness(Field, () => new Field(5n));
    let b = Provable.witness(Field, () => new Field(5n));
    Gadgets.xor(a, b, 16);
    Gadgets.xor(a, b, 32);
    Gadgets.xor(a, b, 48);
    Gadgets.xor(a, b, 64);
  },
  notUnchecked() {
    let a = Provable.witness(Field, () => new Field(5n));
    Gadgets.not(a, 16, false);
    Gadgets.not(a, 32, false);
    Gadgets.not(a, 48, false);
    Gadgets.not(a, 64, false);
  },
  notChecked() {
    let a = Provable.witness(Field, () => new Field(5n));
    Gadgets.not(a, 16, true);
    Gadgets.not(a, 32, true);
    Gadgets.not(a, 48, true);
    Gadgets.not(a, 64, true);
  },
  leftShift() {
    let a = Provable.witness(Field, () => new Field(12));
    Gadgets.leftShift(a, 2);
    Gadgets.leftShift(a, 4);
  },
  rightShift() {
    let a = Provable.witness(Field, () => new Field(12));
    Gadgets.rightShift(a, 2);
    Gadgets.rightShift(a, 4);
  },
  and() {
    let a = Provable.witness(Field, () => new Field(5n));
    let b = Provable.witness(Field, () => new Field(5n));
    Gadgets.and(a, b, 16);
    Gadgets.and(a, b, 32);
    Gadgets.and(a, b, 48);
    Gadgets.and(a, b, 64);
  },
});

// mock ZkProgram API for testing

function constraintSystem(
  name: string,
  obj: { [K: string]: (...args: any) => void }
) {
  let methodKeys = Object.keys(obj);

  return {
    analyzeMethods() {
      let cs: Record<
        string,
        {
          rows: number;
          digest: string;
        }
      > = {};
      for (let key of methodKeys) {
        let { rows, digest } = Provable.constraintSystem(obj[key]);
        cs[key] = {
          digest,
          rows,
        };
      }
      return cs;
    },
    async compile() {
      return {
        verificationKey: { data: '', hash: '' },
      };
    },
    name,
    digest: () => name,
  };
}
```

</file>

<file>

## path: /tests/vk-regression/tsconfig.json

url: https://github.com/o1-labs/o1js/blob/main/tests/vk-regression/tsconfig.json

```
{
  "extends": "../../tsconfig.json",
  "include": ["."],
  "exclude": [],
  "compilerOptions": {
    "rootDir": "../..",
    "baseUrl": "../..",
    "paths": {
      "o1js": ["."]
    }
  }
}
```

</file>

<file>

## path: /tests/vk-regression/vk-regression.json

url: https://github.com/o1-labs/o1js/blob/main/tests/vk-regression/vk-regression.json

```
{
  "Voting_": {
    "digest": "3f56ff09ceba13daf64b20cd48419395a04aa0007cac20e6e9c5f9106f251c3a",
    "methods": {
      "voterRegistration": {
        "rows": 1258,
        "digest": "5572b0d59feea6b199f3f45af7498d92"
      },
      "candidateRegistration": {
        "rows": 1258,
        "digest": "07c8451f1c1ea4e9653548d411d5728c"
      },
      "approveRegistrations": {
        "rows": 1146,
        "digest": "ec68c1d8ab22e779ccbd2659dd6b46cd"
      },
    "vote": {
      "rows": 1672,
      "digest": "fa5671190ca2cc46084cae922a62288e"
    },
    "countVotes": {
      "rows": 5796,
      "digest": "775f327a408b3f3d7bae4e3ff18aeb54"
    }
  },
  "verificationKey": {
    "data":
"AACd9tWcrEA7+0z2zM4uOSwj5GdeIBIROoVsS/yRuSRjKmnpZwY33yiryBLa9HQWpeZDSJI5y91gKJ9g5a
tItQApAhMdOuU5+NrHN3RCJtswX+WPvwaHJnihtSy2FcJPyghvBVTi2i7dtWIPQLVDIzC5ARu8f8H9JWjzjV
VYE/rQLruuq2qUsCrqdVsdRaw+6OjIFeAXS6mzvrVv5iYGslg5CV5mgLBg3xC408jZJ0pe8ua2mcIEDMGEd
SR/+VuhPQaqxZTJPBVhazVc1P9gRyS26SdOohL85UmEc4duqlJOOlXOFuwOT6dvoiUcdQtzuPp1pzA/LHu
eqm9yQG9mlT0Df8uY/A+rwM4l/ypTP/o0+5GCM9jJf9bl/z0DpGWheCJY+LZbIGeBUOpg0Gx1+KZsD9ivWJ0
vxNz8zKcAS1i3FgntjqyfY+62jfTR8PW1Y4wdaFan6jSxaaH6WYnvccAo2QHxEAFL91CfnZB1pwF8NAT395
N/rXr5XhMHFPoCkSHd2+5u+b62pkvFqqZZ9r24SMQOe9Bl2ZfMew2DyFLMPzwTowHw8onMEXcVKabFs9
zQVp66AMf/wlipirNztdguAEgTiVyzydYxNTKRpau/O5JaThaBCqePJzujSXLd5uIguUQkKMjVpfnHKOeoOtlM
Y8PYYFASPZjP4K1Y1XpE5DIc4d5ts+btlepIrTet7yJK5rlsFQfJGzaeTz9BN+g+C2ZK8B+2a2Qrz386FvB+elJ
AkJ2/Agn35oBHB2HobDkF6sRfrXOdH5l+QV7vR2v385RKRtfnmcJeUQcpq5/JTgVwagDJ/FarTN5jFsrBBRT
eW3yZ5/CfVNA7NNWxoKhjBaHVIhn/fLT5sFLYzYdCx/uTsusyZmE2d6iqnLS+j1IXNJX/zR0ZD3aGuoUc4Ma
FZQnN5om4dfpbloe4Roob3BuDhBHTKoYC+nVsyEvDRyiYLEOjJ45/bSwTCfwngYKtNmo3sVTvQ9mqBf0cL
dBCn8skp3S/gz324TFm8iJ+t8EWVKjlhM+1lrOQC7OfL98Sy0lD9j349LjxKcpiLTM7xxR/fSS4Yv9QXnEZxDig
YQO7N+8yMm6PfgqtNLa4gTlCOq1tWaRtaZtq24x+SyOo5P8EXWYuvsV/qMMPNmhoTq85lDI+iwlA1xDTYy
FHBdUe/zfoe5Znk7Ej3dQt+wVKtRgMqH5O4Df/c6DNekL1d6QYnjO0/3LMvY/f/y1+b7nPHI8+1Wqp5jZH8Us
uN63SSMdfBEe6x46AG/R+YS/wH78GKekabWu9QQnUJdjXyXiqF4qRebvfcmpQz91anvVz3ggBqCv4sYqCI
vP0ysDtMdi36zFErV+8SdUu+NsPDGvdPSCGdLuC25izxb21up2HORmlM5R7yuIW3rCiq8DeLD0OHjqOBZ
+IEv9zEkb5fHTJvxoxnZlArtZSBpD6iIDPVDymuK+BsOggZav3K+TytjeD2Gcld5NfyRISFWUIMkZNFQRL8A
QpET6RJnG1HSW0CaRfNeomtjCBWIr85wFCrp06j/D1J8B3EyhloZLJJ6ywxt41smXVugxA8LRTO+6lVBOB
F14jHQCCUl6u7uiWCe1z4/bC5wQXPwWSljp8NVU8Erp1U9ModNK7W63Pkh0efvgSD5d0nLzbfa0jTdxZ1Jk
fKsnvYk43Ed+vmXooHZhUeZAIX8ZCizhb1Gfvm02JFwxYXmiYAOp5wkGzweU2I5zo8r5yZFI1r4XibNQs7eA
fKGRv3gh8/EuLkX/bdettgPvNsI8ndpQ3kL/V8W2PQN4/hjC9AKCYBeXQG42bRncYZdLe++R2KA1ZdPDxQ
PF3sxUIKhzmRWqbozrtv310Maorwv6eZJjldlCJwICR9QgcDwDuNj+UFJnX3RWsdIWsUbI1T4wO0sE2sBiM
X/OqmiGJEAnBegioistlFyfRvm54h+duNOl/ol1Fva7NoXvsL/wThAWUly7bnc7/Al2bBQlUrmEX46UnKXzYntk
ZDee7Lx1u1BBkJAj/5BH1YZOPmMCh498rBUiHmc+4uQqebqNSHdOSgC39ESss4u7GNhWj3fi9XXta6UT9
wapEMGq0WTg2Kry6xNP2YZ5X8eaapRQc/KzYgz9XjQL6TKpqNuGEbRlmfYvIuoFbnOkZI7RYoGp3YheMs
1pQErwOxLzZa9W3Okwx16TSDwPLR0xMdAyogMrOdKN4JSMyNnmOaoVf6PkN+K9fz7RuHtvgjKpuz4vsK
5Z2wRneqPrnfu6PkgHcRQrd0SxqCbN23Z/yp8qOcN6XU49iCNEBjztT00tolQ9hCPMSE/eTZ+ioez7m3pJFV
ks3T5Rk/e+6MeowJWIOv20x6CPS9mhpr1JPwdNFrWdgs19VsobntCpF/rWxksdrYyk=",
    "hash":
"17404505535729023017641438102813310394161673484543048953955534000613641010 79"
  }
},
"Membership_": {
  "digest": "255745fb9365ff4f970b96ed630c01c9c8f63e21744f83fbe833396731d096e2",
  "methods": {
    "addEntry": {
      "rows": 1353,
      "digest": "fa32e32384aef8ce6a7d019142149d95"
    },
    "isMember": {
      "rows": 469,
      "digest": "16dae12385ed7e5aca9161030a426335"
    },
    "publish": {
      "rows": 694,
      "digest": "c7f77b05b8ec477338849af8dcb34a11"
    }
  },
  "verificationKey": {
    "data":
"AACwuS3vTWCwpRIX/QlJQqJcmPO9nPm4+sCfcrqiY1NUMiV9k6Pc8kFkMsbGLst78T8uAnYwc1Ql49kq0I
2GizwshS9xkBcfxRTAAMBHXhf8KDkK39AalVocKIrfWMV0MSShinj0bCxPCc10K0cya4Voy8fud4+hktDOuwj
aAstpEJSbKRHMIki77xHmJWlFUYdkgPg30MU4Ta3ev/h+mcMWmofyhLSQqUbaV6hM95n3Y0Wcn2LRNxJ
P8TRwHndIcylleqPsGMh3P+A+N9c32N4kl29nreMJJdcUrCXK90GLPAFOB9mHIjKk9+9o3eZc3cGQ+jppXo
N3zwO91DeT/GYvXqCZTAudLxIwuJU11UBThG5CKKABa9ulQ1bYGXj9Eydy0vPxfojDeFrnKMi9GKSjiSMz
mOLbIw7Dt+g9ggjsHM5rPrT7dY1VV4ZT9shjlcX3029xnk3Bjz4Q9PiK+A8o6f7L6aVB07I+QY2iDtwSQWuXY
Pohrk85I1UbPfY+giWqFXBtHaN45PMWCyBx0TKaozETCmv0kA5KGTzesYQCECPQ8F2DM+oXz8xly+z9/
Ypt/Zx9NvF7wute/1s6Q/QuAHHgQqvSF2AEzSEy6kDop6fnFtVTxzp0MgW0M9X0uVcRTRJTkcVZSz1JzihG
EjzkEZnZW6tVr6CEkmzXh/t3DSq2vXswFt90jphf6jgLtFJULrvKVg+YCMNM/04QLTGcMmjjzv4LciQ6IVXth7z
hVKxfL1/2peC0r/ZrP8k+Ox4LEBXWMCQE5kfK476bQgrLeKJfQ45PZfgB688DGwaYAxWbcxBV822/aAsA55
ijFY1Xf7S+DiytY4a/u0belIKMDUQqTOq9VwmbDv868zXscUwKpNVR3wy2En/q9M/HJJc4BZyuuQvlQSR59
m0gL4hKHf5Dci/YVvM6ACHmg+5SxCr1pUNKbyy2lsIa5Ma40ZmsTpT4/lQczmGENQSQXA9bFibT0Q+Vj88
5p9heLOCCXyAujC4DhAdYmT1MQ7v4IxcktsWwr3mRVBRM4iPa87OEKZOxq0wWPrGcTnmqV/ihFAcp38V
S2KUNwsiWjprCq1MFDCf1dT4c1U6/mdLP6AI/AJi7REoCfvJfwxSZYr2obhnskD1VjqelMdksHemFbsQDczN
hNcSg1TTD5ZsuG71wj9rSJPEisRCRRd733MLARwv6l24QrqQAp0ebGEbpXqv21bhlr6dYBscuIE2VU9SuGJ
2g6yuuKf4+lfJ2V5TkIxFvlgw5cxTXNQ010JYug38++ZDV+MibXPzg+cODE5wfZ3jon5wVNkAiG642DzXzNj6
7x80zBWLdt3UKnFZs9dpa1fYpTjlJg8T+dnJJiKf2IvmvF8xyi1HAwAFyhDL2dn/w/pDE2Kl9QdpZpQYDEBQg
CCkegsZszQ+2mjxU9pLXzz5GSoqz8jABW5Qo3abBAhvYKKaAs6NoRgeAD6SadFDbQmXaftE+Y1MVOtjn
aZDUBdwahWiJMlkfZpxW1aubEc/GSX8WzCZ8h9HeakcRc7kcN0CR8kmfER3eiZ2JMbt5cQl/afNjwGGAme
XzTaR34AgFjiw/RlZJkhYm9jyf18M8yP94QGBMxd6Y6wrNvOmJHzEnp8aitJsDlZklm8LKbjumlSbLcbBokpID
hFBBKfwP2qsQX7eHLCZ/3mztoFKoIiYXgrHWG8m2SzIJ/ljn6Rg7AxIsPjzZyEw1eXAOC7A1FCT/757ygMsn
k+rLlpDTBYLmhJtQdt61MQFDi5BuCmQ/PY9C/74/k4APl5htiNcCZty/1JElFwjuCQFjvAiMPUMyqp7/ALFapsT
ZqhSs1g6jd8uhuJoTNEqLDvKUUbs0kMvGy8BOG0YXNxmNccabGwBzxmijv6LF/Xinecl4aD8FCh6opY98TJ
nOHd3XSYL1DbLqmmc6CXEM+g5iDGnXr/CkI2Jy37OkF8X03jz4AH0Yj0+J63yH4IS+PrNpKZEXKh7PvXNa
LGGKsFcKEi63/xKPKH0G4RzvFKbkp+IWqtIYjMiwIJMwzmfS1NLLXqqpFiD364eFcXINR2rrDKcoTUp1JkVZ
VfXfKwaRUPWSGFYIYMtwPh2w8ZfubAmXZFpyzstORhFyg9rtVAAy0lcDhQwWVlhFFkR2qbdoy0EFLBrfKq
UIkd1N6vDQQYL1RGaTAv/ybregrJsFo+VP3ZatlR6LnKYWp1m7vPkGm3I6Pus/mvp1k10QGk8nhFuR31Djs
G3lzZ4gXSs1oSv0qbxD2S6g5+Y6cPbITEGX3uQjsunXnQ9PHd22Mk+fqbDakTiCJh6aFqqPNShiAXkGSuC
1oXJHX3zqnbn75dWO0UVhBNAbjYkSnQeyka1wnZb12sR+PlRMvWQVcd93t5L/FiE0ORo=",
    "hash":
"16610506589527352533348678289715227768202510979537802187565243095524972136674"
  }
},
"HelloWorld": {
  "digest": "20cadc6f44ecd546700e9fac15aa2740d4357d46ee03c2627c36be49b02e8227",
  "methods": {
    "update": {
      "rows": 2351,
      "digest": "f5b77fd12fee155fd3a40946dd453962"
    }
  },
  "verificationKey": {
      "data":
"AAAxHIvaXF+vRj2/+pyAfE6U29d1K5GmGbhiKR9lTC6LJ2o1ygGxXERl1oQh6DBxf/hDUD0HOeg/JajCp3V
6b5wytil2mfx8v2DB5RuNQ7VxJWkha0TSnJJsOl0FxhjldBbOY3tUZzZxHpPhHOKHz/ZAXRYFIsf2x+7boXC0
iPurETHN7j5IevHIgf2fSW8WgHZYn83hpVI33LBdN1pIbUc7oWAUQVmmgp04jRqTCYK1oNg+Y9DeIuT4EV
bp/yN7eS7Ay8ahic2sSAZvtn08MdRyk/jm2cLlJbeAAad6Xyz/H9l7JrkbVwDMMPxvHVHs27tNoJCzIlrRzB7pg
3ju9aQOu4h3thDr+WSgFQWKvcRPeL7f3TFjIr8WZ2457RgMcTwXwORKbqJCcyKVNOE+FlNwVkOKER+W
IpC0OlgGuayPFwQQkbb91jaRlJvahfwkbF2+AJmDnavmNpop9T+/Xak1adXIrsRPeOjC+qIKxIbGimoMOoYz
YlevKA80LnJ7HC0IxR+yNLvoSYxDDPNRD+OCCxk5lM2h8IDUiCNWH4FZNJ+doiigKjyZlu/xZ7jHcX7qibu/3
2KFTX85DPSkQM8dADbWQUmeiyX6c8BmLNrtM9m7dAj8BktFxGV9DpdhbakQltUbxbGrb3EcZ+43YFE/y
Wa3/WAQL81kbrXD0yjFthEKR89XcqLS/NP7lwCEej/L8q8R7sKGMCXmgFYluWH4JBSPDgvMxScfjFS33oB
Nb7po8cLnAORzohXoYTSgztklD0mKn6EegLbkLtwwr9ObsLz3m7fp/3wkNWFRkY5xzSZN1VybbQbmpyQN
Cpxd/kdDsvlszqlowkyC8HnKbhnvE0Mrz3ZIk4vSs/UGBSXAoESFCFCPcTq11TCOhE5rumMJErv5LusDHJg
rBtQUMibLU9A1YbF7SPDAR2QZd0yx3wbCC54QZ2t+mZ4s6RQndfRpndXoIZJgari62jHRccBnGpRmURH
G20jukwW6RYDDED7OlvEzEhFlsXyViehlSn4Evb44z+VGilheD0D6v1paoVTv2A4m5ZVGEQOeoCQELABd
oFrIZRrd4+glnXPz8Gy4nOI/rmGgnPa9fSK0N1zMKEexIlapLarEGI7ZVvg5jAqXDlXxVS3HRo/skxgt2LYm8w
LIKLHX0ClznArLVLXkSX18cSoSsVMG3QCSsmH1Oh8xOGUbSHzawovjubcH7qWjIZoghZJ16QB1c0ryiAfH
B48OHhs2p/JZWz8Dp7kfcPkeg2Of2NbupJlNVMLIH4IGWaPAscBRkZ+F4oLqOhJ5as7fAzzU8PQdeZi0Ygs
sGDJVmNEHP61I16KZNcxQqR0EUVwhyMmYmpVjvtfhHi/6I3mfPS+FDxTuf4yaqVF0xg2V3ep/WYnnKPJIe
gxoTFY8pChjyow3PMfhAP5HOnXjHQ2Va9BFo4mfEQXvRzPmIRRVmlVsP8zA+xuHylyiww/Lercce7cq0YA5
PtYS3ge9IDYwXckBUXb5ikD3aIrrv5mvMu6itB7ix2f8lbiF9Fkmc4Bk2ycIWXJDCuBN+2sTFqzUeoT6xY8XWa
OcnDvqOgSm/CCSv38umiOE2jEpsKYxhRc6W70UJkrzd3hr2DiSF1I2B+krpUVK1GeOdCLC5sl7YPzk+pF81
83uI9wse6UTlqIiroKqsggzLBy/IjAfxS0BxFy5zywXqp+NogFkoTEJmR5MaqOkPfap+OsD1IGScY6+X4WW/H
qCWrmA3ZTqDGngQMTGXLCtl6IS/cQpihS1NRbNqOtKTaCB9COQu0oz6RivBlywuaj3MKUdmbQ2gVDj+S
GQItCNaXawyPSBjB9VT+68SoJVySQsYPCuEZCb0V/40n/a7RAbyrnNjP+2HwD7p27Pl1RSzqq35xiPdnycD
1UeEPLpx/ON65mYCkn+KLQZmkqPio+vA2KmJngWTx+ol4rVFimGm76VT0xCFDsu2K0YX0yoLNH4u2Xfm
T9NR8gGfkVRCnnNjlbgHQmEwC75+GmEJ5DjD3d+s6IXTQ60MHvxbTHHInfmPbgKn2SAI0uVoewKC9GyK
6dSaboLw3C48jl0E2kyc+7umhCk3kEeWmt//GSjRNhoq+B+mynXiOtgFs/Am2v1TBjSb+6tcijsf5tFJmeGxlCjJ
nTdNWBkSHpMoo6OFkkpA6/FBAUHLSM7Yv8oYyd0GtwF5cCwQ6aRTbl9oG/mUn5Q92OnDMQcUjpgEho
0Dcp2OqZyyxqQSPrbIIZZQrS2HkxBgjcfcSTuSHo7ONqlRjLUpO5yS95VLGXBLLHuCiIMGT+DW6DoJRtRIS
+JieVWBoX0YsWgYInXrVlWUv6gDng5AyVFkUIFwZk7/3mVAgvXO83ArVKA4S747jT60w5bgV4Jy55sIDM="
,
      "hash":
"28560680247074990771744165492810964987846406526367865642032954725768850073454"
    }
  },
  "TokenContract": {
    "digest": "346c5ce0416c2479d962f0868825b4bcbf68f5beac5e7a93632013a6c57d1be8",
    "methods": {
      "init": {
        "rows": 655,
        "digest": "3941ac88f0b92eec098dfcf46faa4e60"
      },
      "init2": {
        "rows": 652,
        "digest": "1ebb84a10bafd30accfd3e8046d2e20d"
      },
      "deployZkapp": {
        "rows": 702,
        "digest": "e5ac2667a79f44f1e7a65b12d8ac006c"
      },
      "approveUpdate": {
        "rows": 1928,
      "digest": "f8bd1807567dc405f841283227dfb158"
    },
    "approveAny": {
      "rows": 1928,
      "digest": "240aada76b79de1ca67ecbe455621378"
    },
    "approveUpdateAndSend": {
      "rows": 2321,
      "digest": "b1cff49cdc3cc751f802b4b5aee53383"
    },
    "transferToAddress": {
      "rows": 1044,
      "digest": "212879ca2441ccc20f5e58940833cf35"
    },
    "transferToUpdate": {
      "rows": 2326,
      "digest": "a7241cbc2946a3c468e600003a5d9a16"
    },
    "getBalance": {
      "rows": 686,
      "digest": "44a90b65d1d7ee553811759b115d12cc"
    }
  },
  "verificationKey": {
    "data":
"AAAVRdJJF0DehjdPSA0kYGZTkzSfoEaHqDprP5lbtp+BLeGqblAzBabKYB+hRBo7ijFWFnIHV4LwvOlCtrA
hNtk/Ae0EY5Tlufvf2snnstKNDXVgcRc/zNAaS5iW43PYqQnEYsaesXs/y5DeeEaFxwdyujsHSK/UaltNLsCc34
RKG71O/TGRVVX/eYb8saPPV9W5YjPHLQdhqcHRU6Qq7hMEI1ejTXMokQcurz7jtYU/P56OYekAREejgrE
V38U82BbgJigOmh5NhgGTBSAhJ35c9XCsJldUMd5xZiua9cWxGOHm0r7TkcCrV9CEPm5sT7sP7IYQ5dnS
dPoi/sy7moUPRitxw7iGvewRVXro6rIemmbxNSzKXWprnl6ewrB2HTppMUEZRp7zYkFIaNDHpvdw4dvjX6K/i
527/jwX0JL4BideRc+z3FNhj1VBSHhhvMzwFW6aUwSmWC4UCuwDBokkkBtUE0YYH8kwFnMoWWAlDzH
ekrxaVmxWRS0lvkr8IDlsR5kyq8SMXFLgKJjoFr6HZWE4tkO/abEgrsK1A3c9F5r/G2yUdMQZu8JMwxUY5qw
7D09IPsUQ63c5/CJpea8PAHbUlzRl2KhAhm58JzY0th81wwK0uXhv2e0aXMoEpM0YViAu+c/32zmBe6xl97u
BNmNWwlWOLEpHakq46OzONidU3betWNXGJbS4dC4hTNfWM956bK+fwkIlwhM3BC+wOai+M0+y9/y/RSI
8qJkSU3MqOF9+nrifKRyNQ3KILqIyR7LjE0/Z/4NzH7eF3uZTBlqfLdf8WhXdwvOPoP1dCx1shF6g4Hh9V4m
yikRZBtkix1cO5FLUNLNAFw+glg1PB1eA+4ATFuFcfMjxDpDjxqCFCyuQ5TaLuNfYMA7fiO0vB6yqtWgSmC
OID/MQqAhHYRMq4PXk3TUQSle8XBZ67T0+gENjIJleTRgZFG6PgIEwHXcsKIvfFAPklTlnY+5sNVw8yBisV
aFgw36DrHWNavWvsZM5HwD0h1Wk0hkavjEIz9nTxQU+nsZsR+70ALZ69HljR0fUjNU7qpVmpYBlRiFxA/B
Wf8qie2wfhSfy6Q1v5Ee4+3vN/mYuS3uF47LkM1dRTanQ73mLIz80yky+lCNkLWHmZtyWjtMsDFNgupc+yc
+FvFNjJM/ea6u3PROtSyU3rAlmchkKvxO4qfrd0iqav/WbabGDMJhbugO4TNu1/i5omH8pbsjGGHQXk1UYPo
P1SnMVPZ9RXPoWHJn/kePU9QqGxETHF4T7b2Ov7CcZDLuz147VCknmGiziHzbmYJleu4tzSlFsxHPkp2d
9JiDUbO7X66Dh/+84gc5KWpMnEIAF9gITi3cXUglZTjWaASaXcpgHXXGZHZJcrG2VfPNjgTKJ1+CbvyXlvuh
vX+0E2oaPB+BoP0i2iTXQHPNhOY/Gg2h6uKvE5fSSiYC7Rws2TGF1aEM54wX3Ti1qA1cAiNG5y8yk1YMG
Ck3TPqs9MRp0qjgjJbbvFlbgPkkqz5o6c7g8gfhIa4VEJyyI2joqJeIc7vMZFWhquSFHNs0TZKvKLiSAsyNDrp
WZb/1PHxziswKvisk296AJi7hmlM1pKx6S4LlbT2OKLXbgq5HUKfe8QhxG4aOsPSSiVGwvnCrIPdSxLq77M2
7UWXnXHC8mmJmOsGUFj+bdX/u6AgrBhw/w74dDbuNEpC80PbJTugIF/TeDryYsFWCrBnF/WPstgzy3zDD
TZ3DXHVYVxOEvErIynlQEY9Cv9QSxRI3dA+hLtob/L78ZeJSU4Al+Qv0QGZTOxQORosVshOP2eFQ1VMK
GWOpCVvyi8QE4fa+gOgYT0JRm4rkQBZ5WDlYGkamD3euC92Kd7Z39G89h/AqeFACahkAW1a78SzLW6
9mZ+CDLfKp/xQsi2TWgJqGh7QNOEtMnn/2owLzLWd071mvUtT0484Eqx6hUqLJMH70p8oUjQIMsh0mvp1
BWSU8XC6z+UZIpVm2CERrV8BMLmTLOgTNJlEIJQR7zzpJCDFNNOI+Y2ZtdcuU8XHgcsQhQ3PgCACFA
WN3rO+goXoTWdYR/LcqszKzPnMArmPIHWkRM6Mkm13OsHXCVudUbqQjC/pNQZH1VW+RMXnre1vQV
b3fnCy5h28Dce3Q2WzjBSZFhe3iADZpo7gWHM/sqe+Mbnbn8A+RRWVNbtjss9376jN73zV4xPH3un3VjTxr
```

```
zCluqR8MbH8t7mhPBqV5CslmSIbDNruVXtwCf4VS1nssw63PfLzeOSvzhTTsg82rna/+TKl1RIwhD8VFnCDq
/Rk8fdy/+K5qP6GcSTbh6J8ERx4jOOukL9TUCpJkhvo/3ED8GOewmWAwzL8avXuf9AFvhwH3ENp5v4IIGBl
juDJ77vckGmTI=",
      "hash":
"13796172868423455932596117465273580383420853883879480382066094121613342871544"
    }
  },
  "Dex": {
    "digest": "14f902411526156cdf7de9a822a3f6467f7608a135504038993cbc8efeaf720a",
    "methods": {
      "supplyLiquidityBase": {
        "rows": 3749,
        "digest": "08830f49d9e8a4bf683db63c1c19bd28"
      },
      "swapX": {
        "rows": 1986,
        "digest": "e1c79fee9c8f94815daa5d6fee7c5181"
      },
      "swapY": {
        "rows": 1986,
        "digest": "4cf07c1491e7fc167edcf3a26d636f3d"
      },
      "burnLiquidity": {
        "rows": 718,
        "digest": "99fb50066d2aa2f3f7afe801009cb503"
      },
      "transfer": {
        "rows": 1044,
        "digest": "7c188ff9cf8e7db108e2d24f8e097622"
      }
    },
    "verificationKey": {
      "data":
"AADgDFCYyznG8hH/Z695+WW86B544SmJFzz5ObrizTJ4KMqy+pfsOR2Mt2yGViXSJPpAR76RNHNga83
UB8/9OPQIB+uHOnxXH7vN8sUeDQi50gWdXzRlzSS1jsT9t+XsQwHNWgMQp04pKmF+0clYz1zwOO95Bw
HGcQ/olrSYW4tbJN6KW0hN2eESQfUJcwfB6uUzwvGtkFs+aiUykn7KUgUgXQkKgdHHdyFioNHNPmkpiAre
/Ts8BKwwvf5hCa1MtBF6ax6ymlATB4YBL0ETiEPTE/Qk1zGWUSL2UB6aY45/LlfTLCKlyLq7cR3HOucFfBnc
VfzI7D8j5n4wVqY+vAI4cf+Yv7iVRLbeFcycXtsuPQntgBzKa/mcqcWuVM7p2SYRrtKdX8EKvOO6NhfLx4x0at
Ai8pKf+vZR76LSP4iOA8hwXvk6MNvPt1fxCS96ZAKuAzZnAcK+MH1OcKeLj+EHtZmf40WRb3AEG5TWRK
uD6DT5noDclZsE8ROZKUSOKAUGIBvt7MpzOWPPchmnromWEevmXo3GoPUZCKnWX6ZLAtJwAszLUgi
VS8rx3JnLXuXrtcVFto5FFQhwSHZyzuYZAOLg+O5JsHz6EFkkWxwdCSmy1K1LFaS6A78wbTtc9uIslLAntK
xTApVE2lxPzH+TwHBFMkSweXxdP3dGxtecxxpbbLKvz9Clh3WpX0ia/8PSErjEfdpClkDrgo8DG2MpEgFaBc
gfyFNTEhXLnxCiGlwjJ+DdBAfnonMPIkkY6p0SJ5M/KjfmCc2/EsnV7Mhax350ZtrXdzh/HWIWzEZKKxcbERF
bRtf+fkMOOLNpNov1FEFvKOU612vDOIbrVHeBN9mwuepUrJctcfgLc0Mi3Sxs3+NA0I74qm5ktjmplDwgUtK
zIs3IrVFv6b1pg/J32HmwNzJZw2fYzpFE1LDjBSK/SX3axwMy5yEd8+jl4uAdQZpa9UQQIHu1Y1ZMgJSDDic
Xz6D1bZMA1Q2/lU+8AYbldgQVmlLq/lzr63krX+AM84dcwR1Ur7O0YSVR5TXXJhMigCPYsF0/fmLijOWNAg
a8rtMJvF0oZ02aoQv4KpGu9yq72CsoXSpWqu/6C+GSP52zL9QV0VkohE1njGsSrC/EMtWuNxk6avge+WIx
nbAbrFVGoWKdAN3uuZBKQW6ehhi1watI+S5lkpbpTnrK3R/59l19FcR35ItoigIxtMfkv3rdlCOeBVI93oVl5esiH
8AvYGHhulWIvrNfKol3Viir41zv4qMBOcQg8+ygqjwqREU5+qiYeJlQ2AtT0/PVeZWg4mHC39uz1Lld3N2hyyx
Ro+Z0nC/8220uuf9gAnQ+JFixgyYW0NowUtuFj+uYAV9Dh/Zpe4LyAOkU0kBW4CEuOxNr+gz+9h0BoPfBHI
MuuQAUc5L8uMunJC7uBKZiL+/tT1ZGfyIuqU47fEP9Hghxmip8v7gpf+4wB0MVUUwav9QRe9g88ER1HcJP
qYb4EIOc2kbYSX75bT0mAFqR8lwZrj6lbQtNS0QQboG5fzoyYGi8YnSXhC2T5fFDpGJ319GHUsna58o5wk8
```

```
LMwKWNTxq+FN6XiRgu0BFOrtG6MtT1OxYE9Dti6WatGDsWv+KMLDHjxUK1bhiSRnvkWYNcnuDJ0Ry+P
RGHNUijVU0SbchntC2JHdhwKbwIofwKHE8HhvlK8FgQ1VOLDioA26UFzr23LpCTqwSJ7/sAqttNGcPR8MS
eeR9TQvXNYQPKrA7Gh720X+7LD6BuHdy4vkcr9EKBU0ccUJ2ABBiyPdji+AgEbUCL/wrp6/GX8pui5YJGW
x3XmIFj/RnYS2Je5FZ7w74JclD3XhLUo5Dhpq5RznHplpLB9mNdZdm5269US/XCgC/ZKyUxW3+0ajdBY1cL
zF6qglitaYTp3MVUENVOkACM2RyKw6jIK2Leq3qLp6AUz21VXj4WznZcdI8MXqT9v8HxjXbAI9dtbhLRZRpJ
mu/129vrVmwSTHvsVoA7vXyYh/iO3ZMcy+D1x+HZU6Q/oDYCicqOPHxpSc9QGehmNyeGzI//524Gz3Rudk
U7s6MPdLWqZrieRTnWsTIrCDieu4VaIfP8BFz7asYUv0t9jMWpv3yjbY7c5h8N/m7IUXwTQCzFpjPV7HC72B
jVwPaYqh5/oAQsSNcv5I3c2GsCGj5C4hFFoT7eWfVtu/6ibQl0COhRDsegnOBtZ7NGfybI8IIO/4yrgel92bypb3
eSxeMvdE5wzURluGDkBVVIACD8C5W1MzqrejUiiTfc3mkLhQ0xKRRhT0qqkmYWlbGN5hmMOA9YaYx8O
FTgMys1WbzdidWgEkyvvdkWctGlges6eg/lJE61tJ8wGxvJfKtpyDW/2MRvsnO1+2EXIQ2eV3hkxg=",
     "hash":
"6361961148584909856756402479432671413765163664396823312454174383287651676472"
   }
  },
  "Group Primitive": {
   "digest": "Group Primitive",
   "methods": {
    "add": {
     "rows": 30,
     "digest": "8179f9497cc9b6624912033324c27b6d"
    },
    "sub": {
     "rows": 30,
     "digest": "ddb709883792aa08b3bdfb69206a9f69"
    },
    "scale": {
     "rows": 113,
     "digest": "b912611500f01c57177285f538438abc"
    },
    "equals": {
     "rows": 37,
     "digest": "59cd8f24e1e0f3ba721f9c5380801335"
    },
    "assertions": {
     "rows": 19,
     "digest": "7d87f453433117a306b19e50a5061443"
    }
   },
   "verificationKey": {
    "data": "",
    "hash": ""
   }
  },
  "Bitwise Primitive": {
   "digest": "Bitwise Primitive",
   "methods": {
    "rot": {
     "rows": 13,
     "digest": "2c0dadbba96fd7ddb9adb7d643425ce3"
    },
    "xor": {
     "rows": 15,
        "digest": "b3595a9cc9562d4f4a3a397b6de44971"
      },
      "notUnchecked": {
        "rows": 2,
        "digest": "fa18d403c061ef2be221baeae18ca19d"
      },
      "notChecked": {
        "rows": 17,
        "digest": "5e01b2cad70489c7bec1546b84ac868d"
      },
      "leftShift": {
        "rows": 7,
        "digest": "66de39ad3dd5807f760341ec85a6cc41"
      },
      "rightShift": {
        "rows": 7,
        "digest": "a32264f2d4c3092f30d600fa9506385b"
      },
      "and": {
        "rows": 19,
        "digest": "647e6fd1852873d1c326ba1cd269cff2"
      }
    },
    "verificationKey": {
      "data": "",
      "hash": ""
    }
  }
}
```

</file>

<file>

## path: /tests/vk-regression/vk-regression.ts

url: https://github.com/o1-labs/o1js/blob/main/tests/vk-regression/vk-regression.ts

```
import fs from 'fs';
import { Voting_ } from '../../src/examples/zkapps/voting/voting.js';
import { Membership_ } from '../../src/examples/zkapps/voting/membership.js';
import { HelloWorld } from '../../src/examples/zkapps/hello_world/hello_world.js';
import { TokenContract, createDex } from '../../src/examples/zkapps/dex/dex.js';
import { GroupCS, BitwiseCS } from './plain-constraint-system.js';

// toggle this for quick iteration when debugging vk regressions
const skipVerificationKeys = false;

// usage ./run ./tests/regression/vk-regression.ts --bundle --dump ./tests/vk-regression/vk-regression.json
let dump = process.argv[4] === '--dump';
let jsonPath = process.argv[dump ? 5 : 4];
```

```
type MinimumConstraintSystem = {
  analyzeMethods(): Record<
    string,
    {
      rows: number;
      digest: string;
    }
  >;
  compile(): Promise<{
    verificationKey: {
      hash: { toString(): string };
      data: string;
    };
  }>;
  digest(): string;
  name: string;
};

const ConstraintSystems: MinimumConstraintSystem[] = [
  Voting_,
  Membership_,
  HelloWorld,
  TokenContract,
  createDex().Dex,
  GroupCS,
  BitwiseCS,
];

let filePath = jsonPath ? jsonPath : './tests/vk-regression/vk-regression.json';
let RegressionJson: {
  [contractName: string]: {
    digest: string;
    methods: Record<string, { rows: number; digest: string }>;
    verificationKey: {
      hash: string;
      data: string;
    };
  };
};

try {
  RegressionJson = JSON.parse(fs.readFileSync(filePath).toString());
} catch (error) {
  if (!dump) {
    throw Error(
      &nbsp;The requested file ${filePath} does not yet exist, try dumping the verification keys first. npm run
dump-vks&nbsp;
    );
  }
}

async function checkVk(contracts: typeof ConstraintSystems) {
```

```
let errorStack = '';

for await (const c of contracts) {
  let ref = RegressionJson[c.name];
  if (!ref)
    throw Error(
      &nbsp;Verification key for contract ${c.name} was not found, try dumping it first.&nbsp;
    );
  let vk = ref.verificationKey;

  let {
    verificationKey: { data, hash },
  } = await c.compile();

  let methodData = c.analyzeMethods();

  for (const methodKey in methodData) {
    let actualMethod = methodData[methodKey];
    let expectedMethod = ref.methods[methodKey];

    if (actualMethod.digest !== expectedMethod.digest) {
      errorStack += &nbsp;\n\nMethod digest mismatch for ${c.name}.${methodKey}()
Actual
  ${JSON.stringify(
    {
      digest: actualMethod.digest,
      rows: actualMethod.rows,
    },
    undefined,
    2
  )}
\n
Expected
  ${JSON.stringify(
    {
      digest: expectedMethod.digest,
      rows: expectedMethod.rows,
    },
    undefined,
    2
  )}&nbsp;;
    }
  }

  if (data !== vk.data || hash.toString() !== vk.hash) {
    errorStack += &nbsp;\n\nRegression test for contract ${
      c.name
    } failed, because of a verification key mismatch.
Contract has
 ${JSON.stringify(
  {
    data,
    hash,
   },
   undefined,
   2
  )}
\n
but expected was
  ${JSON.stringify(ref.verificationKey, undefined, 2)}&nbsp;;
   }
  }

  if (errorStack) {
   throw Error(errorStack);
  }
}

async function dumpVk(contracts: typeof ConstraintSystems) {
  let newEntries: typeof RegressionJson = {};
  for await (const c of contracts) {
   let data = c.analyzeMethods();
   let digest = c.digest();
   let verificationKey:
    | { data: string; hash: { toString(): string } }
    | undefined;
   if (!skipVerificationKeys) ({ verificationKey } = await c.compile());
   newEntries[c.name] = {
    digest,
    methods: Object.fromEntries(
     Object.entries(data).map(([key, { rows, digest }]) => [
      key,
      { rows, digest },
     ])
    ),
    verificationKey: {
     data: verificationKey?.data ?? '',
     hash: verificationKey?.hash.toString() ?? '0',
    },
   };
  }

  fs.writeFileSync(filePath, JSON.stringify(newEntries, undefined, 2));
}

if (dump) await dumpVk(ConstraintSystems);
else await checkVk(ConstraintSystems);
```

</file>

<file>

## path: /tsconfig.examples.json

url: https://github.com/o1-labs/o1js/blob/main/tsconfig.examples.json

```
{
  "extends": "./tsconfig.json",
  "include": ["./src/examples/**/*.ts"],
  "exclude": [],
  "compilerOptions": {
    "outDir": "dist/",
    "importHelpers": false
  }
}
```

</file>

<file>

## path: /tsconfig.json

url: https://github.com/o1-labs/o1js/blob/main/tsconfig.json

```
{
  "include": ["./src/**/*.ts"],
  "exclude": ["./src/**/*.bc.js", "./src/build", "./src/examples"],
  "compilerOptions": {
    "rootDir": "./src",
    "outDir": "dist",
    "baseUrl": ".", // affects where output files end up
    "target": "es2021", // goal: ship *the most modern syntax* that is supported by *all* browsers that support
our Wasm
    "module": "nodenext", // allow top-level await
    "moduleResolution": "nodenext", // comply with node + "type": "module"
    "esModuleInterop": true, // to silence jest

    "experimentalDecorators": true, // needed for decorators
    "emitDecoratorMetadata": true, // needed for decorators
    "useDefineForClassFields": false, // ensure correct behaviour of class fields with decorators

    "strict": true, // for the full TypeScript experience
    "strictPropertyInitialization": false, // to enable generic constructors, e.g. on CircuitValue
    "importHelpers": true, // reduces size
    "declaration": true, // declaration files are how library consumers get our types
    "noEmitOnError": false, // avoid accidentally shipping with type errors
    "allowJs": true, // to use JSDoc in some places where TS would be too cumbersome
    "sourceMap": true
  }
}
```

</file>

<file>

## path: /tsconfig.mina-signer-web.json

url: https://github.com/o1-labs/o1js/blob/main/tsconfig.mina-signer-web.json

```
{
  "extends": "./tsconfig.mina-signer.json",
  "include": ["./src/mina-signer/MinaSigner.ts", "./src/**/*.web.ts"],
  "exclude": ["./src/examples"],
  "compilerOptions": {
    "outDir": "src/mina-signer/dist/tmp"
  }
}
```

</file>

<file>

## path: /tsconfig.mina-signer.json

url: https://github.com/o1-labs/o1js/blob/main/tsconfig.mina-signer.json

```
{
  "include": ["./src/mina-signer/MinaSigner.ts"],
  "exclude": ["./src/**/*.unit-test.ts"],
  "compilerOptions": {
    "rootDir": "./src",
    "outDir": "src/mina-signer/dist/node",
    "baseUrl": ".", // affects where output files end up
    "target": "es2020", // goal: ship *the most modern syntax* that is supported by *all* browsers that support
our Wasm
    "module": "es2022", // allow top-level await
    "moduleResolution": "nodenext", // comply with node + "type": "module"
    "esModuleInterop": true, // to silence jest

    "strict": true, // for the full TypeScript experience
    "importHelpers": true, // reduces size
    "declaration": true, // declaration files are how library consumers get our types
    "noEmitOnError": false, // avoid accidentally shipping with type errors
    "allowJs": true, // to use JSDoc in some places where TS would be too cumbersome
    "sourceMap": true
  }
}
```

</file>

<file>

## path: /tsconfig.node.json

url: https://github.com/o1-labs/o1js/blob/main/tsconfig.node.json

```
{
  "extends": "./tsconfig.json",
  "include": [
    "./src/index.ts",
    "./src/snarky.js",
    "./src/bindings/js/wrapper.js",
    "./src/mina-signer/src",
    "./src/mina-signer/MinaSigner.ts",
    "./src/js_crypto",
    "./src/provable"
  ],
  "compilerOptions": {
    "outDir": "dist/node"
  }
}
```

</file>

<file>

## path: /tsconfig.test.json

url: https://github.com/o1-labs/o1js/blob/main/tsconfig.test.json

```
{
  "extends": "./tsconfig.json",
  "include": [
    "./src/**/*.unit-test.ts",
    "./src/snarky.js",
    "./src/bindings/js/wrapper.js"
  ],
  "compilerOptions": {
    "outDir": "dist/node"
  }
}
```

</file>

<file>

## path: /tsconfig.web.json

url: https://github.com/o1-labs/o1js/blob/main/tsconfig.web.json

```
{
  "extends": "./tsconfig.json",
  "include": ["./src/index.ts", "./src/snarky.js", "./src/**/*.web.ts"],
  "compilerOptions": {
    "outDir": "dist/web"
  }
}
```

</file>

<file>

## path: /typedoc.json

url: https://github.com/o1-labs/o1js/blob/main/typedoc.json

```
{
  "name": "o1js",
  "plugin": ["typedoc-plugin-markdown", "typedoc-plugin-merge-modules"],
  "out": "o1js-reference",
  "cleanOutputDir": true,
  "githubPages": false,
  "exclude": ["dist/**/*", "src/mina-signer/**/*", "src/examples/**/*"],
  "entryPoints": [
    "src/index.ts",
    "src/snarky.d.ts",
    "src/lib/field.ts",
    "src/lib/group.ts",
    "src/lib/bool.ts"
  ],
  "entryPointStrategy": "resolve"
}
```

</file>

</repository>