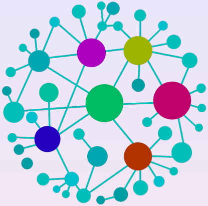


Blockchain Course

Hotmoka

fausto.spoto@univr.it

https://github.com/spoto/blockchain-course

Hotmoka

`https://github.com/Hotmoka/hotmoka`

An open-source implementation of a network of nodes:

- nodes of a blockchain
- IoT devices
- computers in the cloud

Requests are OO-based

- install code in the node
- create an object
- call a method of an object
- methods are implemented in Takamaka (subset of Java)

Documentation

There is an online tutorial on Hotmoka and Takamaka in the `README.md` of the main repository of Hotmoka:
`https://github.com/Hotmoka/hotmoka`

Its examples of Takamaka projects are available here:
`https://github.com/Hotmoka/hotmoka_tutorial`

The API of a Hotmoka node

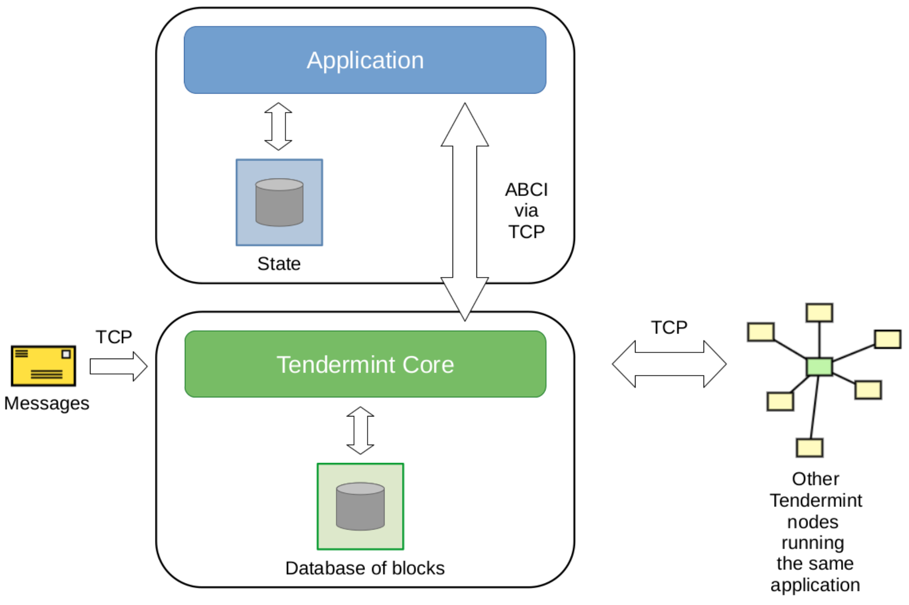
- `[add|post]JarStore(request):TransactionReference`
- `[add|post]ConstructorCall(request):StorageReference`
- `[add|post|run]InstanceMethodCall(request):StorageValue`
- `[add|post|run]StaticMethodCall(request):StorageValue`
- `subscribeToEvents(key):Subscription`
- `getState(StorageReference):State`

`add` calls are synchronous (wait for the result)

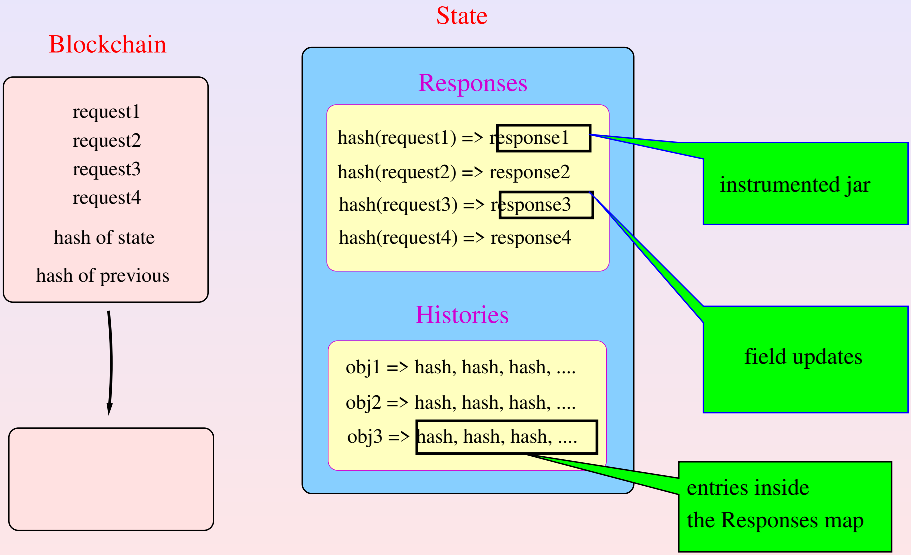
`post` calls are asynchronous (yield a future)

`run` calls are synchronous and only for read-only methods

Nodes can be Tendermint applications



An OO state (hash is sha256)


Blockchain
request1
request2
request3
request4
hash of state
hash of previous
State
Responses
hash(request1) => response1
hash(request2) => response2
hash(request3) => response3
hash(request4) => response4
Histories
obj1 => hash, hash, hash, ...
obj2 => hash, hash, hash, ...
obj3 => hash, hash, hash, ...
instrumented jar
field updates
entries inside
the Responses map

The API of the state

1. get jar at `hash`: access the Responses map and find the jar
2. get object at `obj`: access the Histories at `obj`: for each `hash` there, access the Responses map at `hash`, project the updates on `obj` and reconstruct the state of `obj`
3. put request/response in state: expand Responses with hash(request)⇒response

   if the response contains updates, add hash(request) to the histories of the updated objects
4. `h=get_hash()`: compute the hash of the hash of the Merkle-Patricia trie for Responses and of that for Histories
5. ~~`checkout(h)`~~ ⇒ unused data from points above are garbage-collected

Start experimenting with Hotmoka[*]

Ensure first to have Java version ≥ 11 installed

```
$ cd ~/Opt
$ mkdir moka
$ cd moka
$ wget https://github.com/Hotmoka/hotmoka/
    releases/download/1.0.0/moka_1.0.0_linux.tar.gz
$ tar zxf moka_1.0.0_linux.tar.gz
$ export PATH=$PATH:~/Opt/moka
```

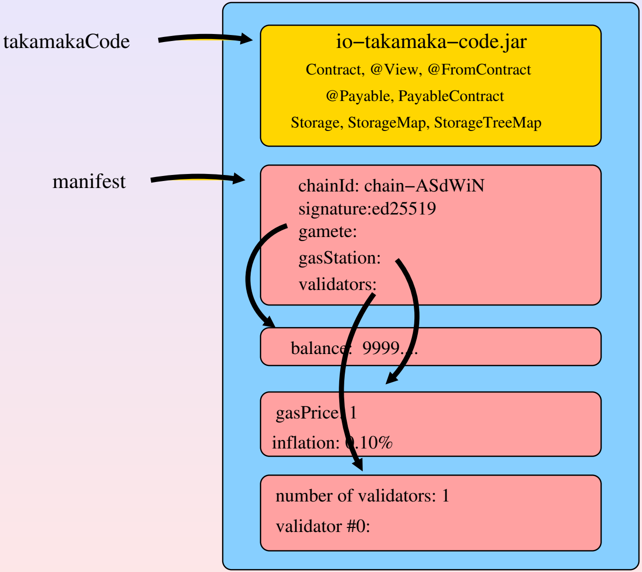
You might want to add the `export` at the end of your `/.bashrc` as well

[*] The network server used in the experiment is joint work with Dinu Berinde

moka info --url panarea.hotmoka.io

```
takamakaCode: 02dfd29348abaa44f720525179fa170f26063c973fd40c3ff368a9402551882c
manifest: 42a8a11aee0405aee5775514b3b0456c7740bbb015b4b87df4776e6e4add7668#0
  chainId: chain-ASdWiN
  maxErrorLength: 300
  maxDependencies: 20
  maxCumulativeSizeOfDependencies: 10000000
  allowsSelfCharged: false
  allowsUnsignedFaucet: true
  skipsVerification: false
  signature: ed25519
  gamete: 4f7d7ca1fbea152d8f323c21e1abcfa1d979c7c4ea667d8457381a26b08a2d71#0
    balance: 999999999999999999999999999999992180
    redBalance: 0
    maxFaucet: 1000000
    maxRedFaucet: 0
  gasStation: 42a8a11aee0405aee5775514b3b0456c7740bbb015b4b87df4776e6e4add7668#10
    gasPrice: 1
    maxGasPerTransaction: 1000000000
    ignoresGasPrice: false
    targetGasAtReward: 1000000
    inflation: 10000 (ie. 0.10%)
    oblivion: 250000 (ie. 25.00%)
  validators: 42a8a11aee0405aee5775514b3b0456c7740bbb015b4b87df4776e6e4add7668#2
    number of validators: 1
    validator #0: 42a8a11aee0405aee5775514b3b0456c7740bbb015b4b87df4776e6e4add7668#1
      id: C220489CDBAE0FAFF8F8286A9C541FD55BA2CE7C
      power: 1
    ticketForNewPoll: 100
    number of polls: 0
  versions: 42a8a11aee0405aee5775514b3b0456c7740bbb015b4b87df4776e6e4add7668#f
    verificationVersion: 0
```

The minimal content of a Hotmoka node's state



takamakaCode

io−takamaka−code.jar

Contract, @View, @FromContract

@Payable, PayableContract

Storage, StorageMap, StorageTreeMap

manifest

chainId: chain−ASdWiN

signature:ed25519

gamete:

gasStation:

validators:

balance: 9999...

gasPrice: 1

inflation: 0.10%

number of validators: 1

validator #0:

The state contains actual Java objects

$$\texttt{manifest: }\underbrace{\texttt{42a8a11aee0405aee5775514b3b0456c7740bbb015b4b87df4776e6e4add7668\#0}}_{\text{machine-independent memory address of an object}}$$

```
moka state 42a8a11aee0405aee5775514b3b0456c7740bbb015b4b87df4776e6e4add7668#0 --url panarea.hotmoka.io
```

```
class io.takamaka.code.governance.Manifest (from jar installed at 02dfd29348abaa44f7205251...)
  allowsSelfCharged:boolean = false
  allowsUnsignedFaucet:boolean = true
  chainId:java.lang.String = "chain-ASdWiN"
  gamete:io.takamaka.code.lang.Account = 4f7d7ca1fbea152d8f323c21e1abcfa1d979c7c4ea667d8457381a26b08a2d71#0
  gasStation:io.takamaka.code.governance.GasStation = 42a8a11aee0405aee5775514b3b0456c7740bbb015b4b8...
  maxCumulativeSizeOfDependencies:long = 10000000
  maxDependencies:int = 20
  maxErrorLength:int = 300
  signature:java.lang.String = "ed25519"
  skipsVerification:boolean = false
  validators:io.takamaka.code.governance.Validators = 42a8a11aee0405aee5775514b3b0456c7740bbb015b4b8...
  versions:io.takamaka.code.governance.Versions = 42a8a11aee0405aee5775514b3b0456c7740bbb015b4b87df4...
^ balance:java.math.BigInteger = 0 (inherited from io.takamaka.code.lang.Contract)
^ balanceRed:java.math.BigInteger = 0 (inherited from io.takamaka.code.lang.Contract)
^ nonce:java.math.BigInteger = 227 (inherited from io.takamaka.code.lang.ExternallyOwnedAccount)
^ publicKey:java.lang.String = "" (inherited from io.takamaka.code.lang.ExternallyOwnedAccount)
```

Creation of a new account

Let us create a new account with 50000000 units of coin:

```
moka create-account 50000000 --payer faucet --url panarea.hotmoka.io
```

```
Free account creation will succeed only if the gamete of the node supports an open unsigned faucet
total gas consumed: 49392
  for CPU: 601
  for RAM: 1478
  for storage: 47313
  for penalty: 0
A new account 06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0 has been created
The keys of the account have been saved
  into the file 06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0.keys
```

Who paid for that?

Gas and coins have been paid by the gamete, then provides an unsigned *faucet*. This is a testnet. A real network has no open unsigned faucet and one must specify the address of the payer account then (with `--payer`)

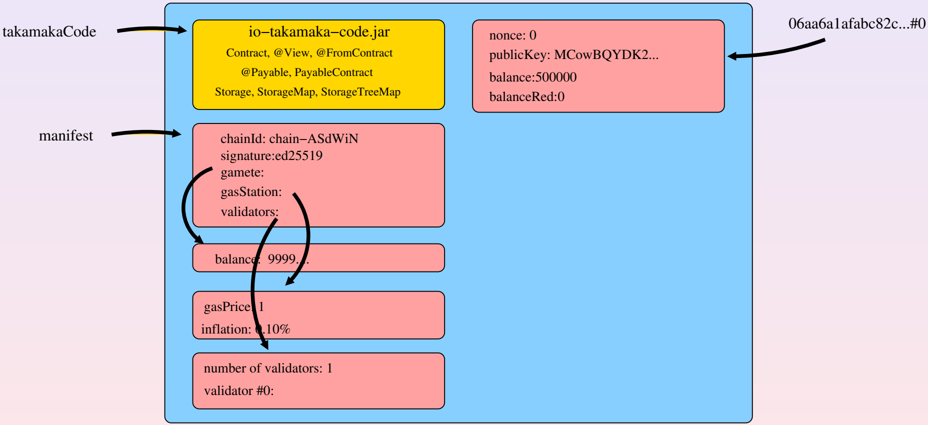
Let us have a look at our first account

```
moka state 06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0 --url panarea.hotmoka.io
```

```
class io.takamaka.code.lang.ExternallyOwnedAccount (from jar installed at 02dfd29348abaa...)
  nonce:java.math.BigInteger = 0
  publicKey:java.lang.String = "MCowBQYDK2VwAyEAk45GxqvRFg88bKZqkqDGxQBHdvvZF+b9YSkl8xs28Ao="
^ balance:java.math.BigInteger = 50000000 (inherited from io.takamaka.code.lang.Contract)
^ balanceRed:java.math.BigInteger = 0 (inherited from io.takamaka.code.lang.Contract)
```

- the nonce starts from 0
- the initial balance is actually 50000000
- the public key is a Base64-encoded ed25519 key
- the public key is stored in the object: no need to send it again at every future method call (like Ethereum does instead)

A new object in state



Let us have a look at the API of our first account

```
moka state 06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0 --api --url

panarea.hotmoka.io
```

```
class io.takamaka.code.lang.ExternallyOwnedAccount (from jar installed at 02dfd29348abaa...)
  nonce:java.math.BigInteger = 0
  publicKey:java.lang.String = "MCowBQYDK2VwAyEAk45GxqvRFg88bKZqkqDGxQBHdvvZF+b9YSkl8xs28Ao="
^ balance:java.math.BigInteger = 50000000 (inherited from io.takamaka.code.lang.Contract)
^ balanceRed:java.math.BigInteger = 0 (inherited from io.takamaka.code.lang.Contract)

  @Payable @FromContract public ExternallyOwnedAccount(java.math.BigInteger,java.lang.String)
  @Payable @FromContract public ExternallyOwnedAccount(long,java.lang.String)
  @Payable @FromContract public ExternallyOwnedAccount(int,java.lang.String)
  public ExternallyOwnedAccount(java.lang.String)

  @View public final java.math.BigInteger nonce()
  @View public final java.lang.String publicKey()
  @View public java.lang.String toString()
^ @View public final java.math.BigInteger balance() (inherited from io.takamaka.code.lang.Contract)
^ @View public final java.math.BigInteger balanceGreen() (inherited from io.takamaka.code.lang.Contract)
^ @View public final java.math.BigInteger balanceRed() (inherited from io.takamaka.code.lang.Contract)
^ public final int compareByStorageReference(io.takamaka.code.lang.Storage)
    (inherited from io.takamaka.code.lang.Storage)
^ public boolean equals(java.lang.Object) (inherited from java.lang.Object)
^ public final native java.lang.Class getClass() (inherited from java.lang.Object)
^ @View public final java.lang.String getClassName() (inherited from io.takamaka.code.lang.Storage)
^ public native int hashCode() (inherited from java.lang.Object)
^ ~~public final native void notify() (inherited from java.lang.Object)~~ X
^ ~~public final native void notifyAll() (inherited from java.lang.Object)~~ X
^ @Payable @FromContract public final void receive(int)
    (inherited from io.takamaka.code.lang.PayableContract)
^ @Payable @FromContract public final void receive(java.math.BigInteger)
    (inherited from io.takamaka.code.lang.PayableContract)
```

Let us call `toString()` on our first account

```
moka call <payer> <receiver> <methodName> [<args>...]
```

We will use our account as payer and as receiver at the same time:

```
moka call 06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0 toString
--url panarea.hotmoka.io
```

```
an externally owned account
calls to @View methods consume no gas
```

`toString()` in class `ExternallyOwnedAccount` is annotated as `@View`

The execution of a method (or constructor)

The request specifies

- payer object, receiver object and actual arguments (*input*)
- classpath, signature of the method
- nonce, chain id, gas price, gas limit
- signature of the request

The computation of the response (ie, the execution of the request)

1. create a classloader form the jar(s) in state for the classpath
2. reconstruct, from the state, RAM objects for *input*
3. execute the method on a normal Java Virtual Machine (in RAM)
4. at the end, identify updates to fields of objects reachable from *input* or return value
5. pack those updates into a response (RAM objects destroyed now)
6. put request/response in state, expanding histories

How can Hotmoka identify updates to fields of objects?

The original code

```
public class C {
  public int i;
  public void foo() {
    i = 42;
  }
}
```

No way to know if `i` changed its value during the execution of `foo()`

How can Hotmoka identify updates to fields of objects?

The instrumented code

```
public class C extends Storage {
  public int i, old_i; // aliased at method start
  public void foo() {
    i = 42;
  }
}
```

`i` changed its value during the execution of `foo()` iff at the end `i`≠`old_i`

How can Hotmoka enforce gas limits?

The original code

```
public class C {
  public void foo() {
    while (...) {
      ...
    }
  }
}
```

This loop might run for very long or even forever

How can Hotmoka enforce gas limits?

The instrumented code

```
static long counter;
public class C {
  public void foo() {
    while (...) {
      if (counter++ >= gaslimit)
        throw new OutOfGasError();
      ...
    }
  }
}
```

Actual gas costs are more fine-grained

Verification and instrumentation of jars in state

Each jar that gets installed in a Hotmoka node undergoes two processes:

1. Verification: absence of frequent errors
   - objects stored in state extend `Storage`
   - non-deterministic or non-terminating library code is not used
   - no synchronization
   - no native code
   - no *dangerous* bytecodes
   - no finalizers
   - no static fields (mostly)
   - code annotations are used correctly
   - ...
2. Instrumentation
   - fields of `Storage` classes get duplicated
   - gas metering is weaved into the code
   - code annotations get implemented *by magic*
   - ...

The Takamaka programming language

Takamaka is the subset of Java that passes the verification of a Hotmoka node. It uses code annotations to implement contract-based aspects:

- `@FromContract` annotates something that can only be called by a contract, not by any other code; hence, it has a `caller()`
- `@Payable` annotates something whose execution requires to pay some cryptocurrency units
- `@View` annotates something whose execution can be run for free, without paying for its gas: it must not generate any update at its end (*pure* code)

Takamaka comes equipped with a support library (`io-takamaka-code`) that defines such annotations and other typical classes that are useful for programming smart contracts

A first example of Takamaka code

1. create a new Java Maven project (skip archetype selection)
2. edit the Maven configuration file `pom.xml` as follows:

```
<project ...>
  <modelVersion>4.0.0</modelVersion>

  <groupId>io.hotmoka</groupId>
  <artifactId>ponzi</artifactId>
  <version>0.0.1</version>

  <properties>
    <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
    <maven.compiler.source>11</maven.compiler.source>
    <maven.compiler.target>11</maven.compiler.target>
    <failOnMissingWebXml>false</failOnMissingWebXml>
  </properties>

  <dependencies> <dependency>
      <groupId>io.hotmoka</groupId>
      <artifactId>io-takamaka-code</artifactId>
      <version>1.0.0</version>
  </dependency> </dependencies>

  <build> <plugins> <plugin>
        <groupId>org.apache.maven.plugins</groupId>
        <artifactId>maven-compiler-plugin</artifactId>
        <version>3.8.1</version>
        <configuration> <release>11</release> </configuration>
  </plugin> </plugins> </build>

</project>
```

A first example of Takamaka code

3. create a source package `io.takamaka.ponzi` inside `src/main/java`
4. create a `module-info.java` in the `src/main/java` directory:

```
module ponzi {
  requires io.takamaka.code;
}
```

A first example of Takamaka code

5. add the following class in package `io.takamaka.ponzi`

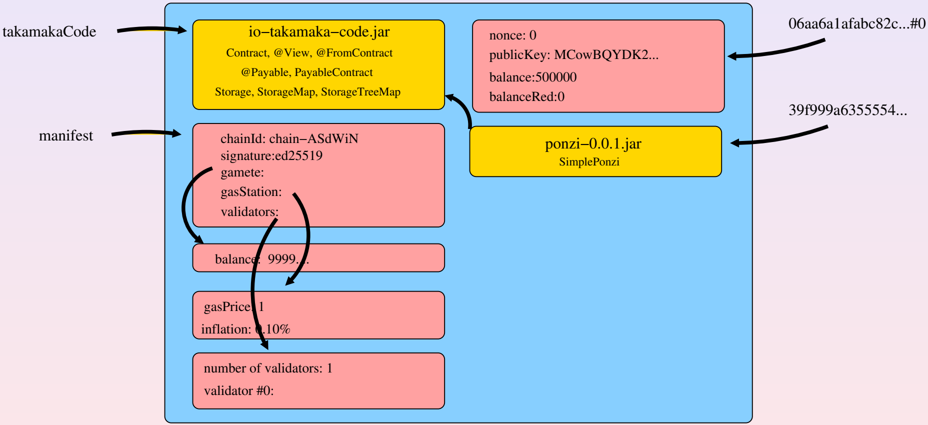
```
package io.takamaka.ponzi;

import static io.takamaka.code.lang.Takamaka.require;
import java.math.BigInteger;
import io.takamaka.code.lang.Contract;
import io.takamaka.code.lang.FromContract;
import io.takamaka.code.lang.Payable;
import io.takamaka.code.lang.PayableContract;
import io.takamaka.code.lang.View;

public class SimplePonzi extends Contract {
  private final BigInteger _10 = BigInteger.valueOf(10L), _11 = BigInteger.valueOf(11L);
  private PayableContract currentInvestor;
  private BigInteger currentInvestment = BigInteger.ZERO;

  public @Payable @FromContract(PayableContract.class) void invest(BigInteger amount) {
    BigInteger minimum = currentInvestment.multiply(_11).divide(_10);
    require(amount.compareTo(minimum) >= 0, () -> "you must invest at least " + minimum);

    if (currentInvestor != null)
      currentInvestor.receive(amount); // no risk of reentrancy

    currentInvestor = (PayableContract) caller();
    currentInvestment = amount;
  }

  public @View BigInteger getCurrentInvestment() {
    return currentInvestment;
  }
}
```

A first example of Takamaka code

6. compile and package everything with Maven:
   `mvn package`
7. the compiled `ponzi-0.0.1.jar` will appear inside the `target` directory of your project, ready to be installed in a Hotmoka node
8. install `ponzi-0.0.1.jar` in our AWS Hotmoka node (`moka install <payer> <jar>`)

```
moka install
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
.../target/ponzi-0.0.1.jar
--url panarea.hotmoka.io

Do you really want to spend up to 443300 gas units to install the jar [Y/N] Y
.../target/ponzi-0.0.1.jar has been installed at
  39f999a63555542eaf5040388d61c20193dee4fb035847a40608c494bf069765
total gas consumed: 298189
  for CPU: 233
  for RAM: 1164
  for storage: 296792
  for penalty: 0
```

A new jar in state



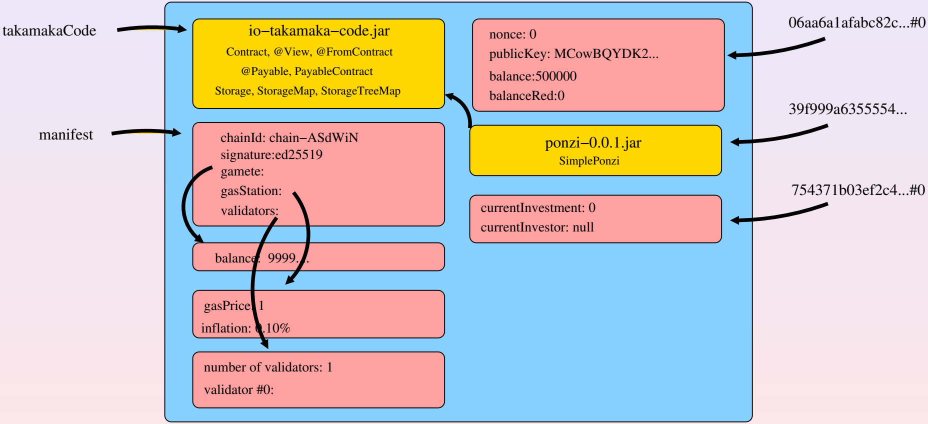
Creation of a new object in state

`moka create <payer> <className> [<args>...]`

We use our account as payer and specify where `SimplePonzi` is defined (the classpath):

```
moka create
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
io.takamaka.ponzi.SimplePonzi
--classpath 39f999a63555542eaf5040388d61c20193dee4fb035847a40608c494bf069765
--url panarea.hotmoka.io

do you really want to spend up to 500000 gas units to call public SimplePonzi() ? [Y/N] Y
the new object has been allocated at 754371b03ef2c413f546e1c3667adf86d545a3587e60f75d2c496863ef442f5c#0
total gas consumed: 42686
  for CPU: 280
  for RAM: 1113
  for storage: 41293
  for penalty: 0
```

A new `SimplePonzi` in state



Call the `invest` method of our contract

```
moka call <payer> <receiver> <methodName> [<args>...]
```

We use our account as payer:

```
moka call
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
754371b03ef2c413f546e1c3667adf86d545a3587e60f75d2c496863ef442f5c#0
invest
40000
--url panarea.hotmoka.io

Do you really want to spend up to 500000 gas units to call
  @Payable @FromContract(PayableContract.class) public void invest(java.math.BigInteger) ? [Y/N] Y
total gas consumed: 10373
  for CPU: 402
  for RAM: 1253
  for storage: 8718
  for penalty: 0
```

Call the `invest` method of our contract, again

```
moka call <payer> <receiver> <methodName> [<args>...]
```

We use our account as payer:

```
moka call
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
754371b03ef2c413f546e1c3667adf86d545a3587e60f75d2c496863ef442f5c#0
invest
40000
--url panarea.hotmoka.io

Do you really want to spend up to 500000 gas units to call
  @Payable @FromContract(PayableContract.class) public void invest(java.math.BigInteger) ? [Y/N] Y
total gas consumed: 500000
  for CPU: 446
  for RAM: 1353
  for storage: 8718
  for penalty: 489483
io.hotmoka.beans.TransactionException: io.takamaka.code.lang.RequirementViolationException:
  you must invest at least 44000@SimplePonzi.java:18
```

A gradual Ponzi scheme

```
package io.hotmoka.examples.ponzi;

import static io.takamaka.code.lang.Takamaka.require;

import java.math.BigInteger;
import io.takamaka.code.lang.Contract;
import io.takamaka.code.lang.FromContract;
import io.takamaka.code.lang.Payable;
import io.takamaka.code.lang.PayableContract;
import io.takamaka.code.util.StorageList;
import io.takamaka.code.util.StorageLinkedList;

public class GradualPonzi extends Contract {
  private final BigInteger MINIMUM = BigInteger.valueOf(1_000L);
  private final StorageList<PayableContract> investors = new StorageLinkedList<>();

  public @FromContract(PayableContract.class) GradualPonzi() {
    investors.add((PayableContract) caller());
  }

  public @Payable @FromContract(PayableContract.class) void invest(BigInteger amount) {
    require(amount.compareTo(MINIMUM) >= 0, () -> "you must invest at least " + MINIMUM);
    BigInteger eachInvestorGets = amount.divide(BigInteger.valueOf(investors.size()));
    investors.stream().forEachOrdered(investor -> investor.receive(eachInvestorGets));
    investors.add((PayableContract) caller());
  }
}
```

An insurance contract

The contract allows one to insure specific days of the year

If it rains on those days, one will get an indemnization larger than the cost of the insurance

- much larger in summer
- just a bit larger in winter

The contract provides the following functionalities:

- construction, upon specification of the oracle:

```
@FromContract @Payable Insurance(BigInteger amount, Contract oracle)
```

- purchase of an insurance for specific days:

```
      @FromContract(PayableContract.class) @Payable void buy
   (long amount, int day, int month, int year, int duration)
```

- notification of rain and indemnization:

```
@FromContract void itRains()
```

An insurance contract

```
public class Insurance extends Contract {
  public final static long MIN = 1_000, MAX = 1_000_000_000;
  private final Contract oracle;
  private final StorageSet<InsuredDay> insuredDays = new StorageTreeSet<>();

  public @FromContract @Payable Insurance(BigInteger amount, Contract oracle) {
    this.oracle = oracle;
  }

  // inner class
  private static class InsuredDay extends Storage { /* shown later */ }

  public @FromContract(PayableContract.class) @Payable void buy
    (long amount, int day, int month, int year, int duration) { /* shown later */ }

  public @FromContract void itRains() { /* shown later */ }
}
```

The inner class

```
private static class InsuredDay extends Storage {
  private final PayableContract payer;
  private final long amount;
  private final int day, month, year;

  private InsuredDay(PayableContract payer, long amount, LocalDate when) {
    this.payer = payer;
    this.amount = amount;
    this.day = when.getDayOfMonth();  this.month = when.getMonthValue();  this.year = when.getYear();
  }

  private boolean isToday() {
    return LocalDate.of(year, month, day).equals(today());
  }

  private boolean isTodayOrBefore() {
    return !LocalDate.of(year, month, day).isAfter(today());
  }

  private static LocalDate today() {
    Instant now = Instant.ofEpochMilli(Takamaka.now());
    return LocalDate.ofInstant(now, ZoneId.of("Europe/Rome"));
  }

  private long indemnization() {
    switch (Season.now()) {  // Season is an enumeration
      case WINTER: return amount * 18 / 10; // 180%
      case SPRING: return amount * 30 / 10; // 300%
      case SUMMER: return amount * 50 / 10; // 500%
      default /* FALL */ : return amount * 28 / 10; // 280%
    }
  }
}
```

Buy an insurance

```
public @FromContract(PayableContract.class) @Payable void buy
    (long amount, int day, int month, int year, int duration) {

  require(duration >= 1, "you must insure at least one day");
  require(duration <= 7, "you cannot insure more than a week");
  require(amount >= MIN * duration,
    () -> "we insure a single day for at least " + MIN + " units of coin");
  require(amount <= MAX * duration,
    () -> "we insure a single day for up to " + MAX + " units of coin");

  // if the date is wrong, this generates an exception
  LocalDate start = LocalDate.of(year, month, day);

  PayableContract payer = (PayableContract) caller();
  for (int offset = 0; offset < duration; offset++)
    insuredDays.add(new InsuredDay
                    (payer, amount / duration, start.plusDays(offset)));
}
```

Pay the indemnization

```java
public @FromContract void itRains() {
  require(caller() == oracle, "only the oracle can call this method");

  // pay who insured today
  insuredDays.stream()
    .filter(InsuredDay::isToday)
    .forEachOrdered(insuredDay ->
          insuredDay.payer.receive(insuredDay.indemnization()));

  // clean-up the set of insured days
  insuredDays.stream()
    .filter(InsuredDay::isTodayOrBefore)
    .forEachOrdered(insuredDays::remove);
}
```

Installation of the jar into a remote node

`mvn clean package` ⇒ generates `target/insurance-0.0.1.jar`

```
moka install <payer> <jar>
```

We use one of our accounts as payer

```
moka install
```

Installation of the jar into a remote node

`mvn clean package` ⇒ generates `target/insurance-0.0.1.jar`

```
moka install <payer> <jar>
```

We use one of our accounts as payer

```
moka install
payer
```

Installation of the jar into a remote node

`mvn clean package` ⇒ generates `target/insurance-0.0.1.jar`

```
moka install <payer> <jar>
```

We use one of our accounts as payer

```
moka install
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
```

Installation of the jar into a remote node

`mvn clean package` ⇒ generates `target/insurance-0.0.1.jar`

```
moka install <payer> <jar>
```

We use one of our accounts as payer

```
moka install
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
jar
```

Installation of the jar into a remote node

`mvn clean package` ⇒ generates `target/insurance-0.0.1.jar`

```
moka install <payer> <jar>
```

We use one of our accounts as payer

```
moka install
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
target/insurance-0.0.1.jar
```

Installation of the jar into a remote node

`mvn clean package` ⇒ generates `target/insurance-0.0.1.jar`

```
moka install <payer> <jar>
```

We use one of our accounts as payer

```
moka install
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
target/insurance-0.0.1.jar
--url panarea.hotmoka.io
```

Installation of the jar into a remote node

`mvn clean package` ⇒ generates `target/insurance-0.0.1.jar`

`moka install <payer> <jar>`

We use one of our accounts as payer

```
moka install
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
target/insurance-0.0.1.jar
--url panarea.hotmoka.io

Do you really want to spend up to 853900 gas units to install the jar [Y/N] Y
target/insurance-0.0.1.jar has been installed
at acb76103738dc7091c867c31bf6fb351f4d07b4bae1c7972e0e3bc3fcbf9e9c3
total gas consumed: 779976
  for CPU: 255
  for RAM: 1326
  for storage: 778395
  for penalty: 0
```

Creation of an instance of the Insurance contract

```
moka create <payer> <class> <args> --classpath <jar>
```

We use one of our accounts as payer and a previously existing oracle as argument for the constructor

```
moka create
```

Creation of an instance of the `Insurance` contract

```
moka create <payer> <class> <args> --classpath <jar>
```

We use one of our accounts as payer and a previously existing oracle as argument for the constructor

```
moka create
payer
```

Creation of an instance of the `Insurance` contract

```
moka create <payer> <class> <args> --classpath <jar>
```

We use one of our accounts as payer and a previously existing oracle as argument for the constructor

```
moka create
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
```

Creation of an instance of the `Insurance` contract

```
moka create <payer> <class> <args> --classpath <jar>
```

We use one of our accounts as payer and a previously existing oracle as argument for the constructor

```
moka create
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
class
```

Creation of an instance of the `Insurance` contract

```
moka create <payer> <class> <args> --classpath <jar>
```

We use one of our accounts as payer and a previously existing oracle as argument for the constructor

```
moka create
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
it.univr.insurance.Insurance
```

Creation of an instance of the `Insurance` contract

```
moka create <payer> <class> <args> --classpath <jar>
```

We use one of our accounts as payer and a previously existing oracle as argument for the constructor

```
moka create
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
it.univr.insurance.Insurance
args(amount, oracle)
```

Creation of an instance of the `Insurance` contract

```
moka create <payer> <class> <args> --classpath <jar>
```

We use one of our accounts as payer and a previously existing oracle as argument for the constructor

```
moka create
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
it.univr.insurance.Insurance
1000000000000 06a080bbc4712862f875eefad00f43dee8f7daf98aec54c984d20861e3a219e6#0
```

Creation of an instance of the `Insurance` contract

```
moka create <payer> <class> <args> --classpath <jar>
```

We use one of our accounts as payer and a previously existing oracle as argument for the constructor

```
moka create
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
it.univr.insurance.Insurance
1000000000000 06a080bbc4712862f875eefad00f43dee8f7daf98aec54c984d20861e3a219e6#0
--classpath jar
```

Creation of an instance of the `Insurance` contract

```
moka create <payer> <class> <args> --classpath <jar>
```

We use one of our accounts as payer and a previously existing oracle as argument for the constructor

```
moka create
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
it.univr.insurance.Insurance
1000000000000 06a080bbc4712862f875eefad00f43dee8f7daf98aec54c984d20861e3a219e6#0
--classpath acb76103738dc7091c867c31bf6fb351f4d07b4bae1c7972e0e3bc3fcbf9e9c3
```

Creation of an instance of the `Insurance` contract

```
moka create <payer> <class> <args> --classpath <jar>
```

We use one of our accounts as payer and a previously existing oracle as argument for the constructor

```
moka create
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
it.univr.insurance.Insurance
1000000000000 06a080bbc4712862f875eefad00f43dee8f7daf98aec54c984d20861e3a219e6#0
--classpath acb76103738dc7091c867c31bf6fb351f4d07b4bae1c7972e0e3bc3fcbf9e9c3
--url panarea.hotmoka.io
```

Creation of an instance of the `Insurance` contract

```
moka create <payer> <class> <args> --classpath <jar>
```

We use one of our accounts as payer and a previously existing oracle as argument for the constructor

```
moka create
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
it.univr.insurance.Insurance
1000000000000 06a080bbc4712862f875eefad00f43dee8f7daf98aec54c984d20861e3a219e6#0
--classpath acb76103738dc7091c867c31bf6fb351f4d07b4bae1c7972e0e3bc3fcbf9e9c3
--url panarea.hotmoka.io

Do you really want to spend up to 500000 gas units to call
public Insurance(java.math.BigInteger,io.takamaka.code.lang.Contract) ? [Y/N] Y

The new object has been allocated
at ffb8b8455979777e81708e3abac85b79ad455dfe8aa6a9849b11241c9c8ad7f7#0
total gas consumed: 47390
  for CPU: 393
  for RAM: 1315
  for storage: 45682
  for penalty: 0
```

Let's insure next weekend against rain

```
moka call <payer> <receiver> <methodName> <args>
```

We use one of our accounts as the buyer of the insurance

```
moka call
```

Let's insure next weekend against rain

```
moka call <payer> <receiver> <methodName> <args>
```

We use one of our accounts as the buyer of the insurance

```
moka call
payer
```

Let's insure next weekend against rain

```
moka call <payer> <receiver> <methodName> <args>
```

We use one of our accounts as the buyer of the insurance

```
moka call
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
```

Let's insure next weekend against rain

```
moka call <payer> <receiver> <methodName> <args>
```

We use one of our accounts as the buyer of the insurance

```
moka call
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
receiver
```

Let's insure next weekend against rain

```
moka call <payer> <receiver> <methodName> <args>
```

We use one of our accounts as the buyer of the insurance

```
moka call
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
ffb8b8455979777e81708e3abac85b79ad455dfe8aa6a9849b11241c9c8ad7f7#0
```

Let’s insure next weekend against rain

```
moka call <payer> <receiver> <methodName> <args>
```

We use one of our accounts as the buyer of the insurance

```
moka call
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
ffb8b8455979777e81708e3abac85b79ad455dfe8aa6a9849b11241c9c8ad7f7#0
methodName
```

Let’s insure next weekend against rain

```
moka call <payer> <receiver> <methodName> <args>
```

We use one of our accounts as the buyer of the insurance

```
moka call
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
ffb8b8455979777e81708e3abac85b79ad455dfe8aa6a9849b11241c9c8ad7f7#0
buy
```

Let’s insure next weekend against rain

```
moka call <payer> <receiver> <methodName> <args>
```

We use one of our accounts as the buyer of the insurance

```
moka call
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
ffb8b8455979777e81708e3abac85b79ad455dfe8aa6a9849b11241c9c8ad7f7#0
buy
args(amount, day, month, year, duration)
```

Let’s insure next weekend against rain

```
moka call <payer> <receiver> <methodName> <args>
```

We use one of our accounts as the buyer of the insurance

```
moka call
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
ffb8b8455979777e81708e3abac85b79ad455dfe8aa6a9849b11241c9c8ad7f7#0
buy
1000000 10 4 2021 2
```

Let's insure next weekend against rain

```
moka call <payer> <receiver> <methodName> <args>
```

We use one of our accounts as the buyer of the insurance

```
moka call
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
ffb8b8455979777e81708e3abac85b79ad455dfe8aa6a9849b11241c9c8ad7f7#0
buy
1000000 10 4 2021 2
--url panarea.hotmoka.io
```

Let’s insure next weekend against rain

```
moka call <payer> <receiver> <methodName> <args>
```

We use one of our accounts as the buyer of the insurance

```
 moka call
 06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e
 ffb8b8455979777e81708e3abac85b79ad455dfe8aa6a9849b11241c9c8ad7f7#0
 buy
 1000000 10 4 2021 2
 --url panarea.hotmoka.io

Do you really want to spend up to 500000 gas units to call
public void buy(long,int,int,int,int) ? [Y/N] Y
total gas consumed: 12340
  for CPU: 1587
  for RAM: 2943
  for storage: 7810
  for penalty: 0
```

On-chain verification: incorrect use of annotations

Assume that the programmer forgets the `FromContract` annotation in `buy`

public ~~@FromContract(PayableContract.class)~~ @Payable void buy (long amount, int day, int month, int year, int duration)

On-chain verification: incorrect use of annotations

Assume that the programmer forgets the `FromContract` annotation in `buy`

public ~~@FromContract(PayableContract.class)~~ @Payable void buy (long amount, int day, int month, int year, int duration)

`mvn clean package` ⇒ regenerates `target/insurance-0.0.1.jar`

On-chain verification: incorrect use of annotations

Assume that the programmer forgets the `FromContract` annotation in `buy`

```
public ~~@FromContract(PayableContract.class)~~ @Payable void buy (long
amount, int day, int month, int year, int duration)
```

`mvn clean package` ⇒ regenerates `target/insurance-0.0.1.jar`

Let's try to install this version of the jar

```
moka install
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
target/insurance-0.0.1.jar
--url panarea.hotmoka.io
```

On-chain verification: incorrect use of annotations

Assume that the programmer forgets the `FromContract` annotation in `buy`

public ~~@FromContract(PayableContract.class)~~ @Payable void buy (long amount, int day, int month, int year, int duration)

`mvn clean package` ⇒ regenerates `target/insurance-0.0.1.jar`

Let's try to install this version of the jar

```
 moka install
 06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
 target/insurance-0.0.1.jar
 --url panarea.hotmoka.io

Do you really want to spend up to 852500 gas units to install the jar [Y/N] Y
total gas consumed: 852500
  for CPU: 255
  for RAM: 1326
  for storage: 381762
  for penalty: 469157     !!!!!!!
io.hotmoka.beans.TransactionException:
io.takamaka.code.verification.VerificationException:
it/univr/insurance/Insurance.java method buy:
@Payable can only be applied to a @FromContract method or constructor
```

On-chain verification: potential non-determinism

Assume to use `forEach` instead of `forEachOrdered` in `itRains`

```
insuredDays.stream().filter(InsuredDay::isToday).forEach(...);
```

On-chain verification: potential non-determinism

Assume to use `forEach` instead of `forEachOrdered` in `itRains`

```
insuredDays.stream().filter(InsuredDay::isToday).forEach(...);
```

`mvn clean package` ⇒ regenerates `target/insurance-0.0.1.jar`

On-chain verification: potential non-determinism

Assume to use `forEach` instead of `forEachOrdered` in `itRains`

```
insuredDays.stream().filter(InsuredDay::isToday).forEach(...);
```

`mvn clean package` ⇒ regenerates `target/insurance-0.0.1.jar`

Let’s try to install this version of the jar

```
moka install
06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
target/insurance-0.0.1.jar
--url panarea.hotmoka.io
```

On-chain verification: potential non-determinism

Assume to use `forEach` instead of `forEachOrdered` in `itRains`

```
insuredDays.stream().filter(InsuredDay::isToday).forEach(...);
```

`mvn clean package` ⇒ regenerates `target/insurance-0.0.1.jar`

Let's try to install this version of the jar

```
 moka install
 06aa6a1afabc82c7161ffcdc2391a2136101aaeb94f64edd53a1d0d1436d610e#0
 target/insurance-0.0.1.jar
 --url panarea.hotmoka.io

Do you really want to spend up to 852500 gas units to install the jar [Y/N] Y
total gas consumed: 853700
  for CPU: 255
  for RAM: 1326
  for storage: 382362
  for penalty: 469757     !!!!!!!
io.hotmoka.beans.TransactionException:
io.takamaka.code.verification.VerificationException:
it/univr/insurance/Insurance.java:95:
illegal call to non-white-listed method java.util.stream.Stream.forEach
```

Off-chain verification

Using the blockchain as a debugger is very expensive...

```
moka verify <jar> --libs dependencies
```

Off-chain verification

Using the blockchain as a debugger is very expensive. . .

```
moka verify <jar> --libs dependencies
```

We verify the jar off-chain, to find all errors

```
moka verify
```

Off-chain verification

Using the blockchain as a debugger is very expensive. . .

```
moka verify <jar> --libs dependencies
```

We verify the jar off-chain, to find all errors

```
moka verify
jar
```

Off-chain verification

Using the blockchain as a debugger is very expensive. . .

```
moka verify <jar> --libs dependencies
```

We verify the jar off-chain, to find all errors

```
moka verify
target/insurance-0.0.1.jar
```

Off-chain verification

Using the blockchain as a debugger is very expensive. . .

```
moka verify <jar> --libs dependencies
```

We verify the jar off-chain, to find all errors

```
moka verify
target/insurance-0.0.1.jar
--libs dependencies
```

Off-chain verification

Using the blockchain as a debugger is very expensive. . .

```
moka verify <jar> --libs dependencies
```

We verify the jar off-chain, to find all errors

```
moka verify
target/insurance-0.0.1.jar
--libs io-takamaka-code-1.0.0.jar
```

Off-chain verification

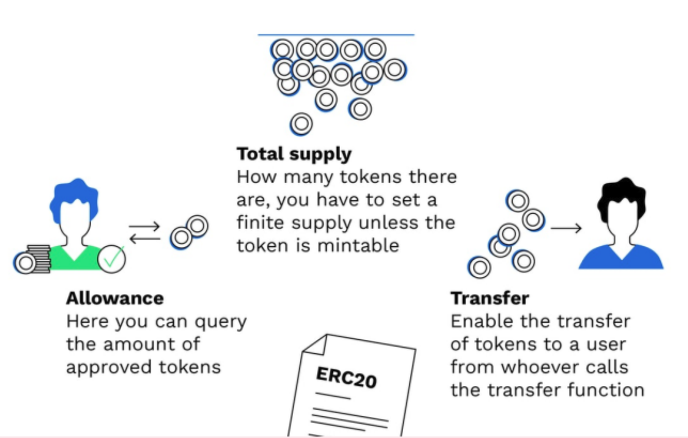
Using the blockchain as a debugger is very expensive. . .

```
moka verify <jar> --libs dependencies
```

We verify the jar off-chain, to find all errors

```
 moka verify
 target/insurance-0.0.1.jar
 --libs io-takamaka-code-1.0.0.jar



it/univr/insurance/Insurance.java method buy:
  @Payable can only be applied to a @FromContract method or constructor
it/univr/insurance/Insurance.java:46:
  caller() can only be used inside a @FromContract method or constructor
it/univr/insurance/Insurance.java:95:
  illegal call to non-white-listed method java.util.stream.Stream.forEach
it/univr/insurance/Insurance.java:99:
  illegal call to non-white-listed method java.util.stream.Stream.forEach
```

ERC20 Tokens

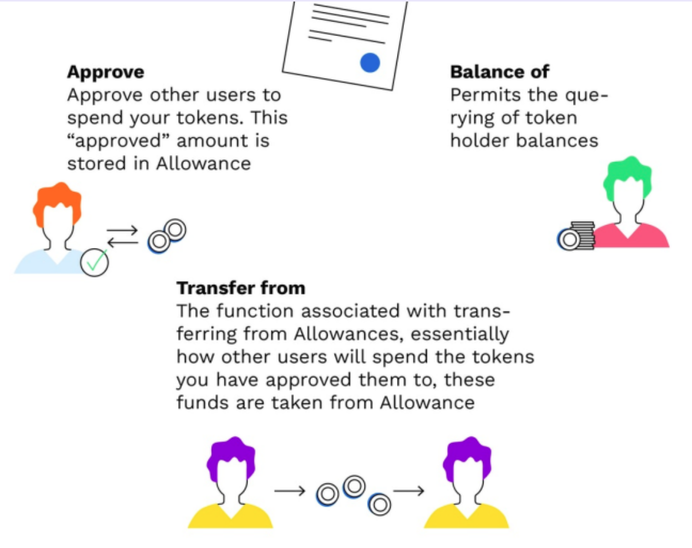


Total supply
How many tokens there are, you have to set a finite supply unless the token is mintable

Allowance
Here you can query the amount of approved tokens

Transfer
Enable the transfer of tokens to a user from whoever calls the transfer function

ERC20 Tokens

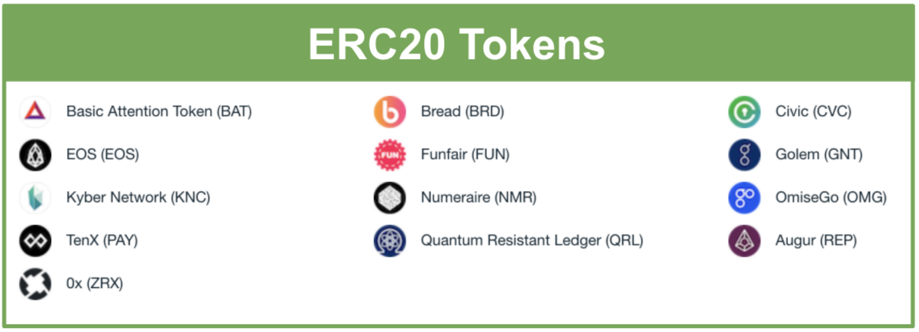


Approve
Approve other users to spend your tokens. This “approved” amount is stored in Allowance

Balance of
Permits the querying of token holder balances

Transfer from
The function associated with transferring from Allowances, essentially how other users will spend the tokens you have approved them to, these funds are taken from Allowance

There are many. . .



ERC20 Tokens

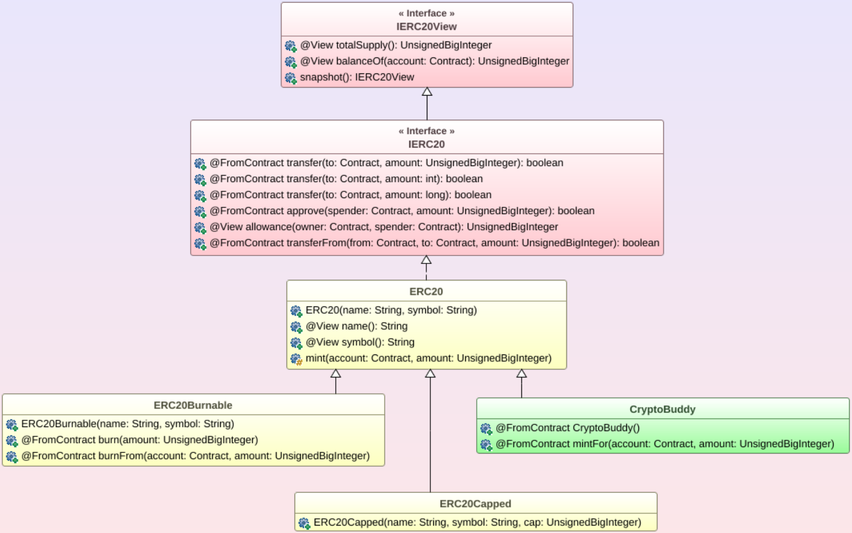
- Basic Attention Token (BAT)
- Bread (BRD)
- Civic (CVC)
- EOS (EOS)
- Funfair (FUN)
- Golem (GNT)
- Kyber Network (KNC)
- Numeraire (NMR)
- OmiseGo (OMG)
- TenX (PAY)
- Quantum Resistant Ledger (QRL)
- Augur (REP)
- 0x (ZRX)

The OpenZeppelin reference implementation

We follow (in part) the implementation by OpenZeppelin

https://docs.openzeppelin.com/contracts/2.x/api/token/erc20

The hierarchy of the implementation*



« Interface »
IERC20View

- @View totalSupply(): UnsignedBigInteger
- @View balanceOf(account: Contract): UnsignedBigInteger
- snapshot(): IERC20View

« Interface »
IERC20

- @FromContract transfer(to: Contract, amount: UnsignedBigInteger): boolean
- @FromContract transfer(to: Contract, amount: int): boolean
- @FromContract transfer(to: Contract, amount: long): boolean
- @FromContract approve(spender: Contract, amount: UnsignedBigInteger): boolean
- @View allowance(owner: Contract, spender: Contract): UnsignedBigInteger
- @FromContract transferFrom(from: Contract, to: Contract, amount: UnsignedBigInteger): boolean

ERC20

- ERC20(name: String, symbol: String)
- @View name(): String
- @View symbol(): String
- mint(account: Contract, amount: UnsignedBigInteger)

ERC20Burnable

- ERC20Burnable(name: String, symbol: String)
- @FromContract burn(amount: UnsignedBigInteger)
- @FromContract burnFrom(account: Contract, amount: UnsignedBigInteger)

CryptoBuddy

- @FromContract CryptoBuddy()
- @FromContract mintFor(account: Contract, amount: UnsignedBigInteger)

ERC20Capped

- ERC20Capped(name: String, symbol: String, cap: UnsignedBigInteger)

* Joint work with Marco Crosara

The IERC20View interface

This interface represents the read-only part of the standard. It allows one to create a snaphot of a token, that is, a fixed, immutable snapshot of its balances

```
public interface IERC20View {
  @View UnsignedBigInteger totalSupply();
  @View UnsignedBigInteger balanceOf(Contract account);
  IERC20View snapshot(); // not in OpenZeppelin
}
```

The `IERC20` interface

This interface represents the transfer part of the standard. Methods that require to identify their caller are annotated as `@FromContract`

```java
public interface IERC20 extends IERC20View {
  @FromContract boolean transfer(Contract recipient, UnsignedBigInteger amount);
  @View UnsignedBigInteger allowance(Contract owner, Contract spender);
  @FromContract boolean approve(Contract spender, UnsignedBigInteger amount);
  @FromContract boolean transferFrom
      (Contract sender, Contract recipient, UnsignedBigInteger amount);

  class Transfer extends Event {
    public final Contract from, to;
    public final UnsignedBigInteger value;

    @FromContract Transfer(Contract from, Contract to, UnsignedBigInteger value) {
      this.from = from;
      this.to = to;
      this.value = value;
    }
  }
}
```

The ERC20 implementation: state

```
public class ERC20 extends Contract implements IERC20 {
  private final StorageMap<Contract, UnsignedBigInteger>
    balances = new StorageTreeMap<>();

  private final StorageMap<Contract, StorageMap<Contract, UnsignedBigInteger>>
    allowances = new StorageTreeMap<>();

  public final UnsignedBigInteger ZERO = new UnsignedBigInteger("0");

  // there are public accessors to these fields (not shown)
  private UnsignedBigInteger totalSupply = ZERO;
  private final String name;
  private final String symbol;
  private short decimals;

  public ERC20(String name, String symbol) {
    this.name = name;
    this.symbol = symbol;
    this.decimals = 18;
  }
}
```

The ERC20 implementation: minting

```
public class ERC20 extends Contract implements IERC20 {

  public final @View UnsignedBigInteger balanceOf(Contract account) {
    return balances.getOrDefault(account, ZERO);
  }

  protected void _mint(Contract account, UnsignedBigInteger amount) {
    require(account != null, "Mint rejected: mint to the null account");
    require(amount != null, "Mint rejected: amount cannot be null");

    beforeTokenTransfer(null, account, amount);
    totalSupply = totalSupply.add(amount);
    balances.put(account, balanceOf(account).add(amount));
  }

  protected void beforeTokenTransfer
      (Contract from, Contract to, UnsignedBigInteger amount) { }
}
```

The ERC20 implementation: transfer

```
public class ERC20 extends Contract implements IERC20 {

  public @FromContract boolean transfer(Contract to, UnsignedBigInteger amount) {
    transfer(caller(), to, amount);
    return true;
  }

  protected void transfer(Contract from, Contract to, UnsignedBigInteger amount) {
    require(from != null, "Transfer rejected: transfer from the null account");
    require(to != null, "Transfer rejected: transfer to the null account");
    require(amount != null, "Transfer rejected: amount cannot be null");

    beforeTokenTransfer(from, to, amount);
    balances.put(from, balancesOf(from).subtract(amount,
      "Transfer rejected: transfer amount exceeds balance"));
    balances.put(to, balanceOf(to).add(amount));
    event(new Transfer(from, to, amount));
  }
}
```

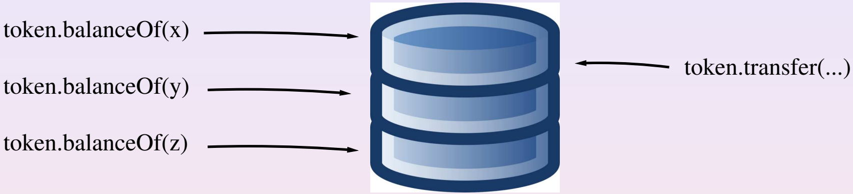
The ERC20 implementation: approval

```
public class ERC20 extends Contract implements IERC20 {

  public @View UnsignedBigInteger allowance(Contract owner, Contract spender) {
    return _allowances.getOrDefault(owner, StorageTreeMap::new)
                            .getOrDefault(spender, ZERO);
  }

  public @FromContract boolean approve(Contract spender, UnsignedBigInteger amount)
    approve(caller(), spender, amount);
    return true;
  }

  protected void approve(Contract owner, Contract spender, UnsignedBigInteger amoun
    require(owner != null, "Approve rejected: approve from the null account");
    require(spender != null, "Approve rejected: approve to the null account");
    require(amount != null, "Approve rejected: amount cannot be null");

    StorageMap<Contract, UnsignedBigInteger> ownerAllowances
      = allowances.getOrDefault(owner, StorageTreeMap::new);
    ownerAllowances.put(spender, amount);
    allowances.put(owner, ownerAllowances);
    event(new Approval(owner, spender, amount));
  }
}
```

The ERC20 implementation: transfer from

```
public class ERC20 extends Contract implements IERC20 {

  public @FromContract boolean transferFrom
            (Contract from, Contract to, UnsignedBigInteger amount) {

    transferFrom(caller(), from, to, amount);
    return true;
  }

  protected final void transferFrom
        (Contract caller, Contract from, Contract to, UnsignedBigInteger amount) {

    transfer(from, to, amount);
    approve(from, caller, allowance(from, caller).subtract(amount,
      "Transfer Rejected: transfer amount exceeds allowance"));
  }
}
```

Inconsistent view



token.balanceOf(x)

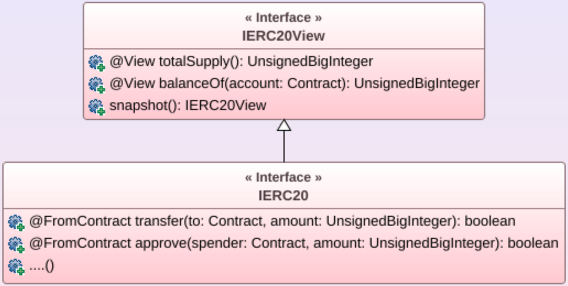
token.balanceOf(y)

token.balanceOf(z)

token.transfer(...)

Between a call to `balanceOf` and the next, the state of the token might change in the database because other users might call the transfer functions, concurrently

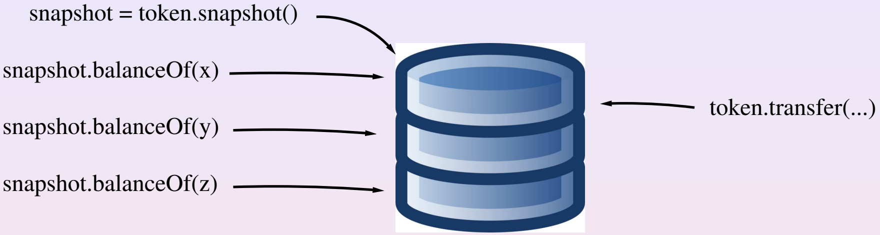
The view/snapshot design pattern


« Interface »
IERC20View
@View totalSupply(): UnsignedBigInteger
@View balanceOf(account: Contract): UnsignedBigInteger
snapshot(): IERC20View
« Interface »
IERC20
@FromContract transfer(to: Contract, amount: UnsignedBigInteger): boolean
@FromContract approve(spender: Contract, amount: UnsignedBigInteger): boolean
....()

An external wallet calls `snapshot()` to get a consistent, immutable view of the token and can then access its content, while other clients can modify the original token

- impossible in Solidity, where maps cannot be cloned

Inconsistent view



- all calls to `balanceOf` refer to the same, consistent state of the token (possibly not the latest)
- the implementation might keep a snapshot and provide a `@View` method to read it

The ERC20 implementation: snapshot in O(1)

```
public class ERC20 extends Contract implements IERC20 {

  public IERC20View snapshot() {   // O(1)

    class SnapshotImpl extends Storage implements IERC20View {
      private final UnsignedBigInteger totalSupply = ERC20.this.totalSupply;
      private final StorageMapView<Contract, UnsignedBigInteger> balances
        = ERC20.this.balances.snapshot(); // O(1)

      public @View UnsignedBigInteger totalSupply() {
        return totalSupply;
      }

      public @View UnsignedBigInteger balanceOf(Contract account) {
        return balances.getOrDefault(account, ZERO);
      }

      public IERC20View snapshot() {
        return this; // a snapshot of a snapshot is the snapshot itself
      }
    }

    return new SnapshotImpl();
  }
}
```

Definition of a new token

```
public class CryptoBuddy extends ERC20 {
  private final Contract owner;

  public @FromContract CryptoBuddy() {
    super("CryptoBuddy", "CB");

    owner = caller();
    UnsignedBigInteger initialSupply = new UnsignedBigInteger("200000");
    UnsignedBigInteger multiplier = new UnsignedBigInteger("10").pow(18);
    _mint(caller(), initialSupply.multiply(multiplier)); // 200’000 * 10 ^ 18
  }

  public @FromContract void mintFor(Contract account, UnsignedBigInteger amount) {
    require(caller() == owner, "Lack of permission");
    _mint(account, amount);
  }
}
```

Exercise

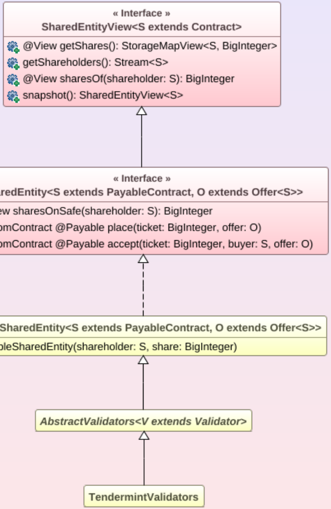
Define your own token

1. implement the token in Takamaka (possibly using Eclipse or similar IDE)
2. generate the jar of your new token
3. use the Hotmoka moka to install the jar in blockchain
4. use the Hotmoka moka to instantiate the token
5. use the Hotmoka moka to interact with the token

A shared entity contract for governance

- the entity is initially split among shareholders, each in general holding a distinct number of shares
- a shareholder can place an offer to sell some or all of its shares
- a buyer can accept an offer and become a shareholder

The hierarchy of the implementation*



« Interface »
SharedEntityView<S extends Contract>

@View getShares(): StorageMapView<S, BigInteger>
getShareholders(): Stream<S>
@View sharesOf(shareholder: S): BigInteger
snapshot(): SharedEntityView<S>

« Interface »
SharedEntity<S extends PayableContract, O extends Offer<S>>

@View sharesOnSafe(shareholder: S): BigInteger
@FromContract @Payable place(ticket: BigInteger, offer: O)
@FromContract @Payable accept(ticket: BigInteger, buyer: S, offer: O)

SimpleSharedEntity<S extends PayableContract, O extends Offer<S>>

SimpleSharedEntity(shareholder: S, share: BigInteger)

AbstractValidators<V extends Validator>

TendermintValidators

* Joint work with Andrea Benini

The SharedEntityView interface

```
public interface SharedEntityView<S extends Contract> {
  @View StorageMapView<S, BigInteger> getShares();
  Stream<S> getShareholders();
  @View BigInteger sharesOf(S shareholder);
  SharedEntityView<S> snapshot();
}
```

Note: generic types do not exist in Solidity

The `SharedEntity` interface: adds a market of shares

```
public interface SharedEntity<S extends PayableContract, O extends Offer<S>>
    extends SharedEntityView<S> {

  @View BigInteger sharesOnSaleOf(S shareholder);

  @FromContract @Payable void place(BigInteger ticket, O offer);

  @FromContract @Payable void accept(BigInteger ticket, S buyer, O offer);
}
```

- who sells its shares receives a payment: hence `S` is constrained to extend `PayableContract`
- Java does not allow one to write `@FromContract(S.class)`
- the interface allows implementations to ask for a ticket when offers are placed and accepted

The Offer inner class of SharedEntity

```
public static class Offer<S extends PayableContract> extends Storage {
  public final S seller;
  public final BigInteger sharesOnSale;
  public final BigInteger cost;
  public final long expiration;

  public @FromContract Offer(S seller, BigInteger sharesOnSale, BigInteger cost, long duration) {
    require(caller() == seller, "only the owner can sell its shares");
    require(sharesOnSale != null && sharesOnSale.signum() > 0, "the shares on sale must be positive");
    require(cost != null && cost.signum() >= 0, "the cost must be a non-negative big integer");
    require(duration >= 0, "the duration cannot be negative");

    this.seller = seller;
    this.sharesOnSale = sharesOnSale;
    this.cost = cost;
    this.expiration = now() + duration;
  }

  public @View boolean isOngoing() {
    return now() <= expiration;
  }
}
```

The SimpleSharedEntity implementation: state

```
public class SimpleSharedEntity<S extends PayableContract, O extends Offer<S>>
      extends PayableContract implements SharedEntity<S, O> {

  private final StorageTreeMap<S, BigInteger> shares = new StorageTreeMap<>();
  private final StorageSet<O> offers = new StorageTreeSet<>();
  private StorageMapView<S, BigInteger> snapshotOfShares;

  public SimpleSharedEntity(S shareholder, BigInteger share) { // simple case
    shares.put(shareholder, share);
    snapshotOfShares = shares.snapshot();
  }

  public @View final StorageMapView<S, BigInteger> getShares() {
    return snapshotOfShares;
  }

  public final Stream<S> getShareholders() {
    return snapshotOfShares.keys();
  }

  public final @View BigInteger sharesOf(S shareholder) {
    return snapshotOfShares.getOrDefault(shareholder, ZERO);
  }
}
```

The SimpleSharedEntity implementation: shares on sale

Add the shares on sale of every offer placed by the shareholder and still ongoing

```
public final @View BigInteger sharesOnSaleOf(S shareholder) {
  return offers.stream()
    .filter(offer -> offer.seller == shareholder && offer.isOngoing())
    .map(offer -> offer.sharesOnSale)
    .reduce(ZERO, BigInteger::add);
}
```

The `SimpleSharedEntity` implementation: place an offer

If the offer is valid, add it to the set of offers, removing expired offers on the way

```
public @FromContract @Payable void place(BigInteger ticket, O offer) {
  require(offer.seller == caller(), "only the seller can place its own offer");
  require(shares.containsKey(offer.seller), "the seller is not a shareholder");
  require(sharesOf(offer.seller).subtract(sharesOnSaleOf(offer.seller))
    .compareTo(offer.sharesOnSale) >= 0, "the seller has not enough shares");
  cleanUpOffers(null);
  offers.add(offer);
}

private void cleanUpOffers(O offerToRemove) {
  offers.stream()
    .filter(offer -> offer == offerToRemove || !offer.isOngoing())
    .forEachOrdered(offers::remove);
}
```

The SimpleSharedEntity implementation: accept an offer

If the offer is valid, remove it, swap the shares and send the payment to the seller

```
public @FromContract @Payable void accept(BigInteger ticket, S buyer, O offer) {
  require(caller() == buyer, "only the future owner can buy the shares");
  require(offers.contains(offer), "unknown offer");
  require(offer.isOngoing(), "the sale offer is not ongoing anymore");
  require(offer.cost.compareTo(ticket) <= 0, "not enough money for the offer");
  cleanUpOffers(offer);
  removeShares(offer.seller, offer.sharesOnSale);
  addShares(buyer, offer.sharesOnSale);
  offer.seller.receive(offer.cost);
  snapshotOfShares = shares.snapshot();
}

private void addShares(S shareholder, BigInteger added) {
  shares.update(shareholder, BigInteger.ZERO, added::add);
}

private void removeShares(S shareholder, BigInteger removed) {
  ... similar
}
```

The SimpleSharedEntity implementation: snapshot

Save the current snapshot of the shares, to answer all questions

```
public final SharedEntityView<S> snapshot() {

  class SharedEntitySnapshotImpl implements SharedEntityView<S> {
    private final StorageMapView<S, BigInteger> snapshotOfShares
        = SimpleSharedEntity.this.snapshotOfShares;

    public StorageMapView<S, BigInteger> getShares() { return snapshotOfShares; }

    public Stream<S> getShareholders() {
      return snapshotOfShares.keys();
    }

    public BigInteger sharesOf(S shareholder) {
      return snapshotOfShares.getOrDefault(shareholder, ZERO);
    }

    public SharedEntityView<S> snapshot() { return this; }
  };

  return new SharedEntitySnapshotImpl();
}
```

Exercise

Define a lottery contract

1. the creator (a payable contract) specifies, at the lottery creation time, the number $N \geq 2$ of tickets to sell
2. payable contracts can call a `void buy()` method of the lottery to buy a ticket (possibly many times). The ticket costs $100000 \cdot \left(1 + \frac{\text{number of tickets already sold}}{N-1}\right)$
3. when the Nth ticket is sold, the method `buy` computes `now() % N` and determines the number of the winning contract
   - it sends 90% of the balance of the lottery to the winner
   - it sends the remaining 10% of the balance of the lottery to its creator
   - it rejects all future calls to `buy`
4. install the contract in blockchain and play with it by using the `moka` tool

Suggestion: keep the ticket holders inside a `io.takamaka.code.util.StorageLinkedList` field (from the Takamaka library)