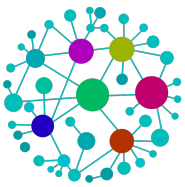
# PROGRAMMING HOTMOKA

LEARN HOW TO PROGRAM HOTMOKA NODES
WITH SMART CONTRACTS IN PURE JAVA BY


## FAUSTO SPOTO

ASSOCIATE PROFESSOR
DEPARTMENT OF COMPUTER SCIENCE
UNIVERSITY OF VERONA, ITALY

# Contents

Chapter 1

# Introduction

More than a decade ago, Bitcoin [Nakamoto08] swept the computer industry as a revolution, providing, for the first time, a reliable technology for building trust over an inherently untrusted computing infrastructure, such as a distributed network of computers. Trust immediately translated into money and Bitcoin became an investment target, exactly at the moment of one of the worst economical turmoil of recent times. Central(-*ized*) banks, fighting against the crisis, looked like dinosaurs in comparison to the *decentralized* nature of Bitcoin.

Nevertheless, the novelty of Bitcoin was mainly related to its *consensus* mechanism based on a *proof of work*, while the programmability of Bitcoin transactions was limited due to the use of a non-Turing-equivalent scripting bytecode [Antonopoulos17].

The next step was hence the use of a Turing-equivalent programming language (up to *gas limits*) over an abstract store of key/value pairs, that can be efficiently kept in a Merkle-Patricia trie. That was Ethereum [AntonopoulosW19], whose Solidity programming language allows one to code any form of *smart contract*, that is, code that becomes an agreement between parties, thanks to the underlying consensus enforced by the blockchain.

Solidity looks familiar to most programmers. Conditionals, loops and structures are there since more than half a century. Programmers assumed that they *knew* Solidity. However, the intricacies of its semantics made learning Solidity harder than expected. Finding good Solidity programmers is still difficult and they are consequently expensive. It is, instead, way too easy to write buggy code in Solidity, that *seems* to work perfectly, up to *that* day when things go wrong, very wrong [AtzeiBC17].

It is ungenerous to blame Solidity for all recent attacks to smart contracts in blockchain. That mainly happened because of the same success of Solidity, that made it the natural target of the attacks. Moreover, once the Pandora's box of Turing equivalence has been opened, you cannot expect anymore to keep the devils at bay, that is, to be able to decide and understand, exactly, what your code will do at run time. And this holds for every programming language, past, present or future.

I must confess that my first encounter with Solidity was a source of frustration. Why was I expected to learn another programming language? and another development environment? and another testing framework? Why was I expected to write code without a support library that

provides proved solutions to frequent problems? What was so special with Solidity after all? Things became even more difficult when I tried to understand the semantics of the language. After twenty-five years of studying and teaching programming languages, compilation, semantics and code analysis (or, possibly, just because of that) I still cannot explain exactly why there are structures and contracts instead of a single composition mechanism in Solidity; nor what is indeed the meaning of `memory` and `storage` and why it is not the compiler that takes care of such gritty details; nor why externally owned accounts are not just a special kind of contracts; nor why Solidity needs such low-level (and uncontrollable) call instructions, that make Java's (horrible) reflection, in comparison, look like a monument to clarity; nor why types are weak in Solidity, so that contracts are held in `address` variables, whose actual type is unknown and cannot be easily enforced at run time [CrafaPZ19], with all consequent programming monsters, such as unchecked casts. It seems that the evolution of programming languages has brought us back to C's `void*` type.

Hence, when I first met people from Ailia SA in fall 2018, I was not surprised to realize that they were looking for a new way of programming smart contracts over the new blockchain that they were developing. I must thank them and our useful discussions, that pushed me to dive in blockchain technology and study many programming languages for smart contracts. The result is Takamaka, a Java framework for writing smart contracts. This means that it allows programmers to use a subset of Java for writing code that can be installed and run in blockchain. Programmers will not have to deal with the storage of objects in blockchain: this is completely transparent to them. This makes Takamaka completely different from other attempts at using Java for writing smart contracts, where programmers must use explicit method calls to persist data to blockchain.

Writing smart contracts in Java entails that programmers do not have to learn yet another programming language. Moreover, they can use a well-understood and stable development platform, together with all its modern tools. Programmers can use features from the latest versions of Java, such as streams and lambda expressions. There are, of course, limitations to the kind of code that can be run inside a blockchain. The most important limitation is that programmers can only call a portion of the huge Java library, whose behavior is deterministic and whose methods are guaranteed to terminate.

The runtime of the Takamaka programming language is included in the Hotmoka project, a framework for collaborating nodes, whose long-term goal is to unify the programming model of blockchain and internet of things. The more scientific aspects of Hotmoka and Takamaka have been published in the last years BeniniGMS21[CrosaraOST21][OlivieriST21][Spoto19][Spoto20].

**Intended Audience**. This book is for software developers who want to use Hotmoka nodes and program smart contracts in Takamaka. It goes deep into the inner working of Hotmoka. For instance, it shows how transactions can be triggered in code, not just with the Moka client of Hotmoka. Less experienced readers, or developers not interested in writing code that interacts with Hotmoka nodes, can just skip these parts and concentrate on the use of the Moka command-line client only. Non-technical users might just be happy with the use of Mokito and Hotwallet, the mobile and web clients of Hotmoka, whose functionalities are of course limited.

**Contribute to Hotmoka**. Hotmoka is a complex project, that requires many and different skills. After years of development, it is ready for the general public. This does not mean that it is bug-free, nor perfect: we expect our users to find all sort of bugs and to suggest improvements. Hence, feel free to write to us at `info@hotmoka.io`, with bugs and improvement requests. If you are a developer, consider the possibility of helping us with the development of the project. In particular, the whole ecosystem of applications running over Hotmoka is missing at the moment

(that is, applications, typically web-based, that use Hotmoka as their backend storage).

**The Example Projects of This Book**. The experiments that we will perform in this book will require one to create Java projects inside a directory that we will name `hotmoka_tutorial`. We suggest that you create and experiment with these projects yourself. However, if you have no time and want to jump immediately to the result, or if you want to compare your work with the expected result, we provide you with the completed examples of this book in a repository that you can clone. Each section of this book will report the project of the repository where you can find the related code. You can clone that completed tutorial examples repository as follows:

```
$ git clone https://github.com/Hotmoka/hotmoka_tutorial.git
```

This will create the `hotmoka_tutorial` directory. Inside that directory, you will find the Java projects of this book, in Maven format. You can import all these projects into Eclipse (File → Import; then specify *Existing Maven Projects* and finally select the `hotmoka_tutorial` directory). They can be imported similarly in IntelliJ and NetBeans.

**Acknowledgments**. I thank the people at Ailia SA, in particular Giovanni Antino, Mario Carlini, Iris Dimni and Francesco Pasetto, who decided to invest in the Takamaka project and who are building their own open-source blockchain that can be programmed in Takamaka. My thank goes also to all students and colleagues who have read and proof-checked this book and its examples, finding bugs and inconsistencies; in particular to Luca Olivieri and Fabio Tagliaferro. Chapter Hotmoka Nodes is a shared work with Dinu Berinde. Chapter Tokens is a shared work with Marco Crosara, Filippo Fantinato, Luca Olivieri and Fabio Tagliaferro.

*Verona, January 2022.*

Chapter 2

# Getting Started with Hotmoka

## Hotmoka in a Nutshell

Hotmoka is the abstract definition of a device that can store objects (data structures) in its persistent memory (its *state*) and can execute, on those objects, code written in a subset of Java called Takamaka. Such a device is called a *Hotmoka node* and such programs are known as *smart contracts*, taking that terminology from programs that run inside a blockchain. It is well true that Hotmoka nodes can be different from the nodes of a blockchain (for instance, they can be an Internet of Things device); however, the most prominent application of Hotmoka nodes is, at the moment, the construction of blockchains whose nodes are Hotmoka nodes.

Every Hotmoka node has its own persistent state, that contains code and objects. Since Hotmoka nodes are made for running Java code, the code inside their state is kept in the standard jar format used by Java, while objects are just a collection of values for their fields, with a class tag that identifies whose class they belong too and a reference (the *classpath*) to the jar where that class is defined. While a device of a Internet of Thing network is the sole responsible for its own state, things are different if a Hotmoka node that is part of a blockchain. There, the state is synchronized and identical across all nodes of the blockchain.

In object-oriented programming, the units of code that can be run on an object are called *methods*. When a method must be run on an object, that object is identified as the *receiver* of the execution of the method. The same happens in Hotmoka. That is, when one wants to run a method on an object, that object must have been already allocated in the state of the node and must be marked as the receiver of the execution of the method. Assume for instance that one wants to run a method on the object in Figure 1, identified as receiver. The code of the method is contained in a jar, previously installed in the state of the node, and referred as *classpath*. This is the jar where the class of the receiver is defined.

The main difference with standard object-oriented programming is that Hotmoka requires one to specify a further object, called *payer*. This is because a Hotmoka node is a public service, that can be used by everyone has an internet connection that can reach the node. Therefore, that service must be paid with the internal cryptocurrency of the node, by providing a measure of execution effort known as *gas*. The payer is therefore a sort of bank account, whose balance

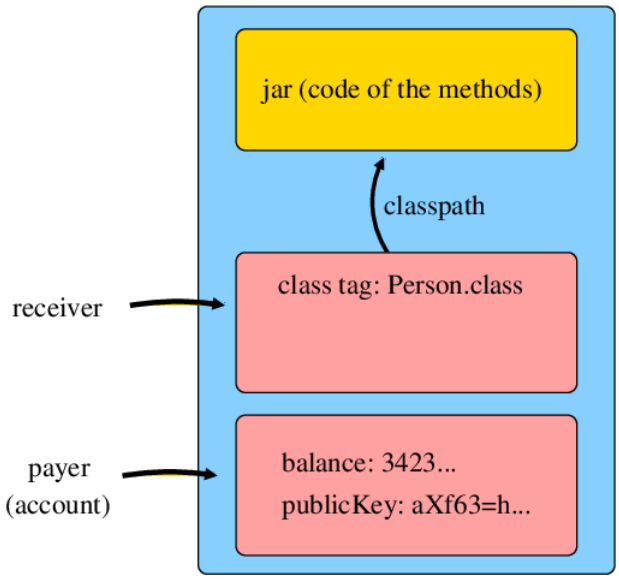



*Figure 1. Receiver, payer and classpath for a method call in a Hotmoka node.*

gets decreased in order to pay for the gas needed for the execution of the method. The payer is accessible inside the method as its *caller*.

> There are many similarities with what happens in Ethereum: the notion of receiver, payer and gas are taken from there. There are, however, also big differences. The first is that the code of the methods is inside a jar *referenced* by the objects, while Ethereum requires to reinstall the code of the contracts each time a contract is instantiated. More importantly, Hotmoka keeps an explicit class tag inside the objects, while contracts are untyped in Ethereum [CrafaPZ19] and are referenced through the untyped `address` type.

Receiver and payer have different roles but are treated identically in Hotmoka: they are objects stored in state at their respective state locations, known as their *storage references*. For instance the caller in Figure 1 might be allocated at the storage reference `75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0`. A storage reference has two parts, separated by a `#` sign. The first part are 64 hexadecimal digits (ie, 32 bytes) that identify the transaction that created the object; the second part is a progressive number that identifies an object created during that transaction: the first object created during the transaction has progressive zero, the second has progressive one, and so on. When a method is called on a Hotmoka node, what is actually specified in the call request are the storage references of the receiver and of the payer (plus the actual arguments to the method, if any).

In Hotmoka, a *transaction* is either

1. the installation of a jar, that modifies the state of the node, and is paid by a payer account, or
2. the execution of a method on a receiver, that yields a returned value and/or has side-effects that modify the state of the node, and is paid by a payer account.

A Hotmoka node can keep track of the transactions that it has executed, so that it is possible,

for instance, to recreate its state by running all the transactions executed in the past, starting from the empty state.

It is very important to discuss at this moment a significant difference with what happens in Bitcoin, Ethereum and most other blockchains. There, an account is not an object, nor a contract, but just a key in the key/value store of the blockchain, whose value is its balance. The key used for an account is typically computed by hashing the public key derived from the private key of the account. In some sense, accounts, in those blockchain, exist independently from the state of the blockchain and can be computed offline: just create a random private key, compute the associated public key and hence its hash. Hotmoka is radically different: an account is an object that must be allocated in state by an explicit transaction (that must be paid, as every transaction). The public key is explicitly stored inside the object (Base64-encoded in its `publicKey` field, see Figure 1). That public key was passed as a parameter for the creation of the payer object and can be passed again for creating more accounts. That is, it is well possible, in Hotmoka, to have more accounts in the state of a node, all distinct, but controlled by the same key.

This has a major consequence. In Bitcoin and Ethereum, an account is identified by twelve words and a password, by using the BIP39 encoding (see Figure 5-6 of [Antonopoulos17]). These twelve words are just a mnemonic representation of 132 bits: 128 bits for the random entropy used to derive the private key of the account and four bits of checksum. In Hotmoka, these 128 bits are not enough, since they identify the key of the account but not the 32 bytes of its storage reference (in this representation, the progressive is assumed to be zero). As a consequence, accounts in Hotmoka are identified by 128+256 bits, plus 12 bits of checksum (and a password), which give rise to 36 words with the BIP39 encoding. By specifying those 36 words across different clients, one can control the same account with all such clients. As usual, those 36 words must be stored in paper and kept in a secure place, since losing them amounts to losing access to the account.

As shown in Figure 1, the code of the objects (contracts) installed in the state of a Hotmoka node consists in jars (Java archives) written in a subset of Java known as Takamaka. This is done in a way completely different from other blockchains:

1. In Hotmoka, programmers code the contracts that want to install in the node and nothing more; they do *not* program the encoding of data into the key/value store of the node (its *keeper*, as it is called in other blockchains); they do *not* program the gas metering; they do *not* program the authentication of the accounts and the verification of their credentials. Everything is automatic in Hotmoka, exactly as in Ethereum, and differently from other blockchains that use general purpose languages such as Java for their smart contracts: there, programmers must take care of all these details, which is difficult, boring and error-prone. If this is done incorrectly, those blockchains will hang.
2. In Hotmoka, the code installed in the node passes a preliminary verification, that checks the correct use of some primitives, that we will introduce in the subsequent chapters, and guarantees that the code is deterministic. This excludes an array of errors in Hotmoka, while other blockchains will hang if, for instance, the code is non-deterministic.

## Hotmoka Clients

In order to query a Hotmoka node, handle accounts and run transactions on the node, one needs a client application. Currently, there are a command-line client, called Moka, a mobile client for Android, called Mokito, and a web client that gets installed in the browser (Chrome or Firefox), called Hotwallet. Mokito and Hotwallet provide basic functionalities only (handling accounts,

querying the state of the objects in the node, running simple transactions), while Moka is the most complete solution, but also the most difficult to use.

## Moka

You can use the `moka` tool to interact with a Hotmoka node, install code in the node and run transactions in the node. The latest version of the tool can be downloaded from https://github.com/Hotmoka/hotmoka/releases. Its source code is maintained inside the main distribution of the Hotmoka project, at https://github.com/Hotmoka/hotmoka, in the submodule `io-hotmoka-tools`. We report below the installation instructions of `moka`. In order to run the tool, you need Java JDK version 11 (or higher) installed in your computer and a recent version of Maven.

### Linux and MacOS

You should download and untar the latest release into the directory where you want to install `moka`. For instance, assuming that the latest version is `1.0.7` and that you want to install it under `~/Opt/moka`, you can run the following commands:

```
$ cd ~/Opt
$ mkdir moka
$ cd moka
$ wget https://github.com/Hotmoka/hotmoka/releases/
     download/v1.0.7/moka_1.0.7.tar.gz
$ tar zxf moka_1.0.7.tar.gz
$ export PATH=$PATH:$(pwd)
```

The dollar sign is the prompt of the shell. In the shell scripts reported in this book, whenever a line is too long, as that starting with wget above, we go to the next line but you should enter the command in a single line, without newlines or spaces: `...releases/download/v...`

The last `export` command expands the command-path of the shell with the `~/Opt/moka` directory, so that `moka` can be invoked from the command shell, regardless of the current directory. You might want to add an `export` at the end of your `~/.bashrc` configuration file, so that the command-path will be expanded correctly the next time you open a shell. For instance, I added the following command at the end of my `~/.bashrc`:

```
export PATH=/home/spoto/Opt/moka:$PATH
```

### Windows

You should download and untar the latest release (https://github.com/Hotmoka/hotmoka/releases) into the directory where you want to install `moka`, by using a software tool such as `7zip` or `Cygwin`. After that, you should be able to run the tool from the command prompt:

```
$ cd directory-where-you-installed-moka
$ moka.bat help
```

In the following of this tutorial, remember to use `moka.bat` to invoke the tool, where our examples use `moka` instead, which is the Linux name of the invocation script.

You might want to add, to the command-path, the directory where you installed `moka`, so that it will be expanded correctly the next time you open a command prompt window. For that, add that directory to the `PATH` environment variable.

### First Usage of `moka`

The `moka` tool should be in the command-path of your shell now. You can check that it works, by invoking `moka` as follows:

```
$ moka help
Usage: moka [COMMAND]
This is version 1.0.7 of the Hotmoka command-line interface.
Commands:
  bind-key            Binds a key to a reference, so that it becomes an account
  burn                Burns coins from an account, if the node allows it
  call                Calls a method of an object or class
  create              Creates an object in the store of a node
  create-account      Creates a new account
  create-key          Creates a new key
  help                Displays help information about the specified command
  faucet              Sets the thresholds for the faucet of the gamete of a node
  import-account      Imports an account
  info                Prints information about a node
  init-memory         Initializes a new Hotmoka node in memory
  init-tendermint     Initializes a new Hotmoka node based on Tendermint
  install             Installs a jar in a node
  instrument          Instruments a jar
  mint                Mints new coins for an account, if the node allows it
  resume-tendermint   Resumes an existing Hotmoka node based on Tendermint
  send                Sends units of coin to a payable contract
  show-account        Shows an account
  start-tendermint    Starts a new Hotmoka node based on Tendermint
  state               Prints the state of an object
  verify              Verifies a jar
```

As you can see above, the `moka help` command prints a description of the available subcommands and exits. You can have a detailed help of a specific subcommand by specifying the subcommand after `help`. For instance, to print the help of the `faucet` subcommand, you can type:

```
$ moka help faucet
Usage: moka faucet [--interactive] [--max-red=<maxRed>]
                   [--password-of-gamete=<passwordOfGamete>] [--url=<url>] <max>
Sets the thresholds for the faucet of the gamete of a node
      <max>                the maximal amount of coins sent at each call to the
                             faucet of the node
                             Default: 0
      --interactive        run in interactive mode
      --max-red=<maxRed>   the maximal amount of red coins sent at each call to
                             the faucet of the node
                             Default: 0
      --password-of-gamete=<passwordOfGamete>
                           the password of the gamete account; if not
                             specified, it will be asked interactively
      --url=<url>          the url of the node (without the protocol)
                             Default: localhost:8080
```

## Mokito

The `moka` tool allows one to perform a large variety of operations on a Hotmoka node. However, it is a technical tool, meant for developers. Most users will only perform simple tasks with a Hotmoka node. For them, it is simpler to use a mobile app, with a simpler user interface. That app, called `Mokito`, is currently available for Android only. You can download it from Google Play and install it in your device, from https://play.google.com/store/apps/details?id=io.hotmoka.android.mokito. Developers interested in its Kotlin source code can find it at https://github.com/Hotmoka/HotmokaAndroid, together with a small Android service for connecting to a remote Hotmoka node.

The first time you will use Mokito on your mobile device, it will connect by default to our testnet Hotmoka node and show the screen in Figure 2. This can be changed in the preferences section of the app, accessible through the menu in the top left area of the app.

## Hotwallet

Hotwallet is a web client that can be installed as an extension to Chrome and Firefox. It performs basic account handling and includes a JavaScript/TypeScript interface that can be queried by web applications that need to access the user's accounts. This architecture has been inspired by MetaMask for Ethereum. You can install Hotwallet inside the Firefox browser as an add-on that can be downloaded from https://addons.mozilla.org/en-US/firefox/addon/hotwallet. Developers interested in its TypeScript source code can find it at https://github.com/Hotmoka/hotwallet-browser. There is also a Hotweb3 JavaScript/TypeScript library for connecting web applications to the Hotwallet browser extension and use its accounts programmatically. Its source code can be found at https://github.com/Hotmoka/hotweb3.

The first time you run Hotwallet, it will ask you to create a key or import an account (see Figure 3). By default, Hootwallet will contact our test node at `panarea.hotmoka.io`, but this can be changed with the selector in the topmost area of the window.

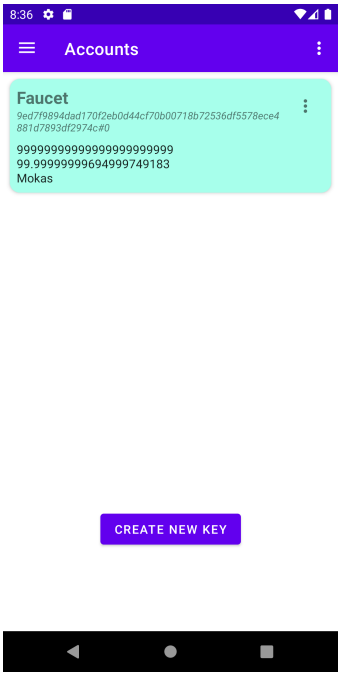



*Figure 2. The starting screen of the Mokito app.*

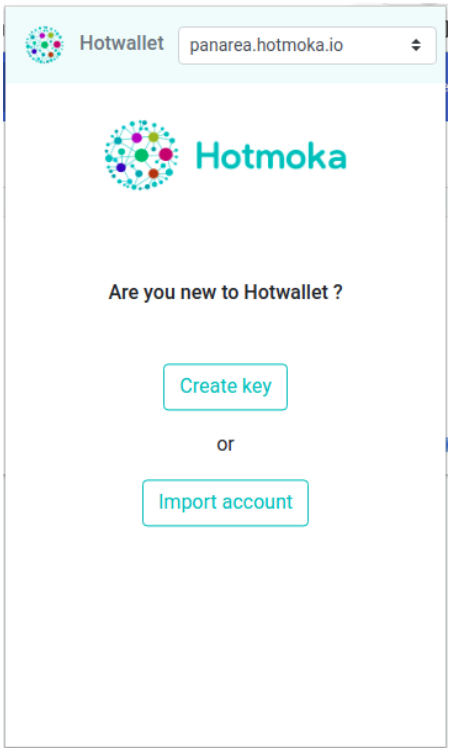



*Figure 3. The starting screen of the Hotwallet extension to the browser.*

## Contacting a Hotmoka Test Node

The examples in this book must be run by a Hotmoka node, typically part of a Hotmoka blockchain. We will show you in a later chapter how you can install your own local node or blockchain. However, for now, it is much simpler to experiment with a node that is part of a public test blockchain that we provide for experimentation. Namely, we have installed a Hotmoka node at the address `panarea.hotmoka.io`. You can verify that you can contact that node by typing the command `moka info` to print information about the node at that address, as you can see below:

```
$ moka info --url panarea.hotmoka.io
Info about the node:
  takamakaCode: 5fd6ae9fe7dbd499621f56814c1f6f1e30718ca9aea69b427dee8c16b9f6c665
  manifest: 188c6c032ca1f4f559e1cd2d3e044ba81e08b6a01934fc12ef0657cb8636c7a8#0
    chainId: marabunta
    maxErrorLength: 300
    signature: ed25519
    ...
    gamete: 5aeca15b70978d3aa4973f2611a775cf9db13c2391f03e7ab2593fe010e31cd5#0
      balance: 99999999999999999999...
      maxFaucet: 10000000000000
      ...
    gasStation: 188c6c032ca1f4f559e1cd2d3e044ba81e08b6a01934fc12ef0657cb8636c7a8#11
      gasPrice: 1
      ...
    validators: 188c6c032ca1f4f559e1cd2d3e044ba81e08b6a01934fc12ef0657cb8636c7a8#2
      totalSupply: 1000000000000000...
      ...
```

The details of this information are irrelevant for now, but something must be clarified, to understand the following sections better. Namely, the `moka info` information is telling us that the node already contains some code and Java objects, as shown in Figure 4.

The `takamakaCode` reference is the pointer to a jar, installed in blockchain, that contains the classes of the Takamaka language runtime. All programs that we will write in Takamaka will depend on that jar, since they will use classes such as `io.takamaka.code.lang.Contract` or annotations such as `io.takamaka.code.lang.View`. The `manifest` reference, instead, points to a Java object that publishes information about the node. Namely, it reports its chain identifier and the signature algorithm that, by default, is used to sign the transactions for the node. The manifest points to another object, called `gamete`, that is an account holding all coins in blockchain, initially. Consequently, the gamete has a rich `balance`, but also another interesting property, called `maxFaucet`, that states how much coins it is willing to give up for free. In a real node, and differently from here, that value should be zero. In this test network, instead, it is a non-zero value that we will exploit for creating our first account, in a minute. The `gasStation` refers to another Java object, that provides information about the gas, such as its current `gasPrice`. Finally, there is another Java object, called `validators`, that keeps information about the validator nodes of the network and contains, for instance, the total supply of the cryptocurrency (how much cryptocurrency has been minted up to now).

As we said in the previous section, Java objects in the Hotmoka node are identified by their *storage reference*, such as `188c6c032ca1f4f559e1cd2d3e044ba81e08b6a01934fc12ef0657cb8636c7a8#11`.

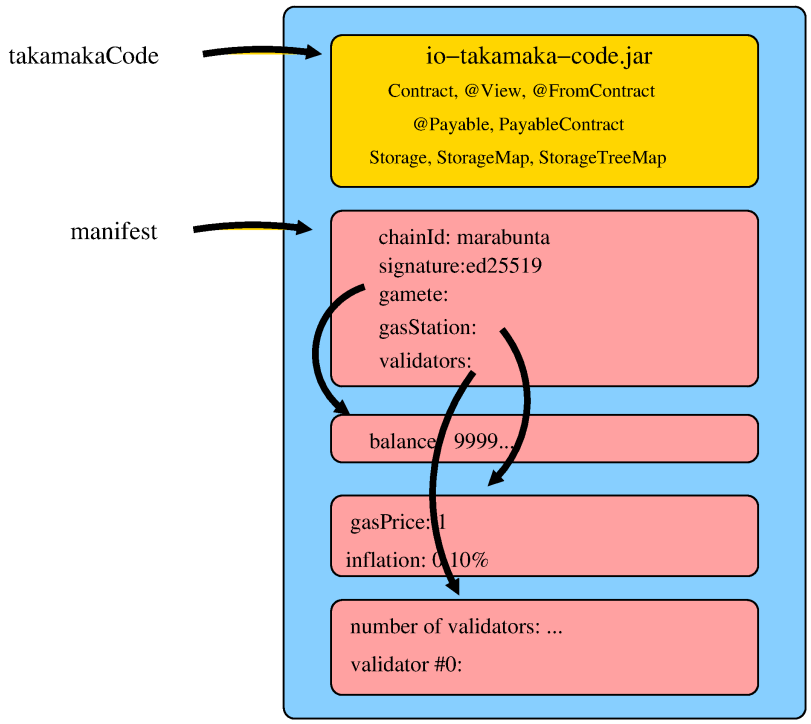



*Figure 4. The state of the test network nodes.*

You can think at a storage reference as a machine-independent pointer inside the memory, or state, of the node.

We have used the `moka` tool to see the manifest of a node. You can also use the Mokito app for that. Namely, tap on the app menu icon on the top-left corner of the screen and select *Manifest* from the menu that will appear (see Figure 5).

After tapping on *Manifest*, a new screen will appear, containing the same information that we found with `moka info` (see Figure 6).

## Creation of a First Account

We need an account in the test network, that we will use later to pay for installing code in blockchain and for running transactions. An account in Hotmoka is something completely different from an account in other blockchains. For instance, in Bitcoin and Ethereum accounts do not really exist, in the sense that they are just an address derived from a private key, that can be used to control information in blockchain. Their creation does not trigger any operation in blockchain, it is performed completely off-chain. Instead, in Hotmoka, an account is a Java object, more precisely an instance of the `io.takamaka.code.lang.ExternallyOwnedAccount` class. That object must be allocated (*created*) in the memory of the Hotmoka node, before it can be used. Moreover, such an object is not special in any way: for instance, as for all other objects in the storage of the node, we must pay for its creation. Currently, we have no accounts and consequently no coins for paying the creation of a new object. In a real blockchain, we could

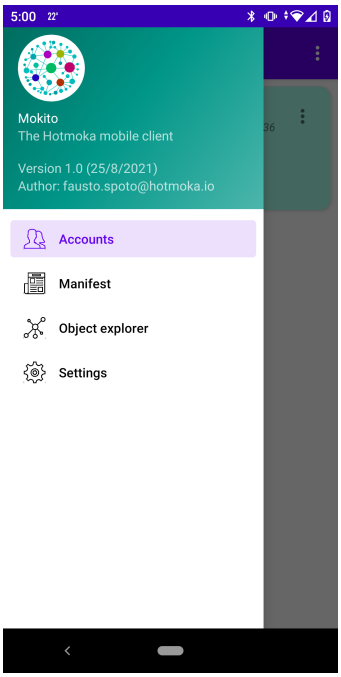



*Figure 5. The menu of the Mokito app.*

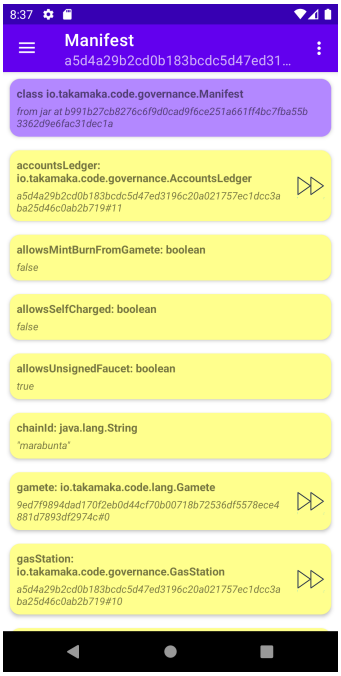



*Figure 6. The manifest of the Hotmoka node, shown in the Mokito app.*

earn coins by working for the network, or as payment for some activity, or by buying coins at an exchange. In this test network, we will use the faucet of the gamete instead, that is willing to send us up to 10000000000000 coins, for free. Namely, you can run the following command in order to ask the faucet to create your first externally owned account, funded with 50000000000 coins, initially. We will execute the following command-line inside the `hotmoka_tutorial` directory, so that it will save the *entropy* of your account there, which will simplify your subsequent work:

```
$ moka create-account 50000000000 --payer faucet --url panarea.hotmoka.io

Please specify the password of the new account: chocolate
Free account creation succeeds only if the gamete supports an open unsigned faucet.
Created account 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0.
Its entropy has been saved into the file
  "75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0.pem".
Please take note of the following passphrase of 36 words:

 1: cage
 2: faint
 3: act
 4: snake
 5: stairs
 6: derive
 7: giraffe
 8: glance
 9: before
10: merry
11: sea
12: decline
13: foot
14: six
15: boost
16: else
17: fix
18: theory
19: post
20: ring
21: private
22: output
23: capable
24: camp
25: daughter
26: dad
27: vault
28: rebuild
29: knife
30: unaware
31: dinner
32: virus
33: device
34: bone
35: way
36: regret
```

A *storage reference* `75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0` has been created.

> Note that this reference will be different in your machine, as well as the 36 words passphrase. Change these accordingly in the subsequent examples.

This storage reference is a machine-independent pointer to your account Java object, inside the node. Moreover, a random sequence of bits, called *entropy*, has been generated and saved into a `.pem` file. From that entropy, and the chosen password, it is possible to derive private and (hence) public key of the account. You should keep the `.pem` file secret since, together with the password of the account (in our case, we chose `chocolate`), it allows its owner to control your account and spend its coins. Note that the password is not written anywhere: if you lose it, there is no way to recover that password.

We will see in a moment what is the use of these 36 words passphrase generated by `moka`. For now, let us check that our account really exists at its address, by querying the node with the `moka state` command:

```
$ moka state 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
  --url panarea.hotmoka.io

This is the state of object
75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
@panarea.hotmoka.io

class io.takamaka.code.lang.ExternallyOwnedAccount
  (from jar installed at
   5fd6ae9fe7dbd499621f56814c1f6f1e30718ca9aea69b427dee8c16b9f6c665)

  nonce:java.math.BigInteger = 0
  publicKey:java.lang.String = "BheU05MT/MGmeytPvrdW+Kggj965oh4SQ6seyOoTw1c="
  balance:java.math.BigInteger =
    50000000000 (inherited from io.takamaka.code.lang.Contract)
  balanceRed:java.math.BigInteger =
    0 (inherited from io.takamaka.code.lang.Contract)
```

Note that the balance and the public key of the account are fields of the account object. Moreover, note that Hotmoka knows which is the class of the object at that address (it is a `io.takamaka.code.lang.ExternallyOwnedAccount`) and where that class is defined (inside the jar at address `5fd6ae9fe7dbd499621f56814c1f6f1e30718ca9aea69b427dee8c16b9f6c665`, that is, `takamakaCode`).

> This is completely different from what happens, for instance, in Ethereum, where externally owned accounts and contract references and untyped at run time, so that it is not possible to reconstruct their class in a reliable way. Moreover, note that we have been able to create an object in blockchain without sending the bytecode for its code: that bytecode was already installed, at `takamakaCode`, and we do not need to repeat it every time we instantiate an object. Instead, the new object will refer to the jar that contains its bytecode.

In the following, you can use the `moka state` command on any object, not just on your own account, whenever you want to inspect its state (that includes the state inherited from its superclasses).

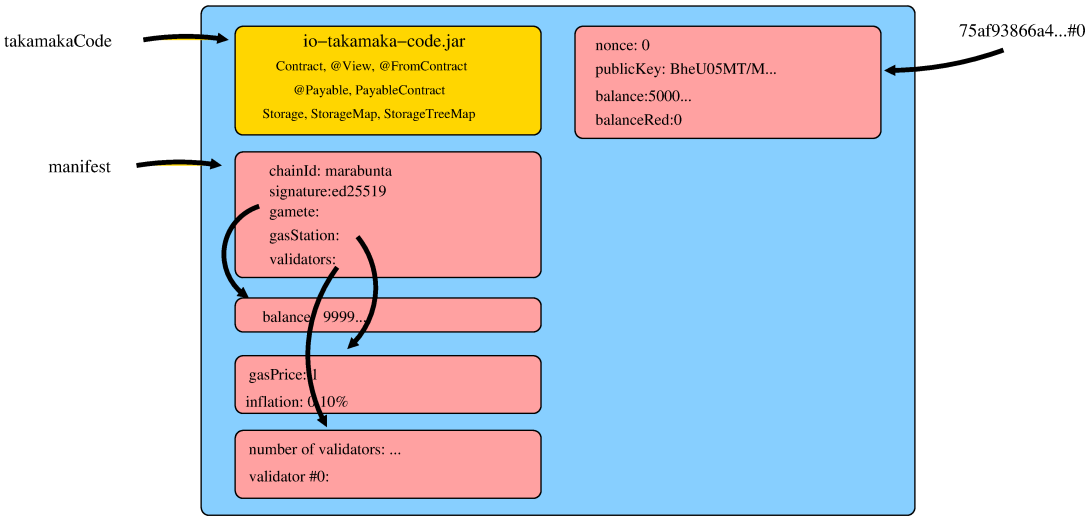



*Figure 7. The state of the test network nodes after the creation of our new account.*

Figure 7 shows the state of the network nodes after the creation of our new account. Since out test node is part of a blockchain, it is not only its state that has been modified, but also that of all nodes that are part of the blockchain.

Whenever your account will run out of coins, you can recharge it with the `moka send` command, using, again, the faucet as source of coins. Namely, if you want to recharge your account with 200000 extra coins, you can type:

```
$ moka send 200000
    75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --payer faucet --url panarea.hotmoka.io
```

You can then use the `moka state` command to verify that the balance of your account has been actually increased with 200000 extra coins.

The creation of a new account from the faucet is possible from the Mokito app as well. Namely, use the menu of the app to tap on the *Accounts* item to see the list of available accounts (Figure 2). From there, tap on the menu icon on the right of the *Faucet* account and select *Create a new account* (see Figure 8).

A form will appear, where you can specify the name for the account, its password and the initial balance (that will be paid by the faucet). For instance, you can fill it as in Figure 9.

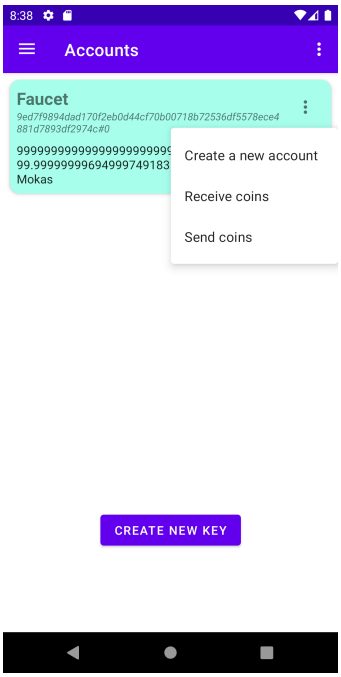



*Figure 8. The menu for creating a new account with Mokito.*

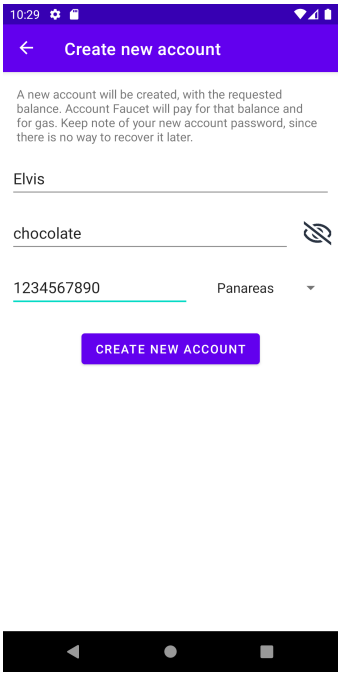



*Figure 9. The form specifying a new account Elvis.*

The name of the accounts is a feature of Mokito to simplify the identification of the accounts. However, keep in mind that accounts have no name in Hotmoka: they are just identified by their storage reference. For instance, `moka` currently does not allow one to associate names to accounts.

After tapping on the `Create new account` button, the new account will be created and its information will be shown, as in Figure 10. Again, note in this screen the storage reference of the new account and the presence of a 36 words passphrase.

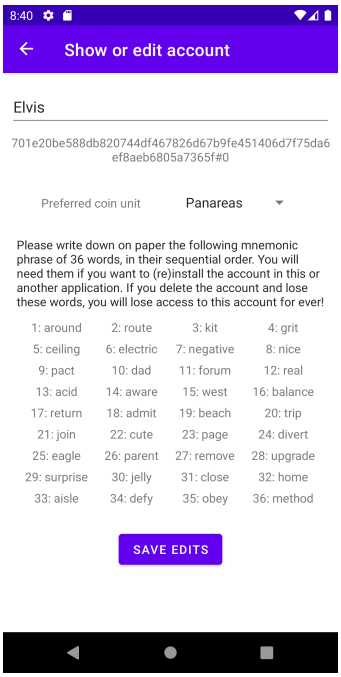



*Figure 10. The new account Elvis.*

If you go back to the accounts screen (by using the top-left menu of Mokito), you will see that Elvis has been added to your accounts (see Figure 11).

## Importing Accounts

We have created `75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0` with `moka` and `701e20be588db820744df467826d67b9fe451406d7f75da6ef8aeb6805a7365f#0` with Mokito. We might want to *import* the former in Mokito and the latter in `moka`, and we want to import both inside Hotwallet, so that we can operate on both accounts with all three tools. In order to import `701e20be588db820744df467826d67b9fe451406d7f75da6ef8aeb6805a7365f#0` in `moka`, we can use the `moka import-account` command and insert its 36 words passphrase:

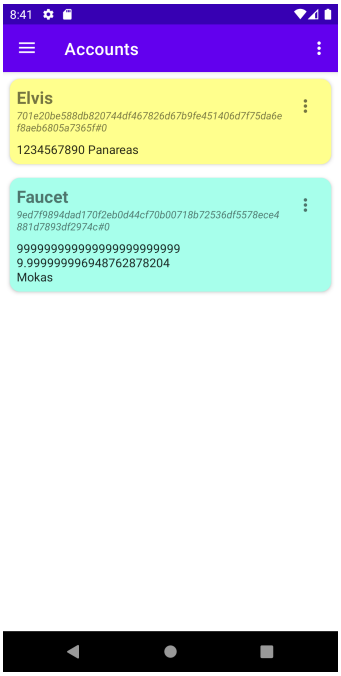



*Figure 11. The new account Elvis has been added.*

```
$ moka import-account

Insert the 36 words of the passphrase of the account to import:
word #1: rail
word #2: double
word #3: bag
word #4: dove
word #5: fluid
...
word #34: bounce
word #35: deposit
word #36: hotel
The account 701e20be588db820744df467826d67b9fe451406d7f75da6ef8aeb6805a7365f#0
  has been imported.
Its entropy has been saved into the file
  "701e20be588db820744df467826d67b9fe451406d7f75da6ef8aeb6805a7365f#0.pem".
```

From this moment, it is possible to control that account with `moka` (if we remember its password, that is, `chocolate`).

Vice versa, with Mokito, go to the accounts page, show its top-right menu and select *Import account* (see Figure 12). In the screen that will appear, insert the name that you want to give to the account, its password and its 36 words passphrase (Figure 13). Tap on the *Import Account* button. The new account will show in the list of available accounts (Figure 14). From this moment, it will be possible to control the account from Mokito.

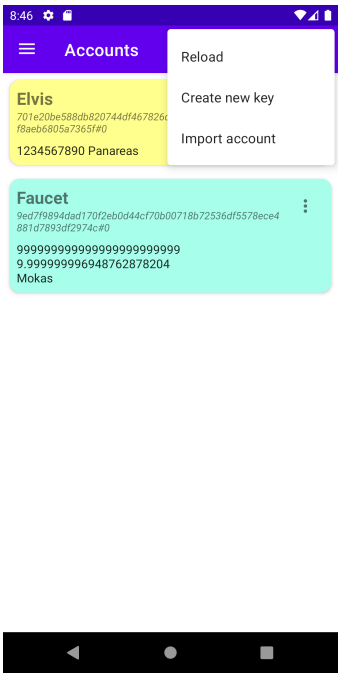



*Figure 12. The menu of the accounts screen.*

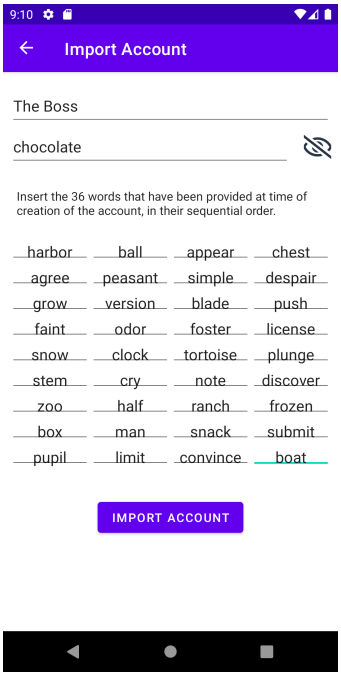



*Figure 13. Inserting the 36 words passphrase in Mokito.*

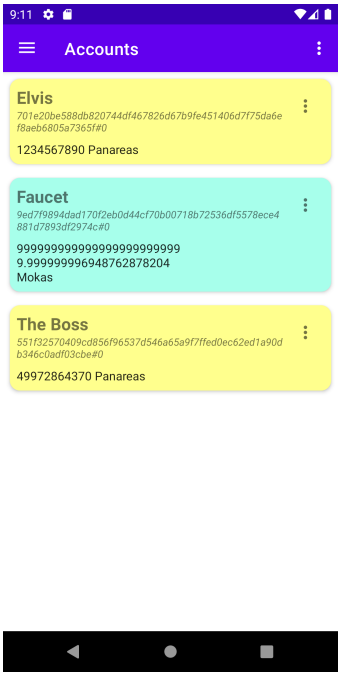



*Figure 14. The new account The Boss has been added.*

In Hotwallet, click on the top-right menu ans select *Import account*. From there, you can insert the password and the 36 words passphrase of each account and see them imported in Hotwallet. You can see the list of all available accounts with the same menu, if you click on *Account list*. You can click on each account and switch from one to the other. Every time you change the account, Hotwallet requires to insert its password.

As you have seen, the 36 words and the password of an account are enough for moving accounts around different clients. Note that clients do not *contain* accounts but only the cryptographic information needed to access the accounts. If a client is uninstalled, the accounts that it used still exist in the remote Hotmoka node and can still be re-imported and used in some other client, if we have written down their 36 words and still remember their passwords.

## Anonymous Payments

The fact that accounts in Hotmoka are not just identified by their public key, but also by their storage reference inside the state of a node, makes it a bit more difficult, but not impossible, to execute anonymous transactions. We do not advocate the use of anonymicity here, but it is true that, sometimes, one wants to remain anonymous and still receive a payment.

> Anonymicity is often used for illegal actions such as ramsonware and blackmailing. We are against such actions. This section simply shows that anonymicity can be achieved in Hotmoka as well, although it is a bit harder than with other blockchains.

Suppose for instance that somebody, whom we call Anonymous, wants to receive from us a

payment of 10,000 coins, but still wants to remain unknown. He can receive the payment in many ways:

1. He could send us an anonymous email asking us to pay to a specific account, already existing in the state of the node. But this is not anonymous, since, in Hotmoka, an account is an object and there must have been a transaction that created that object, whose payer is likely to be Anonymous or somebody in his clique. That is, the identity of Anonymous can be inferred from the creator of the account. Therefore, Anonymous discards this possibility.
2. He could send us an anonymous email asking us to create a new account with a given public key, whose associate private key he controls, and to charge it with 10,000 coins. After that, we are expected to send him an email where we notify him the storage reference where `moka create-account` has allocated the account. But this means that we must know his email address, which is definitely against the idea of anonymicity. Therefore, Anonymous discards this possibility as well.
3. He could send us an anonymous email asking us *to pay to a given public key*, whose associated private key he controls. After we pay to that key, he autonomously and anonymously recovers the storage reference of the resulting account, without any interaction with us. This is definitely anoymous and that is the technique that Anonymous will choose.

Let us show how the third possibility works. Anonymous starts by creating a new private/public key, exactly as one would do in Ethereum and other blockchains. He runs the following command:

```
$ moka create-key

Please specify the password of the new key: kiwis
A new key HhKzZWgc6Fad6J1dxx1seEuJZB9m4JhwEbti1VBW52Nr has been created.
Its entropy has been saved into the file
  "./HhKzZWgc6Fad6J1dxx1seEuJZB9m4JhwEbti1VBW52Nr.pem".
```

Note that there is no `--url` part in the `moka create-key` command, since this operation runs completely off-line: no object gets created in the state of any Hotmoka node for now. Anonymous pastes the new key into an anonymous email message to us:

```
Please pay 10000 coins to the key HhKzZWgc6Fad6J1dxx1seEuJZB9m4JhwEbti1VBW52Nr.
```

Once we receive this email, we use (for instance) our previous account to send 10000 coins to that key:

```
$ moka send 10000 HhKzZWgc6Fad6J1dxx1seEuJZB9m4JhwEbti1VBW52Nr
    --anonymous
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io

Please specify the password of the payer account: chocolate
Do you really want to spend up to 700000 gas units to send the coins [Y/N] Y
Total gas consumed: 63544
  for CPU: 985
  for RAM: 2141
  for storage: 60418
  for penalty: 0
The owner of the key can now see the new account associated to the key.
```

And that's all! No other interaction is needed with Anonymous. He will check from time to time to see if we have paid, by running the command `moka bind-key` until it succeeds:

```
$ moka bind-key HhKzZWgc6Fad6J1dxx1seEuJZB9m4JhwEbti1VBW52Nr
    --url panarea.hotmoka.io

Cannot bind: nobody has paid anonymously to the key
  HhKzZWgc6Fad6J1dxx1seEuJZB9m4JhwEbti1VBW52Nr up to now.

$ moka bind-key HhKzZWgc6Fad6J1dxx1seEuJZB9m4JhwEbti1VBW52Nr
    --url panarea.hotmoka.io

Cannot bind: nobody has paid anonymously to the key
  HhKzZWgc6Fad6J1dxx1seEuJZB9m4JhwEbti1VBW52Nr up to now.

$ moka bind-key HhKzZWgc6Fad6J1dxx1seEuJZB9m4JhwEbti1VBW52Nr
    --url panarea.hotmoka.io

A new account 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
  has been created.
Its entropy has been saved into the file
  "./75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0.pem".
```

Once `moka bind-key` succeeds, Anonymous can enjoy his brand new account, that he can control with the `kiwis` password.

So how does that work? The answer is that the `--anonymous` option to `moka send` creates the account `75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0` with the public key of Anonymous inside it, so that Anonymous will be able to control that account. But there is more: the `moka send` command will also associate that account to the key `HhKzZWgc6Fad6J1dxx1seEuJZB9m4JhwEbti1VBW52Nr` inside a hash map contained in the manifest of the node, called *accounts ledger*. The `moka bind-key` command will simply query that hash map, to see if somebody has already bound an account to `HhKzZWgc6Fad6J1dxx1seEuJZB9m4JhwEbti1VBW52Nr`.

If, inside the accounts ledger, there is an account $C$ already associated to the key `HhKzZWgc6Fad6J1dxx1seEuJZB9m4JhwEbti1VBW52Nr`, then the `moka send` command will not create a new account but will increase the balance of $C$ and the `moka bind-key` command will consequently yield $C$. This is a security measure in order to avoid payment disruptions due to the association of dummy accounts to some keys or to repeated payments to the same key. In any case, the public key of $C$ can only be `HhKzZWgc6Fad6J1dxx1seEuJZB9m4JhwEbti1VBW52Nr`, since the accounts ledger enforces that constraint when it gets populated with accounts: if somebody associates a key $K$ to an account $C$, then the public key contained inside $C$ must be $K$.

Anonymous payments are possible with Mokito and Hotwallet as well. Both clients allow one to create a key and pay to a key.

Should one use anonymous payments, always? The answer is no, since anonymicity incurs into an extra gas cost: that of modifying the accounts ledger. If there is no explicit need for anonymicity, it is cheaper to receive payments as described in points 1 and 2 considered above, probably without the need of anonymous emails.

## Installation of the Source Code

You will *not* need to download and install to source code of Hotmoka in this book. Nevertheless, an important aspect of blockchain technology is that trust is based also on the public availability of its code. Moreover, you will need to download the source code if you want to understand its inner working, or contribute to the development of the project or fork the project.

Hence, we show below how to download the source code of Hotmoka and of the runtime of the Takamaka language. You will need Java JDK version at least 11.

Clone the project with:

```
$ git clone https://github.com/Hotmoka/hotmoka.git
```

then `cd` to the `hotmoka` directory and compile, package, test and install the Hotmoka jars:

```
$ mvn clean install
```

All tests should pass and all projects should be successfully installed:

```
[INFO] Reactor Summary for Hotmoka 1.0.7:
[INFO]
[INFO] Hotmoka ..................................... SUCCESS [  0.949 s]
[INFO] io-takamaka-code ............................ SUCCESS [  2.276 s]
[INFO] io-hotmoka-constants ........................ SUCCESS [  0.090 s]
[INFO] io-hotmoka-whitelisting ..................... SUCCESS [  0.419 s]
[INFO] io-hotmoka-verification ..................... SUCCESS [  0.566 s]
[INFO] io-hotmoka-beans ............................ SUCCESS [  0.557 s]
[INFO] io-hotmoka-instrumentation .................. SUCCESS [  0.380 s]
[INFO] io-hotmoka-nodes ............................ SUCCESS [  0.147 s]
[INFO] io-hotmoka-crypto ........................... SUCCESS [  0.472 s]
...
[INFO] io-hotmoka-examples ......................... SUCCESS [  1.691 s]
[INFO] io-hotmoka-tests ............................ SUCCESS [04:02 min]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time:  04:25 min
[INFO] Finished at: 2021-09-23T18:57:52+02:00
[INFO] ------------------------------------------------------------------------
```

If you are not interested in running the tests, append `-DskipTests` after the word `install`.

If you want to edit the source code inside an IDE, you can import it in Eclipse, NetBeans or IntelliJ. In Eclipse, use the File → Import → Existing Maven Projects menu item and import the parent Maven project contained in the `hotmoka` directory that you cloned from GitHub. This should create, inside Eclipse, also its submodule projects. You should see, inside Eclipse's project explorer, something like Figure 15. You will then be able to compile, package, test and install the Hotmoka jars inside Eclipse itself, by right-clicking on the `parent` project and selecting `Run As` and then the `Mavel install` target. You will also be able to run the tests inside the Eclipse JUnit runner, by right-clicking on the `io-hotmoka-tests` subproject and selecting `Run As` and then the `JUnit Test` target.

The Maven configuration of the project specifies that all modules and their dependencies get copied into the `modules` directory as result of compilation with Maven (also from inside Eclipse), classified as automatic, explicit and unnamed modules (as from Java 9 onwards). You can see this by typing:

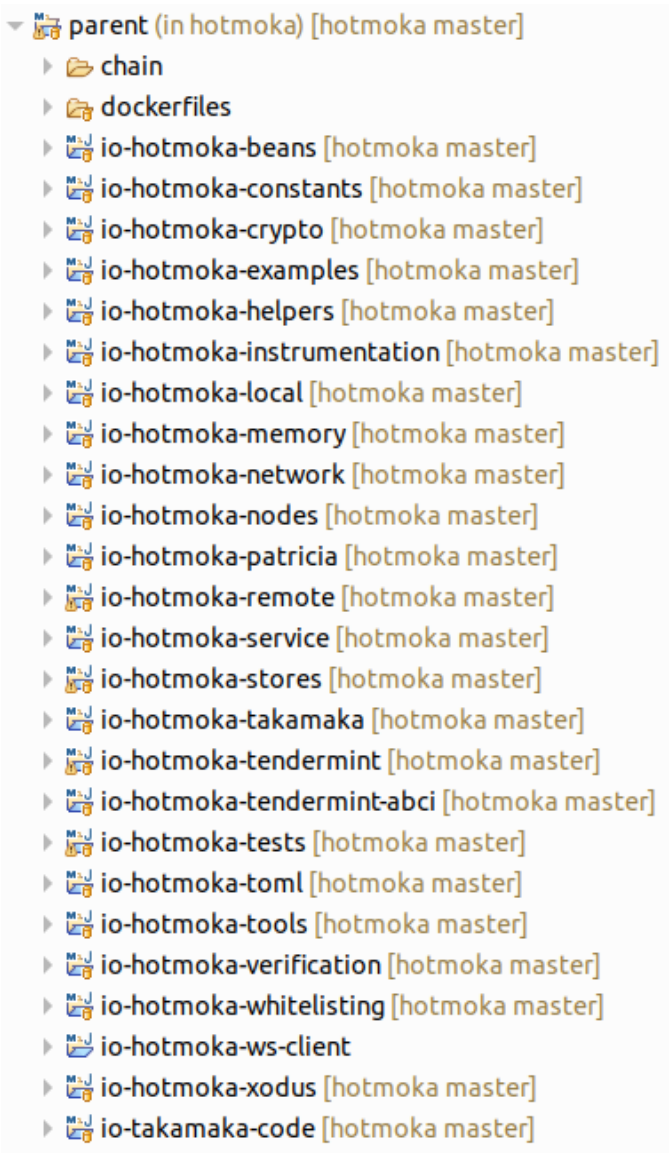



*Figure 15. The Eclipse projects of Hotmoka.*

```
$ ls -R modules

modules/:
automatic  explicit  unnamed

modules/automatic:
bcel-6.2.jar
spring-beans-5.2.7.RELEASE.jar
spring-core-5.2.7.RELEASE.jar
...
io-hotmoka-tendermint-abci-1.0.7.jar
io-hotmoka-toml-1.0.7.jar
io-hotmoka-ws-client-1.0.7.jar
io-hotmoka-xodus-1.0.7.jar
...

modules/explicit:
bcprov-jdk15on-1.70.jar
io-hotmoka-local-1.0.7.jar
io-hotmoka-verification-1.0.7.jar
gson-2.8.6.jar
io-hotmoka-memory-1.0.7.jar
io-hotmoka-service-1.0.7.jar
io-hotmoka-crypto-1.0.7.jar
io-hotmoka-patricia-1.0.7.jar
io-hotmoka-tendermint-1.0.7.jar
...
io-takamaka-code-1.0.7.jar
it-univr-bcel-1.1.0.jar
picocli-4.6.1.jar

modules/unnamed:
animal-sniffer-annotations-1.19.jar
jakarta.el-3.0.3.jar
...
```

It is not possible to discuss here the difference between these kinds of modules (see [MakB17] for that). Just remember that explicit and automatic modules must be put in the module path, while unnamed modules must stay in the class path. Eclipse should do this automatically for us, as does the `moka` script that we have installed previously.

Chapter 3

# A First Takamaka Program

Takamaka is the language that can be used to write smart contracts for Hotmoka nodes. Hotmoka nodes and Takamaka code have exactly the same relation as Ethereum nodes and Solidity code.

Let us start from a simple example of Takamaka code. Since we are writing Java code, there is nothing special to learn or install before starting writing programs in Takamaka. Just use your preferred integrated development environment (IDE) for Java. Or even do everything from command-line, if you prefer. Our examples below will be shown for the Eclipse IDE, using Java 11 or later, but you can perfectly well use the IntelliJ IDE instead.

Our goal will be to create a Java class that we will instantiate and use in blockchain. Namely, we will learn how to create an object of that class, that will be persisted in blockchain, and how we can later call the `toString()` method on that instance in blockchain.

## Creation of the Eclipse Project

**[See the `family` project inside the `hotmoka_tutorial` repository]**

Let us create a Maven project `family` inside Eclipse, in the `hotmoka_tutorial` directory. For that, in the Eclipse's Maven wizard (New → Maven project) specify the options *Create a simple project (skip archetype selection)* and deselect the *Use default Workspace directory* option, specifying a subdirectory `family` of the `hotmaok_tutorial` directory as *Location* instead. Hence, *Location* should be something that ends with `.../tutorial/family`. Do not add the project to any working set. Use `io.hotmoka` as Group Id and `family` as Artifact Id.

> The Group Id can be changed as you prefer, but we will stick to `io.hotmoka` to show the exact files that you will see in Eclipse.

By clicking *Finish* in the Eclipse's Maven wizard, you should see a new Maven project in the Eclipse's explorer. Currently, Eclipse creates a default `pom.xml` file that uses Java 5 and has no dependencies. Replace hence the content of the `pom.xml` file of the `family` project with the code that follows:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>io.hotmoka</groupId>
  <artifactId>family</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>1.0.7</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>
```

that specifies to use Java 11 and provides the dependency to `io-takamaka-code`, that is, the run-time classes of the Takamaka smart contracts.

> We are using `1.0.7` here, as version of the Hotmoka and Takamaka runtime projects. Replace that, if needed, with the latest version of such projects.

Since the `pom.xml` file has changed, Eclipse will normally show an error in the `family` project. To solve it, you need to update the Maven dependencies of the project: right-click on the `family` project → Maven → Update Project...

As you can see, we are importing the dependency `io-takamaka-code`, that contains the Takamaka runtime. This will be downloaded from Maven and everything should compile without errors.

The result in Eclipse should look similar to what is shown in Figure 16.

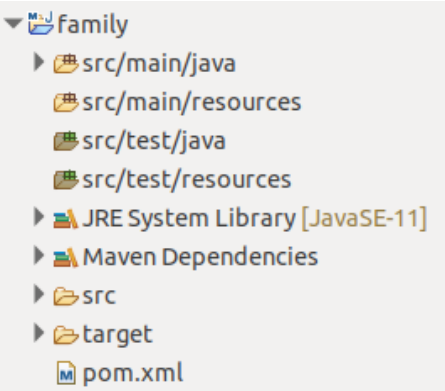



*Figure 16. The `family` Eclipse project.*

Create a `module-info.java` file inside `src/main/java` (right-click on the `family` project → Configure → Create module-info.java → Create), to state that this project depends on the module containing the runtime of Takamaka:

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

Create a package `io.takamaka.family` inside `src/main/java`. Inside that package, create a Java source `Person.java`, by copying and pasting the following code:

```
package io.takamaka.family;

public class Person {
  private final String name;
  private final int day;
  private final int month;
  private final int year;
  public final Person parent1;
  public final Person parent2;

  public Person(String name, int day, int month, int year,
                Person parent1, Person parent2) {

    this.name = name;
    this.day = day;
    this.month = month;
    this.year = year;
    this.parent1 = parent1;
    this.parent2 = parent2;
  }

  public Person(String name, int day, int month, int year) {
    this(name, day, month, year, null, null);
  }
```

```
  @Override
  public String toString() {
    return name + " (" + day + "/" + month + "/" + year + ")";
  }
}
```

This is a plain old Java class and should not need any comment.

Package the project into a jar, by running the following shell command inside the directory of the project (that is, the subdirectory `family` of the directory `hotmoka_tutorial`):

```
$ mvn package
```

A `family-0.0.1.jar` file should appear inside the `target` directory. Only the compiled class files will be relevant: Hotmoka nodes will ignore source files, manifest and any resources in the jar; the same compiled `module-info.class` is irrelevant for them. All such files can be removed from the jar, to reduce the gas cost of their installation in the store of a node, but we do not care about this optimization here. The result should look as in Figure 17:

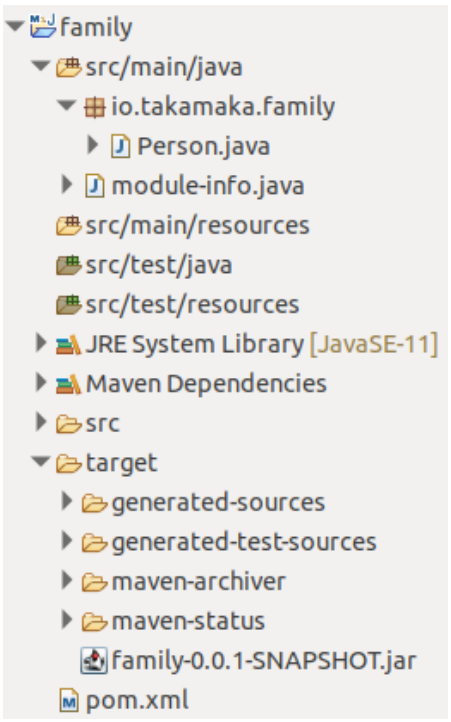



*Figure 17. The `family` Eclipse project, exported in jar.*

## Installation of the Jar in a Node

**[See the `runs` project inside the `hotmoka_tutorial` repository]**

We have generated the jar containing our code and we want to send it now to a Hotmoka node, where it will be installed. This means that it will become available to programmers who want to use its classes, directly or as dependencies of their programs. In order to install a jar in the Hotmoka node that we have used in the previous chapter, we can use the `moka` command-line tool, specifying which account will pay for the installation of the jar. The cost of the installation depends on the size of the jar and on the number of its dependencies. The `moka` tool uses a heuristics to foresee the cost of installation. Move inside the `hotmoka_tutorial` directory, if you are not there already, so that `moka` will find your saved entropy there, and run the `moka install` command:

```
$ cd hotmoka_tutorial
$ moka install family/target/family-0.0.1.jar
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io

Please specify the password of the payer account: chocolate
Do you really want to spend up to 494900 gas units to install the jar [Y/N] Y
family/target/family-0.0.1.jar has been installed
at 967a1c00eaf6e24cfe52efcdee1a0f037209fbc2bac63e1e9801d0e7860c5a8f
Total gas consumed: 219355
  for CPU: 261
  for RAM: 1299
  for storage: 217795
  for penalty: 0
```

As you can see above, the jar has been installed at a reference, that can be used later to refer to that jar. This has costed some gas, paid by our account. You can verify that the balance of the account has been decreased, through the `moka state` command.

The state of the the Hotmoka nodes of the testnet is now as in Figure 18. As that figure shows, a dependency has been created, automatically, from `family-0.0.1.jar` to `io-takamaka-code.jar`. This is because all Takamaka code will use the run-time classes of the Takamaka language, hence the `moka install` command adds them, by default. Note that a dependency must already be installed in the node before it can be used as dependency of other jars.

What we have done above is probably enough for most users, but sometimes you need to perform the same operation in code, for instance in order to implement a software application that connects to a Hotmoka node and runs some transactions. Therefore, we describe below how you can write a Java program that installs the same jar in the Hotmoka node, without using the `moka install` command. A similar translation in code can be performed for all examples in this tutorial, but we will report it only for a few of them.

Let us hence create another Eclipse Maven project `runs`, inside `hotmoka_tutorial`, exactly as we did in the previous section for the `family` project. Specify Java 11 (or later) in its build configuration. Use `io.hotmoka` as Group Id and `runs` as Artifact Id. This is specified in the following `pom.xml`, that you should copy inside the `runs` project, replacing that generated by Eclipse:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                      http://maven.apache.org/xsd/maven-4.0.0.xsd">
```

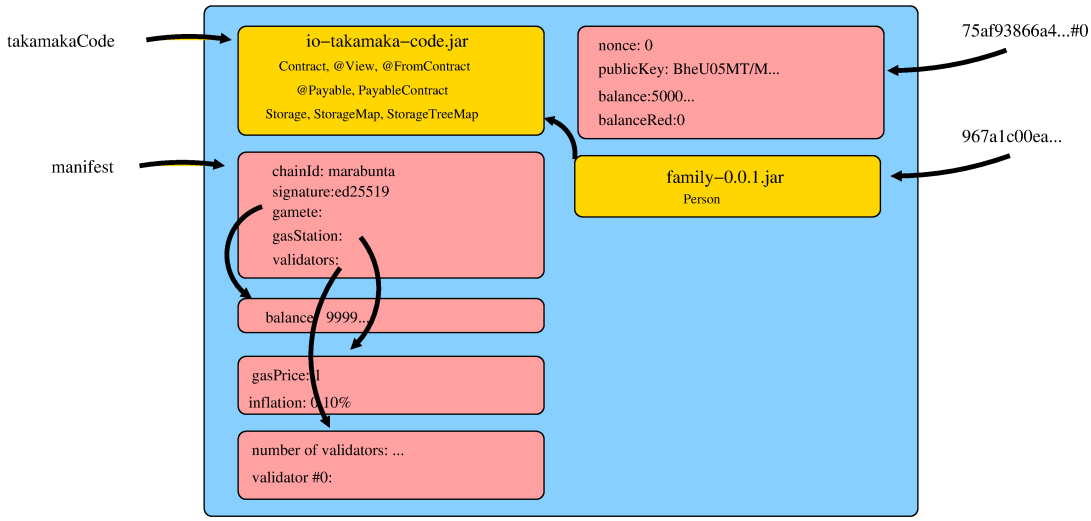



*Figure 18. The state of the test network nodes after the installation of our jar.*

```
<modelVersion>4.0.0</modelVersion>

<groupId>io.hotmoka</groupId>
<artifactId>runs</artifactId>
<version>0.0.1</version>
<packaging>jar</packaging>

<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>

<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>

<dependencies>
  <dependency>
    <groupId>io.hotmoka</groupId>
    <artifactId>io-hotmoka-remote</artifactId>
```

```
      <version>1.0.7</version>
    </dependency>
    <dependency>
      <groupId>io.hotmoka</groupId>
      <artifactId>io-hotmoka-helpers</artifactId>
      <version>1.0.7</version>
    </dependency>
    <dependency>
      <groupId>io.hotmoka</groupId>
      <artifactId>io-hotmoka-tendermint</artifactId>
      <version>1.0.7</version>
    </dependency>
    <dependency>
      <groupId>io.hotmoka</groupId>
      <artifactId>io-hotmoka-memory</artifactId>
      <version>1.0.7</version>
    </dependency>
    <dependency>
      <groupId>io.hotmoka</groupId>
      <artifactId>io-hotmoka-helpers</artifactId>
      <version>1.0.7</version>
    </dependency>
    <dependency>
      <groupId>io.hotmoka</groupId>
      <artifactId>io-hotmoka-service</artifactId>
      <version>1.0.7</version>
    </dependency>
  </dependencies>

</project>
```

This `pom.xml` specifies a few dependencies. We do not need all of them now, but we will need them along the next sections, hence let us insert them all already. These dependencies get automatically downloaded from the Maven repository.

Since we modified the file `pom.xml`, Eclipse should show an error for the `runs` project. To fix it, you need to update the Maven dependencies of the project: right-click on the `runs` project → Maven → Update Project...

Leave directory `src/test/java` empty, by deleting its content, if not already empty.

The result should look as in Figure 19.

Create a `module-info.java` inside `src/main/java`, containing:

```
module runs {
  requires io.hotmoka.beans;
  requires io.hotmoka.nodes;
  requires io.hotmoka.helpers;
  requires io.hotmoka.remote;
  requires io.hotmoka.crypto;
  requires io.hotmoka.memory;
  requires io.hotmoka.tendermint;
  requires io.hotmoka.service;
```

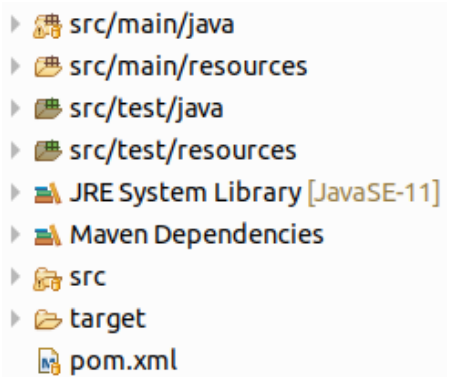



*Figure 19. The `runs` Eclipse project.*

```
  requires io.hotmoka.constants;
}
```

Again, we do not need all these dependencies already, but we will need them later.

Create a package `runs` inside `src/main/java` and add the following class `Family.java` inside it:

```
package runs;

import static java.math.BigInteger.ONE;

import java.math.BigInteger;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.security.KeyPair;

import io.hotmoka.beans.references.TransactionReference;
import io.hotmoka.beans.requests.InstanceMethodCallTransactionRequest;
import io.hotmoka.beans.requests.JarStoreTransactionRequest;
import io.hotmoka.beans.requests.SignedTransactionRequest;
import io.hotmoka.beans.requests.SignedTransactionRequest.Signer;
import io.hotmoka.beans.signatures.CodeSignature;
import io.hotmoka.beans.values.BigIntegerValue;
import io.hotmoka.beans.values.StorageReference;
import io.hotmoka.beans.values.StringValue;
import io.hotmoka.crypto.Account;
import io.hotmoka.crypto.SignatureAlgorithm;
import io.hotmoka.crypto.SignatureAlgorithmForTransactionRequests;
import io.hotmoka.helpers.GasHelper;
import io.hotmoka.helpers.SignatureHelper;
import io.hotmoka.nodes.Node;
import io.hotmoka.remote.RemoteNode;
import io.hotmoka.remote.RemoteNodeConfig;

public class Family {
```

```
// change this with your account's storage reference
private final static String
  ADDRESS = "75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0";

public static void main(String[] args) throws Exception {

  // the path of the user jar to install
  Path familyPath = Paths.get("../family/target/family-0.0.1.jar");

  RemoteNodeConfig config = new RemoteNodeConfig.Builder()
      .setURL("panarea.hotmoka.io")
      .build();

  try (Node node = RemoteNode.of(config)) {
      // we get a reference to where io-takamaka-code-1.0.7.jar has been stored
      TransactionReference takamakaCode = node.getTakamakaCode();

      // we get the signing algorithm to use for requests
      SignatureAlgorithm<SignedTransactionRequest> signature
        = SignatureAlgorithmForTransactionRequests.mk
      (node.getNameOfSignatureAlgorithmForRequests());

      StorageReference account = new StorageReference(ADDRESS);
      KeyPair keys = loadKeys(node, account);

      // we create a signer that signs with the private key of our account
      Signer signer = Signer.with(signature, keys.getPrivate());

      // we get the nonce of our account: we use the account itself as caller and
      // an arbitrary nonce (ZERO in the code) since we are running
      // a @View method of the account
      BigInteger nonce = ((BigIntegerValue) node
        .runInstanceMethodCallTransaction(new InstanceMethodCallTransactionRequest
          (account, // payer
          BigInteger.valueOf(50_000), // gas limit
          takamakaCode, // class path for the execution of the transaction
          CodeSignature.NONCE, // method
          account))) // receiver of the method call
        .value;

      // we get the chain identifier of the network
      String chainId = ((StringValue) node
        .runInstanceMethodCallTransaction(new InstanceMethodCallTransactionRequest
          (account, // payer
          BigInteger.valueOf(50_000), // gas limit
          takamakaCode, // class path for the execution of the transaction
          CodeSignature.GET_CHAIN_ID, // method
          node.getManifest()))) // receiver of the method call
        .value;

      GasHelper gasHelper = new GasHelper(node);

      // we install family-0.0.1.jar in the node: our account will pay
```

```
        TransactionReference family = node
          .addJarStoreTransaction(new JarStoreTransactionRequest
            (signer, // an object that signs with the payer's private key
            account, // payer
            nonce, // payer's nonce: relevant since this is not a call to a @View method!
            chainId, // chain identifier: relevant since this is not a call to a @View method!
            BigInteger.valueOf(300_000), // gas limit: enough for this very small jar
            gasHelper.getSafeGasPrice(), // gas price: at least the current gas price
            takamakaCode, // class path for the execution of the transaction
            Files.readAllBytes(familyPath), // bytes of the jar to install
            takamakaCode)); // dependencies of the jar that is being installed

        // we increase our copy of the nonce, ready for further
        // transactions having the account as payer
        nonce = nonce.add(ONE);

        System.out.println("family-0.0.1.jar installed at: " + family);
    }
  }

  private static KeyPair loadKeys(Node node, StorageReference account) throws Exception {
    return new Account(account, "..")
      .keys("chocolate", new SignatureHelper(node).signatureAlgorithmFor(account));
  }
}
```

As you can see, this class creates an instance of a `RemoteNode`, that represents a Hotmoka node installed in a remote host. By specifying the URL of the host, the `RemoteNode` object exposes all methods of a Hotmoka node. It is an `AutoCloseable` object, hence it is placed inside a try-with-resource statement that guarantees its release at the end of the `try` block. By using that remote node, our code collects some information about the node: the reference to the `io-takamaka-code` jar already installed inside it (`takamakaCode`) and the signature algorithm used by the node (`signature`), that it uses to construct a `signer` object that signs with the private key of our account, loaded from disk.

The `loadKeys` method accesses the `.pem` file that we have previously created with `moka`. It does it by looking in the parent directory `..`, that is where that file should be.

Like every Hotmoka node, the observable state of the remote node can only evolve through *transactions*, that modify its state in an atomic way. Namely, the code above performs three transactions:

1. A call to the `nonce()` method of our account: this is a progressive counter of the number of transactions already performed with our account. It starts from zero, but our account has been already used for other transactions (through the `moka` tool). Hence we better ask the node about it. As we will see later, this transaction calls a `@View` method. All calls to `@View` methods have the nice feature of being *for free*: nobody will pay for them. Because of that, we do not need to sign this transaction, or to provide a correct nonce, or specify a gas price. The limitation of such calls is that their transactions are not checked by consensus, hence we have to trust the node we ask. Moreover, they can only read, never write the data in the store of the node.
2. A call to the `getChainId()` method of the manifest of the node. Also this is a

call to a `@View` method, hence it costs nothing. The code invokes the method `runInstanceMethodCallTransaction()` of the node, with the `account` object as receiver of the call.

3. The addition of our jar in the node. This time the transaction has a cost and our account is specified as payer. The signer of our account signs the transaction. Nonce of our account and chain identifier of the network are relevant, as well as the gas price, that must at least match that of the network. The code uses the `addJarStoreTransaction()` method, that executes a new transaction on the node, whose goal is to install a jar inside it. The jar is provided as a sequence of bytes (`Files.readAllBytes("../family/target/family-0.0.1.jar")`, assuming that the `family` project is a sibling of the project `runs`). The request passed to `addJarStoreTransaction()` specifies that the transaction can cost up to 300,000 units of gas, that can be bought at a price returned by the `gasHelper` object. The request specifies that its class path is `node.getTakamakaCode()`: this is the reference to the `io-takamaka-code` jar already installed in the node. Finally, the request specifies that `family-0.0.1.jar` has only a single dependency: `io-takamaka-code`. This means that when, later, we will refer to `family-0.0.1.jar` in a class path, this class path will indirectly include its dependency `io-takamaka-code` as well (see Figure 18).

As in Ethereum, transactions in Hotmoka are paid in terms of gas consumed for their execution. Calls to `@View` methods do not actually modify the state of the node and are executed locally, on the node that receives the request of the transaction. Hence, they can be considered as run *for free*. Instead, we have used an actual gas price for the last transaction that installs the jar in blockchain. This could be computed with a sequence of calls to `@View` methods (get the manifest, then the gas station inside the manifest, then the gas price inside the gas station). In order to simplify the code, we have used the `GasHelper` class, that does exactly that for us.

You can run the program by selecting class `Family` in Eclipse and then the run menu option or the run green arrow of Eclipse.

You should see the following on the screen:

```
family-0.0.1.jar installed at:
84fde74632ced7e3654d40d796873c5542a4a17c4a1ee987dadebe5ea8f3f351
```

The exact address will change. In any case, note that this reference to the jar is functionally equivalent to that obtained before with the `moka install` command: they point to the same jar.

## Creation of an Object of our Program

**[See projects `runs` and `family_storage` inside the `hotmoka_tutorial` repository]**

The jar of our program is in the store of the node now: the `moka install` command has installed it at `967a1c00eaf6e24cfe52efcdee1a0f037209fbc2bac63e1e9801d0e7860c5a8f` and our code at `84fde74632ced7e3654d40d796873c5542a4a17c4a1ee987dadebe5ea8f3f351`. We can use either of them, interchangeably, as class path for the execution of a transaction that tries to run the constructor of `Person` and add a brand new `Person` object into the store of the node. We can perform this through the `moka` tool:

```
$ cd hotmoka_tutorial # if you are not already there
$ moka create
    io.takamaka.family.Person
    "Albert Einstein" 14 4 1879 null null
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --classpath 967a1c00eaf6e24cfe52efcdee1a0f037209fbc2bac63e1e9801d0e7860c5a8f
    --url panarea.hotmoka.io

Please specify the password of the payer account: chocolate
Do you really want to spend up to 500000 gas units to call
public Person(String,int,int,int,Person,Person) ? [Y/N] Y
Total gas consumed: 500000
  for CPU: 290
  for RAM: 1225
  for storage: 11310
  for penalty: 487175
io.hotmoka.beans.TransactionException: java.lang.IllegalArgumentException:
an object of class io.takamaka.family.Person cannot be kept in store
since it does not implement io.takamaka.code.lang.Storage
```

The `moka create` command requires to specify who pays for the object creation (our account), then the fully-qualified name of the class that we want to instantiate (`io.takamaka.family.Person`) followed by the actual arguments passed to its constructor. The classpath refers to the jar that we have installed previously. The `moka create` command asks for the password of the payer account and checks if we really want to proceed (and pay). Then it ends up in failure (`TransactionException`). Note that all offered gas has been spent. This is a sort of *penalty* for running a transaction that fails. The rationale is that this penalty should discourage potential denial-of-service attacks, when a huge number of failing transactions are thrown at a node. At least, that attack will cost a lot. Moreover, note that the transaction, although failed, does exist. Indeed, the nonce of the caller has been increased, as you can check with `moka state` on your account.

But we still have not understood why the transaction failed. The reason is in the exception message: `an object of class io.takamaka.family.Person cannot be kept in store since it does not implement io.takamaka.code.lang.Storage`. Takamaka requires that all objects stored in a node extend the `io.takamaka.code.lang.Storage` class. That superclass provides all the machinery needed in order to keep track of updates to such objects and persist them in the store of the node, automatically.

Do not get confused here. Takamaka does **not** require all objects to extend `io.takamaka.code.lang.Storage`. You can use objects that do not extend that superclass in your Takamaka code, both instances of your classes and instances of library classes from the `java.*` hierarchy, for instance. What Takamaka does require, instead, is that objects *that must be kept in the store of a node* do extend `io.takamaka.code.lang.Storage`. This must be the case, for instance, for objects created by the constructor invoked through the `moka create` command.

Let us modify the `io.takamaka.family.Person.java` source code, inside the `family` project then:

```
package io.takamaka.family;

import io.takamaka.code.lang.Storage;

public class Person extends Storage {
  ... unchanged code ...
}
```

Extending `io.takamaka.code.lang.Storage` is all a programmer needs to do in order to let instances of a class be stored in the store of a node. There is no explicit method to call to keep track of updates to such objects and persist them in the store of the node: Hotmoka nodes will automatically deal with them.

We can use the `io.takamaka.code.lang.Storage` class and we can run the resulting compiled code since that class is inside `io-takamaka-code`, that has been included in the class path as a dependency of `family-0.0.1.jar`.

Regenerate `family-0.0.1.jar`, by running `mvn package` again, inside the `family` project, since class `Person` has changed. Then run again the `moka create` command. This time, the execution should complete without exception:

```
$ cd family
$ mvn clean package
$ cd ..
$ moka install family/target/family-0.0.1.jar
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io
...
has been installed at
  d3e02c711680cc4d2f11c1593572512d35f2aaab3b62782933880b1c030239e3
...
$ moka create
    io.takamaka.family.Person
    "Albert Einstein" 14 4 1879 null null
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --classpath d3e02c711680cc4d2f11c1593572512d35f2aaab3b62782933880b1c030239e3
    --url panarea.hotmoka.io

Please specify the password of the payer account: chocolate
Do you really want to spend up to 500000 gas units to call
public Person(String,int,int,int,Person,Person) ? [Y/N] Y

The new object has been allocated at
  37735b40020370f0b1d0a7d1b83f60e591579868e383cf06156f28853a159155#0

Total gas consumed: 41721
  for CPU: 291
  for RAM: 1235
  for storage: 40195
  for penalty: 0
```

The new object has been allocated at a storage reference that can be used to refer to it, also in the future: `37735b40020370f0b1d0a7d1b83f60e591579868e383cf06156f28853a159155#0`. You can verify that it is actually there and that its fields are correctly initialized, by using the `moka state` command:

```
$ cd hotmoka_tutorial
$ moka state 37735b40020370f0b1d0a7d1b83f60e591579868e383cf06156f28853a159155#0
    --url panarea.hotmoka.io

This is the state of object
37735b40020370f0b1d0a7d1b83f60e591579868e383cf06156f28853a159155#0
@panarea.hotmoka.io

class io.takamaka.family.Person (from jar installed at
    d3e02c711680cc4d2f11c1593572512d35f2aaab3b62782933880b1c030239e3)

  day:int = 14
  month:int = 4
  name:java.lang.String = "Albert Einstein"
  parent1:io.takamaka.family.Person = null
  parent2:io.takamaka.family.Person = null
  year:int = 1879
```

Compared with Solidity, where contracts and accounts are just untyped *addresses*, objects (and hence accounts) are strongly-typed in Takamaka. This means that they are tagged with their run-time type (see the output of `moka state` above), in a boxed representation, so that it is possible to check that they are used correctly, ie., in accordance with the declared type of variables, or to check their run-time type with checked casts and the `instanceof` operator. Moreover, Takamaka has information to check that such objects have been created by using the same jar that stays in the class path later, every time an object gets used (see the information `from jar installed at` in the output of `moka state` above).

We can perform the same object creation in code, instead of using the `moka create` command. Namely, the following code builds on the previous example and installs a jar by adding a further transaction that calls the constructor of `Person`:

```
package runs;

import static io.hotmoka.beans.Coin.panarea;
import static io.hotmoka.beans.types.BasicTypes.INT;
import static java.math.BigInteger.ONE;

import java.math.BigInteger;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.security.KeyPair;

import io.hotmoka.beans.SignatureAlgorithm;
import io.hotmoka.beans.references.TransactionReference;
import io.hotmoka.beans.requests.ConstructorCallTransactionRequest;
import io.hotmoka.beans.requests.InstanceMethodCallTransactionRequest;
import io.hotmoka.beans.requests.JarStoreTransactionRequest;
import io.hotmoka.beans.requests.SignedTransactionRequest;
```

```
import io.hotmoka.beans.requests.SignedTransactionRequest.Signer;
import io.hotmoka.beans.signatures.CodeSignature;
import io.hotmoka.beans.signatures.ConstructorSignature;
import io.hotmoka.beans.types.ClassType;
import io.hotmoka.beans.values.BigIntegerValue;
import io.hotmoka.beans.values.IntValue;
import io.hotmoka.beans.values.StorageReference;
import io.hotmoka.beans.values.StringValue;
import io.hotmoka.crypto.Account;
import io.hotmoka.crypto.SignatureAlgorithmForTransactionRequests;
import io.hotmoka.helpers.GasHelper;
import io.hotmoka.helpers.SignatureHelper;
import io.hotmoka.nodes.Node;
import io.hotmoka.remote.RemoteNode;
import io.hotmoka.remote.RemoteNodeConfig;

public class Family2 {

  // change this with your account's storage reference
  private final static String ADDRESS =
    "75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0";

  private final static ClassType PERSON = new ClassType("io.takamaka.family.Person");

  public static void main(String[] args) throws Exception {

    // the path of the user jar to install
    Path familyPath = Paths.get("../family/target/family-0.0.1.jar");

    RemoteNodeConfig config = new RemoteNodeConfig.Builder()
      .setURL("panarea.hotmoka.io")
      .build();

    try (Node node = RemoteNode.of(config)) {
      // we get a reference to where io-takamaka-code has been stored
      TransactionReference takamakaCode = node.getTakamakaCode();

      // we get the signing algorithm to use for requests
      SignatureAlgorithm<SignedTransactionRequest> signature
        = SignatureAlgorithmForTransactionRequests.mk
            (node.getNameOfSignatureAlgorithmForRequests());

      StorageReference account = new StorageReference(ADDRESS);
      KeyPair keys = loadKeys(node, account);

      // we create a signer that signs with the private key of our account
      Signer signer = Signer.with(signature, keys.getPrivate());

      // we get the nonce of our account: we use the account itself as caller and
      // an arbitrary nonce (ZERO in the code) since we are running
      // a @View method of the account
      BigInteger nonce = ((BigIntegerValue) node
        .runInstanceMethodCallTransaction(new InstanceMethodCallTransactionRequest
```

```
    (account, // payer
    BigInteger.valueOf(50_000), // gas limit
    takamakaCode, // class path for the execution of the transaction
    CodeSignature.NONCE, // method
    account))) // receiver of the method call
  .value;

// we get the chain identifier of the network
String chainId = ((StringValue) node
  .runInstanceMethodCallTransaction(new InstanceMethodCallTransactionRequest
    (account, // payer
    BigInteger.valueOf(50_000), // gas limit
    takamakaCode, // class path for the execution of the transaction
    CodeSignature.GET_CHAIN_ID, // method
    node.getManifest()))) // receiver of the method call
  .value;

GasHelper gasHelper = new GasHelper(node);

// we install family-0.0.1.jar in the node: our account will pay
TransactionReference family = node
  .addJarStoreTransaction(new JarStoreTransactionRequest
    (signer, // an object that signs with the payer's private key
    account, // payer
    nonce, // payer's nonce: relevant since this is not a call to a @View method!
    chainId, // chain identifier: relevant since this is not a call to a @View method!
    BigInteger.valueOf(300_000), // gas limit: enough for this very small jar
    gasHelper.getSafeGasPrice(), // gas price: at least the current gas price of the network
    takamakaCode, // class path for the execution of the transaction
    Files.readAllBytes(familyPath), // bytes of the jar to install
    takamakaCode)); // dependencies of the jar that is being installed

// we increase our copy of the nonce, ready for further
// transactions having the account as payer
nonce = nonce.add(ONE);

// call the constructor of Person and store in albert the new object in blockchain
StorageReference albert = node.addConstructorCallTransaction
  (new ConstructorCallTransactionRequest
    (signer, // an object that signs with the payer's private key
    account, // payer
    nonce, // payer's nonce: relevant since this is not a call to a @View method!
    chainId, // chain identifier: relevant since this is not a call to a @View method!
    BigInteger.valueOf(50_000), // gas limit: enough for a small object
    panarea(gasHelper.getSafeGasPrice()), // gas price, in panareas
    family, // class path for the execution of the transaction

    // constructor Person(String,int,int,int)
    new ConstructorSignature(PERSON, ClassType.STRING, INT, INT, INT),

    // actual arguments
    new StringValue("Albert Einstein"), new IntValue(14),
    new IntValue(4), new IntValue(1879)
```

```
      ));

      System.out.println("New object allocated at " + albert);

      // we increase our copy of the nonce, ready for further
      // transactions having the account as payer
      nonce = nonce.add(ONE);
    }
  }

  private static KeyPair loadKeys(Node node, StorageReference account) throws Exception {
    return new Account(account, "..").keys
      ("chocolate", new SignatureHelper(node).signatureAlgorithmFor(account));
  }
}
```

The new transaction is due to the `addConstructorCallTransaction()` method, that expands the node with a new transaction that calls a constructor. We use our account as payer for the transaction, hence we sign the request with its private key. The class path includes `family-0.0.1.jar` and its dependency `io-takamaka-code`. The signature of the constructor specifies that we are referring to the second constructor of `Person`, the one that assumes `null` as parents. The actual parameters are provided; they must be instances of the `io.hotmoka.beans.values.StorageValue` interface. We provide 50,000 units of gas, which should be enough for a constructor that just initializes a few fields. We are ready to pay `panarea(gasHelper.getSafeGasPrice())` units of coin for each unit of gas. This price could have been specified simply as `gasHelper.getSafeGasPrice()` but we used the static method `io.hotmoka.beans.Coin.panarea()` to generate a `BigInteger` corresponding to the smallest coin unit of Hotmoka nodes, a *panarea*. Namely, the following units of coin exist:

| Value (in panas) | Exponent | Name | Short Name |
|---|---|---|---|
| 1 | 1 | panarea | pana |
| 1,000 | $10^3$ | alicudi | ali |
| 1,000,000 | $10^6$ | filicudi | fili |
| 1,000,000,000 | $10^9$ | stromboli | strom |
| 1,000,000,000,000 | $10^{12}$ | vulcano | vul |
| 1,000,000,000,000,000 | $10^{15}$ | salina | sali |
| 1,000,000,000,000,000,000 | $10^{18}$ | lipari | lipa |
| 1,000,000,000,000,000,000,000 | $10^{21}$ | moka | moka |

with corresponding static methods in `io.hotmoka.beans.Coin`.

Run `Family2` from Eclipse. You should see the following on the console:

```
New object allocated at
2bb2100a2368c0f80446e0a179e31949e05c1f1f7ef57058a2ca9d7d7622e81a#0
```

The exact address will change at any run.

# Calling a Method on an Object in a Hotmoka Node

**[See projects `runs` and `family_exported` inside the `hotmoka_tutorial` repository]**

In the previous section, we have created an object of class `Person` in the store of the node. Let us invoke the `toString()` method on that object now. For that, we can use the `moka call` command, specifying our `Person` object as *receiver*.

In object-oriented languages, the *receiver* of a call to a non-`static` method is the object over which the method is executed, that is accessible as `this` inside the code of the method. In our case, we want to invoke `albert.toString()`, where `albert` is the object that we have created previously, hence the receiver of the call. The receiver can be seen as an implicit actual argument passed to a (non-`static`) method.

```
$ moka call
    37735b40020370f0b1d0a7d1b83f60e591579868e383cf06156f28853a159155#0
    toString
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io

Please specify the password of the payer account: chocolate
Do you really want to spend up to 500000 gas units to call
public java.lang.String toString() ? [Y/N] Y

Total gas consumed: 0
  for CPU: 0
  for RAM: 0
  for storage: 0
  for penalty: 0
io.hotmoka.beans.TransactionRejectedException:
cannot pass as argument a value of the non-exported type io.takamaka.family.Person
```

Command `moka call` requires to specify, as its first arguments, the receiver of the call (the `Person` object created previously) and the name of the method to call (`toString`), followed by the actual arguments of the call, if any. We use the switch `--payer` to specify the payer of the transaction (our account).

As you can see above, the result is deceiving.

This exception occurs when we try to pass the `Person` object as receiver of `toString()` (the receiver is a particular case of an actual argument). That object has been created in store, has escaped the node and is available through its storage reference. However, it cannot be passed back into the node as argument of a call since it is not *exported*. This is a security feature of Hotmoka. Its reason is that the store of a node is public and can be read freely. Everybody can see the objects created in the store of a Hotmoka node and their storage references can be used to invoke their methods and modify their state. This is true also for objects meant to be private state components of other objects and that are not expected to be freely modifiable from outside the node. Because of this, Hotmoka requires that classes, whose instances can be passed into the node as arguments to methods or constructors, must be annotated as `@Exported`. This means that the programmer acknowledges the use of these instances from outside the node.

Note that all objects can be passed, from *inside* the blockchain, as arguments to methods of code in the node. The above limitation applies to objects passed from *outside* the node only.

Let us modify the `Person` class again:

```
...
import io.takamaka.code.lang.Exported;
...

@Exported
public class Person extends Storage {
  ...
}
```

Package the project `family` and try again to call the `toString` method:

```
$ cd family
$ mvn clean package
$ cd ..
$ moka install family/target/family-0.0.1.jar
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io
...
has been installed at
  b7fffc9dffff205774ae6753fa612a89429b886ec62fad4817a3370545b1a158
...
$ moka create
    io.takamaka.family.Person
    "Albert Einstein" 14 4 1879 null null
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --classpath b7fffc9dffff205774ae6753fa612a89429b886ec62fad4817a3370545b1a158
    --url panarea.hotmoka.io

...
The new object has been allocated at
  2bb311f3bb43e8b2550429347a09a9f730b6eb2688bb8ebb7c15c3c96c48e42e#0
...
$ moka call
    2bb311f3bb43e8b2550429347a09a9f730b6eb2688bb8ebb7c15c3c96c48e42e#0
    toString
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io

...
Albert Einstein (14/4/1879)

Total gas consumed: 19707
  for CPU: 350
  for RAM: 1360
  for storage: 17997
  for penalty: 0
```

This time, the correct answer `Albert Einstein (14/4/1876)` appears on the screen.

> In Ethereum, the only objects that can be passed, from outside the blockchain, as argument to method calls into blockchain are contracts. Namely, in Solidity it is possible to pass such objects as their untyped *address* that can only be cast to contract classes. Takamaka allows more, since *any* object can be passed as argument, not only contracts, as long as its class is annotated as `@Exported`. This includes all contracts since the class `io.takamaka.code.lang.Contract`, that we will present later, is annotated as `@Exported` and `@Exported` is an inherited Java annotation.

We can do the same in code, instead of using the `moka call` command. Namely, we can expand the `Family2` class seen before in order to run a further transaction, that calls `toString`. Copy then the following `Family3` class inside the `runs` package of the `runs` project:

```
package runs;

import static io.hotmoka.beans.Coin.panarea;
import static io.hotmoka.beans.types.BasicTypes.INT;
import static java.math.BigInteger.ONE;

import java.math.BigInteger;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.security.KeyPair;

import io.hotmoka.beans.SignatureAlgorithm;
import io.hotmoka.beans.references.TransactionReference;
import io.hotmoka.beans.requests.ConstructorCallTransactionRequest;
import io.hotmoka.beans.requests.InstanceMethodCallTransactionRequest;
import io.hotmoka.beans.requests.JarStoreTransactionRequest;
import io.hotmoka.beans.requests.SignedTransactionRequest;
import io.hotmoka.beans.requests.SignedTransactionRequest.Signer;
import io.hotmoka.beans.signatures.CodeSignature;
import io.hotmoka.beans.signatures.ConstructorSignature;
import io.hotmoka.beans.signatures.NonVoidMethodSignature;
import io.hotmoka.beans.types.ClassType;
import io.hotmoka.beans.values.BigIntegerValue;
import io.hotmoka.beans.values.IntValue;
import io.hotmoka.beans.values.StorageReference;
import io.hotmoka.beans.values.StorageValue;
import io.hotmoka.beans.values.StringValue;
import io.hotmoka.crypto.Account;
import io.hotmoka.crypto.SignatureAlgorithmForTransactionRequests;
import io.hotmoka.helpers.GasHelper;
import io.hotmoka.helpers.SignatureHelper;
import io.hotmoka.nodes.Node;
import io.hotmoka.remote.RemoteNode;
import io.hotmoka.remote.RemoteNodeConfig;

public class Family3 {
```

```
// change this with your account's storage reference
private final static String ADDRESS =
  "75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0";

private final static ClassType PERSON = new ClassType("io.takamaka.family.Person");

public static void main(String[] args) throws Exception {

  // the path of the user jar to install
  Path familyPath = Paths.get("../family/target/family-0.0.1.jar");

  RemoteNodeConfig config = new RemoteNodeConfig.Builder()
    .setURL("panarea.hotmoka.io")
    .build();

  try (Node node = RemoteNode.of(config)) {
    // we get a reference to where io-takamaka-code has been stored
    TransactionReference takamakaCode = node.getTakamakaCode();

    // we get the signing algorithm to use for requests
    SignatureAlgorithm<SignedTransactionRequest> signature
      = SignatureAlgorithmForTransactionRequests.mk
    (node.getNameOfSignatureAlgorithmForRequests());

    StorageReference account = new StorageReference(ADDRESS);
    KeyPair keys = loadKeys(node, account);

    // we create a signer that signs with the private key of our account
    Signer signer = Signer.with(signature, keys.getPrivate());

    // we get the nonce of our account: we use the account itself as caller and
    // an arbitrary nonce (ZERO in the code) since we are running
    // a @View method of the account
    BigInteger nonce = ((BigIntegerValue) node
      .runInstanceMethodCallTransaction(new InstanceMethodCallTransactionRequest
        (account, // payer
        BigInteger.valueOf(50_000), // gas limit
        takamakaCode, // class path for the execution of the transaction
        CodeSignature.NONCE, // method
        account))) // receiver of the method call
      .value;

    // we get the chain identifier of the network
    String chainId = ((StringValue) node
      .runInstanceMethodCallTransaction(new InstanceMethodCallTransactionRequest
        (account, // payer
        BigInteger.valueOf(50_000), // gas limit
        takamakaCode, // class path for the execution of the transaction
        CodeSignature.GET_CHAIN_ID, // method
        node.getManifest()))) // receiver of the method call
      .value;

    GasHelper gasHelper = new GasHelper(node);
```

```
// we install family-0.0.1.jar in the node: our account will pay
TransactionReference family = node
  .addJarStoreTransaction(new JarStoreTransactionRequest
    (signer, // an object that signs with the payer's private key
    account, // payer
    nonce, // payer's nonce: relevant since this is not a call to a @View method!
    chainId, // chain identifier: relevant since this is not a call to a @View method!
    BigInteger.valueOf(300_000), // gas limit: enough for this very small jar
    gasHelper.getSafeGasPrice(), // gas price: at least the current gas price of the network
    takamakaCode, // class path for the execution of the transaction
    Files.readAllBytes(familyPath), // bytes of the jar to install
    takamakaCode)); // dependencies of the jar that is being installed

nonce = nonce.add(ONE);

// call the constructor of Person and store in albert the new object in blockchain
StorageReference albert = node.addConstructorCallTransaction
  (new ConstructorCallTransactionRequest
    (signer, // an object that signs with the payer's private key
    account, // payer
    nonce, // payer's nonce: relevant since this is not a call to a @View method!
    chainId, // chain identifier: relevant since this is not a call to a @View method!
    BigInteger.valueOf(50_000), // gas limit: enough for a small object
    panarea(gasHelper.getSafeGasPrice()), // gas price, in panareas
    family, // class path for the execution of the transaction

    // constructor Person(String,int,int,int)
    new ConstructorSignature(PERSON, ClassType.STRING, INT, INT, INT),

    // actual arguments
    new StringValue("Albert Einstein"), new IntValue(14),
    new IntValue(4), new IntValue(1879)
));

nonce = nonce.add(ONE);

StorageValue s = node.addInstanceMethodCallTransaction
  (new InstanceMethodCallTransactionRequest
   (signer, // an object that signs with the payer's private key
   account, // payer
   nonce, // payer's nonce: relevant since this is not a call to a @View method!
   chainId, // chain identifier: relevant since this is not a call to a @View method!
   BigInteger.valueOf(50_000), // gas limit: enough for toString()
   panarea(gasHelper.getSafeGasPrice()), // gas price, in panareas
   family, // class path for the execution of the transaction

   // method to call: String Person.toString()
   new NonVoidMethodSignature(PERSON, "toString", ClassType.STRING),

   // receiver of the method to call
   albert
  ));
```

```
      // we increase our copy of the nonce, ready for further
      // transactions having the account as payer
      nonce = nonce.add(ONE);

      // print the result of the call
      System.out.println(s);
    }
  }

  private static KeyPair loadKeys(Node node, StorageReference account) throws Exception {
    return new Account(account, "..")
      .keys("chocolate", new SignatureHelper(node).signatureAlgorithmFor(account));
  }
}
```

The interesting part is the call to `addInstanceMethodCallTransaction()` at the end of the previous listing. It requires to resolve method `Person.toString()` using `albert` as receiver (the type `ClassType.STRING` is the return type of the method) and to run the resolved method. It stores the result in `s`, that subsequently prints on the standard output.

Run class `Family3` from Eclipse. You will obtain the same result as with `moka call`:

```
Albert Einstein (14/4/1879)
```

As we have shown, method `addInstanceMethodCallTransaction()` can be used to invoke an instance method on an object in the store of the node. This requires some clarification. First of all, note that the signature of the method to call is resolved and the resolved method is then invoked. If such resolved method is not found (for instance, if we tried to call `tostring` instead of `toString`), then `addInstanceMethodCallTransaction()` would end up in a failed transaction. Moreover, the usual resolution mechanism of Java methods applies. If, for instance, we invoked `new NonVoidMethodSignature(ClassType.OBJECT, "toString", ClassType.STRING)` instead of `new NonVoidMethodSignature(PERSON, "toString", ClassType.STRING)`, then method `toString` would have be resolved from the run-time class of `albert`, looking for the most specific implementation of `toString()`, up to the `java.lang.Object` class, which would anyway end up in running `Person.toString()`.

Method `addInstanceMethodCallTransaction()` can be used to invoke instance methods with parameters. If a `toString(int)` method existed in `Person`, then we could call it and pass 2019 as its argument, by writing:

```
StorageValue s = node.addInstanceMethodCallTransaction
  (new InstanceMethodCallTransactionRequest(
    ...

    // method to call: String Person.toString(int)
    new NonVoidMethodSignature(PERSON, "toString", ClassType.STRING, INT),

    // receiver of the method to call
    albert,

    // actual argument(s)
```

```
    new IntValue(2019)
  ));
```

where we have added the formal parameter `INT` (ie., `io.hotmoka.beans.types.BasicTypes.INT`) and the actual argument `new IntValue(2019)`.

Method `addInstanceMethodCallTransaction()` cannot be used to call a static method. For that, use `addStaticMethodCallTransaction()` instead, that accepts a request similar to that for `addInstanceMethodCallTransaction()`, but without a receiver.

## Storage Types and Constraints on Storage Classes

We have seen how to invoke a constructor of a class to build an object in the store of a node or to invoke a method on an object in the store of a node. Both constructors and methods can receive arguments. Constructors yield a reference to a new object, freshly allocated; methods might yield a returned value, if they are not declared as `void`. This means that there is a bidirectional exchange of data from outside the node to inside it, and back. But not any kind of data can be exchanged:

1. The values that can be exchanged from inside the node to outside the node are called *storage values*.
2. The values that can be exchanged from outside the node to inside the node are the same *storage values* as above, with the extra constraint that objects must belong to an `@Exported` class.

The set of *storage values* is the union of

1. primitive values of Java (characters, bytes, shorts, integers, longs, floats, doubles and booleans);
2. reference values whose class extends `io.takamaka.code.lang.Storage` (that is, *storage objects*);
3. `null`;
4. elements of an `enum` without instance non-transient fields;
5. a few special reference values: `java.math.BigInteger`s and `java.lang.String`s.

Storage values cross the node's boundary inside wrapper objects. For instance the integer 2019 is first wrapped into `new IntValue(2019)` and then passed as a parameter to a method or constructor. In our previous example, when we called `Person.toString()`, the result `s` was actually a wrapper of a `java.lang.String` object. Boxing and unboxing into/from wrapper objects is automatic: our class `Person` does not show that machinery.

What should be retained of the above discussion is that constructors and methods of Takamaka classes, if we want them to be called from outside the node, must receive storage values as parameters and must return storage values (if they are not `void` methods). A method that expects a parameter of type `java.util.HashSet`, for instance, can be defined and called from Takamaka code, inside the node, but cannot be called from outside the node, such as, for instance, from the `moka` tool or from our `Family` class. The same occurs if the method returns a `java.util.HashSet`.

We conclude this section with a formal definition of storage objects. We have already said that storage objects can be kept in the store of a node and their class must extend `io.takamaka.code.lang.Storage`. But there are extra constraints. Namely, fields of a storage

objects are part of the representation of such objects and must, themselves, be kept in store. Hence, a storage object:

1. has a class that extends (directly or indirectly) `io.takamaka.code.lang.Storage`, and
2. is such that all its fields hold storage values (primitives, storage objects, `null`, elements of `enum`s without instance non-transient fields, a `java.math.BigInteger` or a `java.lang.String`).

Note that the above conditions hold for the class `Person` defined above. Instead, the following are examples of what is **not** allowed in a field of a storage object:

1. arrays
2. collections from `java.util.*`

We will see later how to overcome these limitations.

> Again, we stress that such limitations only apply to storage objects. Other objects, that needn't be kept in the store of a node but are useful for the implementation of Takamaka code, can be defined in a completely free way and used in code that runs in the node.

## Transactions Can Be Added, Posted and Run

We have executed transactions on a Hotmoka node with methods `addJarStoreTransaction()`, `addConstructorCallTransaction()` and `addInstanceMethodCallTransaction()`. These methods, whose name starts with `add`, are *synchronous*, meaning that they block until the transaction is executed (or fails). If they are invoked on a node with a notion of commit, such as a blockchain, they guarantee to block until the transaction is actually committed. In many cases, when we immediately need the result of a transaction before continuing with the execution of the subsequent statements, these methods are the right choice. In many other cases, however, it is unnecessary to wait until the transaction has completed its execution and has been committed. In those cases, it can be faster to execute a transaction through a method whose name starts with `post`, such as `postJarStoreTransaction()`, `postConstructorCallTransaction()` or `postInstanceMethodCallTransaction()`. These methods are called *asynchronous*, since they terminate immediately, without waiting for the outcome of the transaction they trigger. Hence they cannot return their outcome immediately and return a *future* instead, whose `get()` value, if and when invoked, will block until the outcome of the transaction is finally available.

For instance, instead of the inefficient:

```
StorageValue s = node.addInstanceMethodCallTransaction
  (new InstanceMethodCallTransactionRequest(
    signer,
    account,
    nonce,
    chainId,
    BigInteger.valueOf(50_000),
    panarea(gasHelper.getSafeGasPrice()),
    family,
    new NonVoidMethodSignature(PERSON, "toString", ClassType.STRING),
    albert
  ));
```

```
// code that does not use s
// .....
```

one can write the more efficient:

```
CodeSupplier<StorageValue> future = node.postInstanceMethodCallTransaction
  (new InstanceMethodCallTransactionRequest(
    signer,
    account,
    nonce,
    chainId,
    BigInteger.valueOf(50_000),
    panarea(gasHelper.getSafeGasPrice()),
    family,
    new NonVoidMethodSignature(PERSON, "toString", ClassType.STRING),
    albert
  ));

// code that does not use s
// .....

// the following will be needed only if s is used later
StorageValue s = future.get();
```

There is a third way to execute a transaction. Namely, calls to methods annotated as `@View` can be performed through the `runInstanceMethodCallTransaction()` (for instance methods) and `runStaticMethodCallTransaction()` (for static methods). As we have hinted before, these executions are performed locally, on the node they are addressed to, and do not add a transaction that must be replicated in each node of the network, for consensus, and that costs gas for storage. These executions are free and do not require a correct nonce, signature, or chain identifier, which is a great simplification.

Chapter 4

# The Notion of Smart Contract

A contract is a legal agreement among two or more parties. A good contract should be unambiguous, since otherwise its interpretation could be questioned or misunderstood. A legal system normally enforces the validity of a contract. In the context of software development, a *smart contract* is a piece of software with deterministic behavior, whose semantics should be clear and enforced by a consensus system. Blockchains provide the perfect environment where smart contracts can be deployed and executed, since their (typically) non-centralized nature reduces the risk that a single party overthrows the rules of consensus, by providing for instance a non-standard semantics for the code of the smart contract.

Contracts are allowed to hold and transfer money to other contracts. Hence, traditionally, smart contracts are divided into those that hold money but have no code (*externally owned accounts*), and those that, instead, contain code (*smart contracts*). The formers are typically controlled by an external agent (a wallet, a human or a software application, on his behalf) while the latters are typically controlled by their code. Takamaka implements both alternatives as instances of the abstract library class `io.takamaka.code.lang.Contract` (inside `io-takamaka-code`). That class extends `io.takamaka.code.lang.Storage`, hence its instances can be kept in the store of the node. Moreover, that class is annotated as `@Exported`, hence nodes can receive references to contract instances from the outside world. The Takamaka library defines subclasses of `io.takamaka.code.lang.Contract`, that we will investigate later. Programmers can define their own subclasses as well.

This chapter presents a simple smart contract, whose goal is to enforce a Ponzi investment scheme: each investor pays back the previous investor, with at least a 10% reward; as long as new investors keep coming, each investor gets at least a 10% reward; the last investor, instead, will never see his/her investment back. The contract has been inspired by a similar Ethereum contract, shown at page 145 of [IyerD08].

We will develop the contract in successive versions, in order to highlight the meaning of different language features of Takamaka.

# A Simple Ponzi Scheme Contract

**[See project `ponzi_simple` inside the `hotmoka_tutorial` repository]**

Create a new Maven Java 11 (or later) project in Eclipse, named `ponzi`. You can do this by duplicating the project `family` (make sure to store the project inside the `hotmoka_tutorial` directory, as a sibling of `family` and `runs`). Use the following `pom.xml`:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <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>1.0.7</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>
```

and the following `module-info.java`:

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

Create package `io.takamaka.ponzi` inside `src/main/java` and add the following `SimplePonzi.java`

source inside that package:

```
package io.takamaka.ponzi;

import static io.takamaka.code.lang.Takamaka.require;
import java.math.BigInteger;
import io.takamaka.code.lang.Contract;

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

  public void invest(Contract investor, BigInteger amount) {
    // new investments must be at least 10% greater than current
    BigInteger minimumInvestment = currentInvestment.multiply(_11).divide(_10);
    require(amount.compareTo(minimumInvestment) >= 0,
      () -> "you must invest at least " + minimumInvestment);

    // document new investor
    currentInvestor = investor;
    currentInvestment = amount;
  }
}
```

> This code is only the starting point of our discussion and is not functional yet. The real final version of this contract will appear at the end of this section.

Look at the code of `SimplePonzi.java` above. The contract has a single method, named `invest`. This method lets a new `investor` invest a given `amount` of coins. This amount must be at least 10% higher than the current investment. The expression `amount.compareTo(minimumInvestment) >= 0` is a comparison between two Java `BigInteger`s and should be read as the more familiar `amount >= minimumInvestment`: the latter cannot be written in this form, since Java does not allow comparison operators to work on reference types. The static method `io.takamaka.code.lang.Takamaka.require()` is used to require some precondition to hold. The `require(condition, message)` call throws an exception if `condition` does not hold, with the given `message`. If the new investment is at least 10% higher than the current one, it will be saved in the state of the contract, together with the new investor.

> You might wonder why we have written `require(..., () -> "you must invest at least " + minimumInvestment)` instead of the simpler `require(..., "you must invest at least " + minimumInvestment)`. Both are possible and semantically almost identical. However, the former uses a lambda expression that computes the string concatenation lazily, only if the message is needed; the latter always computes the string concatenation, instead. Hence, the first version consumes less gas, in general, and is consequently preferrable. This technique simulates lazy evaluation in a language, like Java, that has only eager evaluation for actual arguments. This technique has been used since years, for instance in JUnit assertions.

## The `@FromContract` and `@Payable` Annotations

**[See project `ponzi_annotations` inside the `hotmoka_tutorial` repository]**

The previous code of `SimplePonzi.java` is unsatisfactory, for at least two reasons, that we will overcome in this section:

1. Any contract can call `invest()` and let *another* `investor` contract invest in the game. This is against our intuition that each investor decides when and how much he (himself) decides to invest.
2. There is no money transfer. Anybody can call `invest()`, with an arbitrary `amount` of coins. The previous investor does not get the investment back when a new investor arrives since, well, he never really invested anything.

Let us rewrite `SimplePonzi.java` in the following way:

```
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;

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

  public @FromContract void invest(BigInteger amount) {
    // new investments must be at least 10% greater than current
    BigInteger minimumInvestment = currentInvestment.multiply(_11).divide(_10);
    require(amount.compareTo(minimumInvestment) >= 0,
      () -> "you must invest at least " + minimumInvestment);

    // document new investor
    currentInvestor = caller();
    currentInvestment = amount;
  }
}
```

The difference with the previous version of `SimplePonzi.java` is that the `investor` argument of `invest()` has disappeared. At its place, `invest()` has been annotated as `@FromContract`. This annotation **restricts** the possible uses of method `invest()`. Namely, it can only be called from a contract object $c$ or from an external wallet, with a paying contract $c$, that pays for a transaction that runs `invest()`. It cannot, instead, be called from the code of a class that is not a contract. The instance of contract $c$ is available, inside `invest()`, as `caller()`. This is, indeed, saved, in the above code, into `currentInvestor`.

The annotation `@FromContract` can be applied to both methods and constructors. If a `@FromContract` method is redefined, the redefinitions must also be annotated as `@FromContract`.

Method `caller()` can only be used inside a `@FromContract` method or constructor and refers to the contract that called that method or constructor or to the contract that pays for a call, from a wallet, to the method or constructor. Hence, it will never yield `null`. If a `@FromContract` method or constructor calls another method $m$, then the `caller()` of the former is **not** available inside $m$, unless the call occurs, syntactically, on `this`, in which case the `caller()` is preserved. By *syntactically*, we mean through expressions such as `this.m(...)` or `super.m(...)`.

The use of `@FromContract` solves the first problem: if a contract invests in the game, then it is the caller of `invest()`. However, there is still no money transfer in this version of `SimplePonzi.java`. What we still miss is to require the caller of `invest()` to actually pay for the `amount` units of coin. Since `@FromContract` guarantees that the caller of `invest()` is a contract and since contracts hold money, this means that the caller contract of `invest()` can be charged `amount` coins at the moment of calling `invest()`. This can be achieved with the `@Payable` annotation, that we apply to `invest()`:

```
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;

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

  public @Payable @FromContract void invest(BigInteger amount) {
    // new investments must be at least 10% greater than current
    BigInteger minimumInvestment = currentInvestment.multiply(_11).divide(_10);
    require(amount.compareTo(minimumInvestment) >= 0,
      () -> "you must invest at least " + minimumInvestment);

    // document new investor
    currentInvestor = caller();
    currentInvestment = amount;
  }
}
```

When a contract calls `invest()` now, that contract will be charged `amount` coins, automatically. This means tha these coins will be automatically transferred to the balance of the instance of `SimplePonzi` that receives the call. If the balance of the calling contract is too low for that, the call will be automatically rejected with an insufficient funds exception. The caller must be able to pay for both `amount` and the gas needed to run `invest()`. Hence, he must hold a bit more than `amount` coins at the moment of calling `invest()`.

The `@Payable` annotation can only be applied to a method or constructor that is also annotated as `@FromContract`. If a `@Payable` method is redefined, the redefinitions must also be annotated as `@Payable`. A `@Payable` method or constructor must have a first argument of type `int`, `long` or `java.math.BigInteger`, dependending on the amount of coins that the programmer allows one to transfer at call time. The name of that argument is irrelevant, but we will keep using `amount` for it.

## Payable Contracts

**[See project `ponzi_payable` inside the `hotmoka_tutorial` repository]**

The `SimplePonzi.java` class is not ready yet. Namely, the code of that class specifies that investors have to pay an always increasing amount of money to replace the current investor. However, in the current version of the code, the replaced investor never gets his previous investment back, plus the 10% award (at least): money keeps flowing inside the `SimplePonzi` contract and remains stuck there, forever. The code needs an apparently simple change: just add a single statement before the update of the new current investor. That statement should send `amount` units of coin back to `currentInvestor`, before it gets replaced:

```
// document new investor
if (currentInvestor != null)
  currentInvestor.receive(amount);
currentInvestor = caller();
currentInvestment = amount;
```

In other words, a new investor calls `invest()` and pays `amount` coins to the `SimplePonzi` contract (since `invest()` is `@Payable`); then this `SimplePonzi` contract transfers the same `amount` of coins to pay back the previous investor. Money flows through the `SimplePonzi` contract but does not stay there for long.

The problem with this simple line of code is that it does not compile. There is no `receive()` method in `io.takamaka.code.lang.Contract`: a contract can receive money only through calls to its `@Payable` constructors and methods. Since `currentInvestor` is, very generically, an instance of `Contract`, that has no `@Payable` methods, there is no method that we can call here for sending money back to `currentInvestor`. This limitation is a deliberate design choice of Takamaka.

Solidity programmers will find this very different from what happens in Solidity contracts. Namely, these always have a *fallback function* that can be called for sending money to a contract. A problem with Solidity's approach is that the balance of a contract is not fully controlled by its payable methods, since money can always flow in through the fallback function (and also in other, more surprising ways). This led to software bugs, when a contract found itself richer then expected, which violated some (wrong) invariants about its state. For more information, see page 181 of [AntonopoulosW19] (*Unexpected Ether*).

So how do we send money back to `currentInvestor`? The solution is to restrict the kind of contracts that can participate to the Ponzi scheme. Namely, we limit the game to contracts that implement class `io.takamaka.code.lang.PayableContract`, a subclass of `io.takamaka.code.lang.Contract` that, yes, does have a payable `receive()` method. This is

not really a restriction, since the typical players of our Ponzi contract are externally owned accounts, that are instances of `PayableContract`.

Let us hence apply the following small changes to our `SimplePonzi.java` class:

1. The type of `currentInvestment` must be restricted to `PayableContract`.
2. The `invest()` method must be callable by `PayableContract`s only.
3. The return value of `caller()` must be cast to `PayableContract`, which is safe because of point 2 above.

The result is the following:

```
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;

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

  public @Payable @FromContract(PayableContract.class) void invest(BigInteger amount) {
    // new investments must be at least 10% greater than current
    BigInteger minimumInvestment = currentInvestment.multiply(_11).divide(_10);
    require(amount.compareTo(minimumInvestment) >= 0,
      () -> "you must invest at least " + minimumInvestment);

    // document new investor
    if (currentInvestor != null)
      currentInvestor.receive(amount);

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

Note the use of `@FromContract(PayableContract.class)` in the code above: a method or constructor annotated as `@FromContract(C.class)` can only be called by a contract whose class is `C` or a subclass of `C`. Otherwise, a run-time exception will occur.

## The `@View` Annotation

**[See project `ponzi_view` inside the `hotmoka_tutorial` repository]**

Our `SimplePonzi.java` code can still be improved. As it is now, an investor must call `invest()` and be ready to pay a sufficiently large `amount` of coins to pay back and replace the previous

investor. How much is *large* actually large enough? Well, it depends on the current investment. But that information is kept inside the contract and there is no easy way to access it from outside. An investor can only try with something that looks large enough, running a transaction that might end up in two scenarios, both undesirable:

1. The amount invested was actually large enough, but larger than needed: the investor invested more than required in the Ponzi scheme, taking the risk that no one will ever invest more and pay him back.
2. The amount invested might not be enough: the `require()` function will throw an exception that makes the transaction running `invest()` fail. The investment will not be transferred to the `SimplePonzi` contract, but the investor will be penalized by charging him all the gas provided for the transaction. This is unfair since, after all, the investor had no way to know that the proposed investment was not large enough.

Hence, it would be nice and fair to provide investors with a way to access the value in the `currentInvestment` field. This is actually a piece of cake: just add this method to `SimplePonzi.java`:

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

This solution is perfectly fine but can be improved. Written this way, an investor that wants to call `getCurrentInvestment()` must run a Hotmoka transaction through the `addInstanceMethodCallTransaction()` method of the node, creating a new transaction that ends up in the store of the node. That transaction will cost gas, hence its side-effect will be to reduce the balance of the calling investor. But the goal of the caller was just to access information in the store of the node, not to modify the store through side-effects. The balance reduction for the caller is, indeed, the *only* side-effect of that call! In cases like this, Takamaka allows one to specify that a method is expected to have no side-effects on the visible state of the node, but for the change of the balance of the caller. This is possible through the `@View` annotation. Import that class in the Java source and edit the declaration of `getCurrentInvestment()`, as follows:

```
import io.takamaka.code.lang.View;
...
  public @View BigInteger getCurrentInvestment() {
    return currentInvestment;
  }
```

An investor can now call that method through another API method of the Hotmoka nodes, called `runInstanceMethodCallTransaction()`, that does not expand the store of the node, but yields the response of the transaction, including the returned value of the call. If method `getCurrentInvestment()` had side-effects beyond that on the balance of the caller, then the execution will fail with a run-time exception. Note that the execution of a `@View` method still requires gas, but that gas is given back at the end of the call. The advantage of `@View` is hence that of allowing the execution of `getCurrentInvestment()` for free and without expanding the store of the node with useless transactions, that do not modify its state. Moreover, transactions run through `runInstanceMethodCallTransaction()` do not need a correct nonce, chain identifier or signature, hence any constant value can be used for them. This simplifies the call. For the same reason, transactions run through `runInstanceMethodCallTransaction()` do not count for the computation of the nonce of the caller.

The annotation `@View` is checked at run time if a transaction calls the `@View` method from outside the blockchain, directly. It is not checked if, instead, the method is called indirectly, from other Takamaka code. The check occurs at run time, since the presence of side-effects in computer code is undecidable. Future versions of Takamaka might check `@View` at the time of installing a jar in a node, as part of bytecode verification. That check can only be an approximation of the run-time check.

If a `@View` method is called through the `moka call` command, the `moka` tool will automatically perform a `runInstanceMethodCallTransaction()` internally, to spare gas.

## The Hierarchy of Contracts

Figure 23 shows the hierarchy of Takamaka contract classes. The topmost abstract class `io.takamaka.code.lang.Contract` extends `io.takamaka.code.lang.Storage`, since contracts are meant to be stored in a node (as are other classes that are not contracts, such as our first `Person` example). Programmers typically extend `Contract` to define their own contracts. This is the case, for instance, of our `SimplePonzi` class. Class `Storage` provides a `caller()` final protected method that can be called inside `@FromContract` methods and constructors, to access the calling contract. Class `Contract` provides a final `@View` method `balance()` that can be used to access the private `balance` field of the contract. Note that class `Contract` is annotated with the inherited annotation `@Exported`, hence contracts, such as instances of `SimplePonzi`, can be receivers of calls from outside the node and can be passed as arguments to calls from outside the node. Instances of `Storage` are not normally `@Exported`, unless their class is explicitly annotated as `@Exported`, as we did for `Person`.

The abstract subclass `PayableContract` is meant for contracts that can receive coins from other contracts, through their final `receive()` methods. Its concrete subclass named `ExternallyOwnedAccount` is a payable contract that can be used to pay for a transaction. Such *accounts* are typically controlled by humans, through a wallet, but can be subclassed and instantiated freely in Takamaka code. Their constructors allow one to build an externally owned account and fund it with an initial amount of coins. As we have seen in sections Installation of the Jar in a Hotmoka Node, Creation of an Object of our Program and Calling a Method on an Object in a Hotmoka Node, the methods of Hotmoka nodes that start a transaction require to specify a payer for that transaction. Such a payer is required to be an instance of `ExternallyOwnedAccount`, or an exception will be thrown. In our previous examples, we have used, as payer, an account created by the `moka create-account` command, that is an instance of `io.takamaka.code.lang.ExternallyOwnedAccount`. `ExternallyOwnedAccounts` have a private field `nonce` that can be accessed through the public `@View` method `nonce()`: it yields a `BigInteger` that specifies the next nonce to use for the next transaction having that account as caller. This nonce gets automatically increased after each such transaction.

Instances of `ExternallyOwnedAccounts` hold their public key in their private `publicKey` field, as a Base64-encoded string, that can be accessed through the `publicKey()` method. That key is used to verify the signature of the transactions having that account as caller. As we will see later, there is a default signature algorithms for transactions and that is what `ExternallyOwnedAccounts` use. However, it is possible to require a specific signature algorithm, that overrides the default for the node. For that, it is enough to

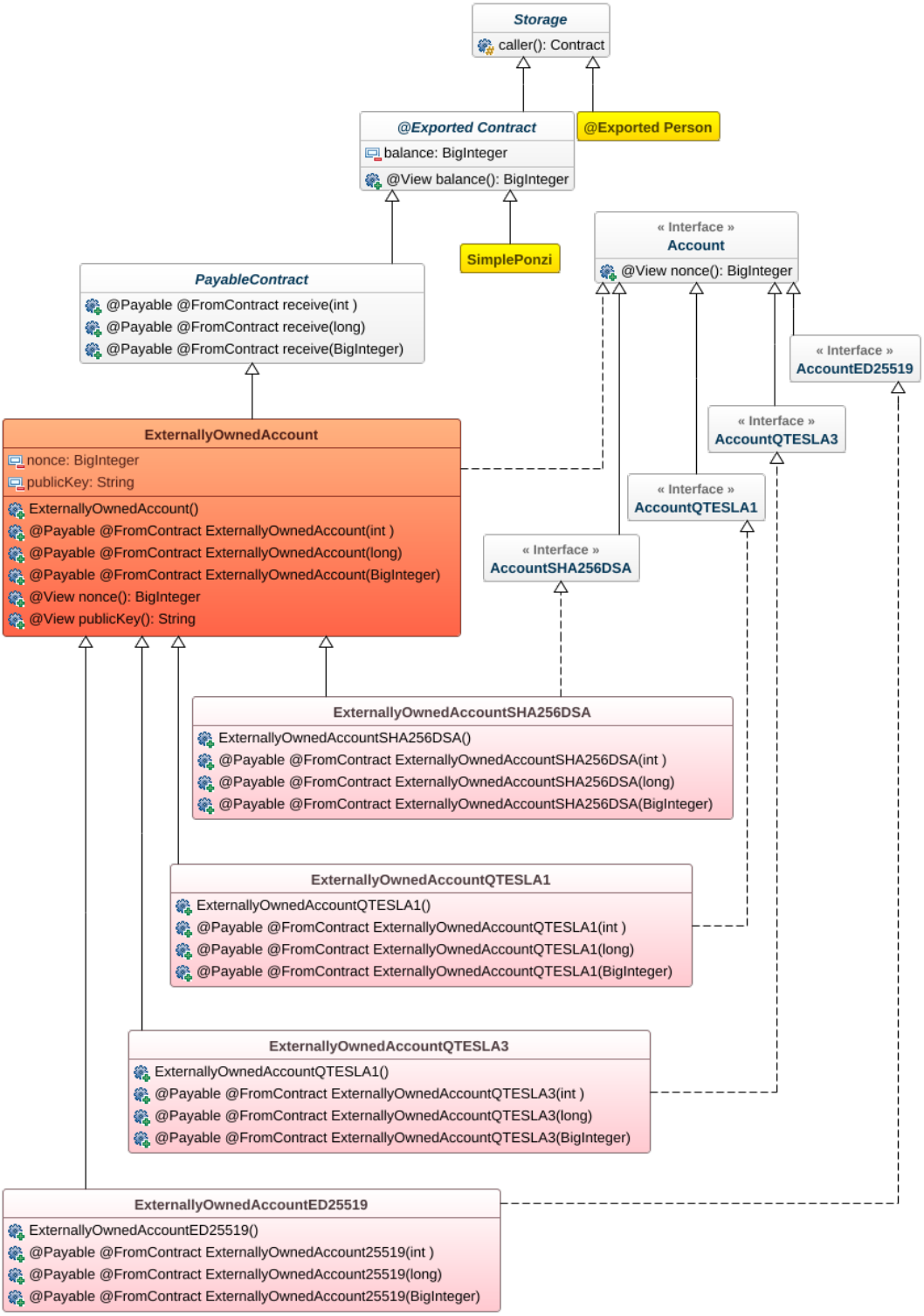



*Figure 23. The hierarchy of contract classes.*

instantiate classes `ExternallyOwnedAccountSHA256DSA`, `ExternallyOwnedAccountED25519`, `ExternallyOwnedAccountQTESLA1` or `ExternallyOwnedAccountQTESLA3`. The latter two use a quantum-resistant signature algorithm (see Signatures and Quantum-Resistance for more details). This means that it is possible to mix many signature algorithms for signing transactions inside the same Hotmoka node, as we will show later.

## Red and Green Balances

**[See project `redgreen` inside the `hotmoka_tutorial` repository]**

The `Contract` class of Takamaka has a double balance. Namely, it has a normal (*green*) balance and an extra, stable *red* balance. That is, contracts have the ability to keep an extra red balance, that should be a stable coin, if the underlying blockchain supports that feature.

For instance, the following contract allows payees to register by calling the `addAsPayee()` method. Moreover, the contract distributes green coins sent to the `distributeGreen()` method and red coins sent to the `distributeRed()` method, sending the rest to the owner of the contract (in general, there is a rest because of arithmetic approximation). Hence, the contract holds coins only temporarily. The `@RedPayable` annotation states that the `distributeRed()` method can receive red coins when called. Class `StorageLinkedList` holds a list of contracts and will be discussed in the next chapter.

```
package io.takamaka.redgreen;

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.RedPayable;
import io.takamaka.code.util.StorageLinkedList;
import io.takamaka.code.util.StorageList;

public class Distributor extends Contract {
  private final StorageList<PayableContract> payees = new StorageLinkedList<>();
  private final PayableContract owner;

  public @FromContract(PayableContract.class) Distributor() {
    owner = (PayableContract) caller();
  }

  public @FromContract(PayableContract.class) void addAsPayee() {
    payees.add((PayableContract) caller());
  }

  public @Payable @FromContract void distributeGreen(BigInteger amount) {
    int size = payees.size();
    if (size > 0) {
      BigInteger eachGets = amount.divide(BigInteger.valueOf(size));
      payees.forEach(payee -> payee.receive(eachGets));
      owner.receive(balance());
```

```
    }
  }

  public @RedPayable @FromContract void distributeRed(BigInteger amount) {
    int size = payees.size();
    if (size > 0) {
      BigInteger eachGets = amount.divide(BigInteger.valueOf(size));
      payees.forEach(payee -> payee.receiveRed(eachGets));
      owner.receiveRed(balanceRed());
    }
  }
}
```

Chapter 5

# The Support Library

This chapter presents the support library of the Takamaka language, that contains classes for simplifying the definition of smart contracts.

In Storage Types and Constraints on Storage Classes, we said that storage objects must obey to some constraints. The strongest of them is that their fields of reference type, in turn, can only hold storage objects. In particular, arrays are not allowed there. This can be problematic, in particular for contracts that deal with a dynamic, variable, potentially unbound number of other contracts.

Therefore, most classes of the support library deal with such constraints, by providing fixed or variable-sized collections that can be used in storage objects, since they are storage objects themselves. Such utility classes implement lists, arrays and maps and are consequently generally described as *collections*. They have the property of being storage classes, hence their instances can be kept in the store of a Hotmoka node, *as long as only storage objects are added as elements of the collection*. As usual with collections, these utility classes have generic type, to implement collections of arbitrary, but fixed types. This is not problematic, since Java (and hence Takamaka) allows generic types.

## Storage Lists

Lists are an ordered sequence of elements. In a list, it is typically possible to access the first element in constant time, while accesses to the $n$th element require to scan the list from its head and consequently have a cost proportional to $n$. Because of this, lists are **not**, in general, random-access data structures, whose $n$th element should be accessible in constant time. It is also possible to add an element at the beginning of a list, in constant time. The size of a list is not fixed: lists grow in size as more elements are added.

Java has many classes for implementing lists, all subclasses of `java.util.List<E>`. They can be used in Takamaka, but not as fields of a storage class. For that, Takamaka provides an implementation of lists with the storage class `io.takamaka.code.util.StorageLinkedList<E>`. Its instances are storage objects and can consequently be held in fields of storage classes and can be stored in a Hotmoka node, *as long as only storage objects are added to the list*. Takamaka

lists provide constant-time access and addition to both ends of a list. We refer to the JavaDoc of `StorageLinkedList<E>` for a full description of its methods. They include methods for adding elements to either ends of the list, for accessing and removing elements, for iterating on a list and for building a Java array `E[]` holding the elements of a list.

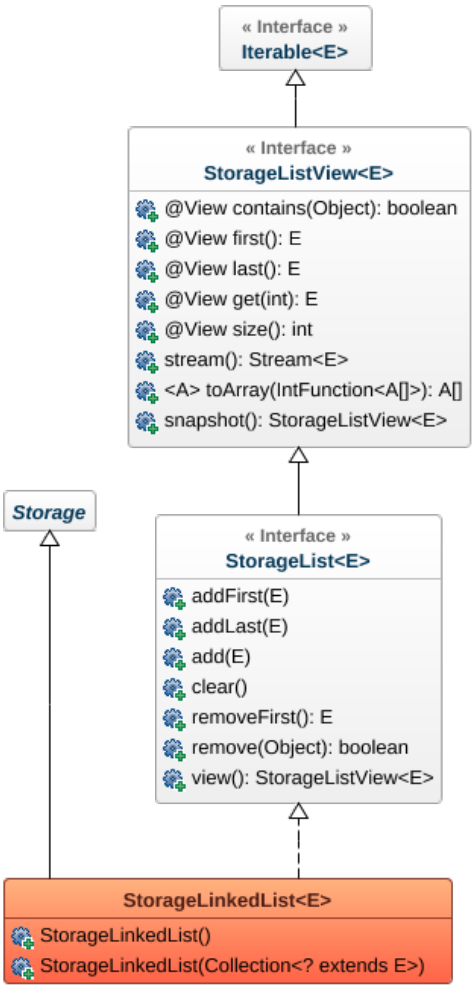



*Figure 24. The hierarchy of storage lists.*

Figure 24 shows the hierarchy of the `StorageLinkedList<E>` class. It implements the interface `StorageList<E>`, that defines the methods that modify a list. That interface extends the interface `StorageListView<E>` that, instead, defines the methods that read data from a list, but do not modify it. This distinction between the *read-only* interface and the *modification* interface is typical of all collection classes in the Takamaka library, as we will see. For the moment, note that this distinction is useful for defining methods `snapshot()` and `view()`. Both return a `StorageListView<E>` but there is an important difference between them. Namely, `snapshot()` yields a *frozen* view of the list, that cannot and will never be modified, also if the original list

gets subsequently updated. Instead, `view()` yields a *view* of a list, that is, a read-only list that changes whenever the original list changes and exactly in the same way: if an element is added to the original list, the same automatically occurs to the view. In this sense, a view is just a read-only alias of the original list. Both methods can be useful to export data, safely, from a node to the outside world, since both methods return an `@Exported` object without modification methods. Method `snapshot()` runs in linear time (in the length of the list) while method `view()` runs in constant time.

> It might seem that `view()` is just an upwards cast to the interface `StorageListView<E>`. This is wrong, since that method does much more. Namely, it applies the façade design pattern to provide a *distinct* list that lacks any modification method and implements a façade of the original list. To appreciate the difference to a cast, assume to have a `StorageList<E> list` and to write `StorageListView<E> view = (StorageListView<E>) list`. This upwards cast will always succeed. Variable `view` does not allow to call any modification method, since they are not in its type `StorageListView<E>`. But a downwards cast back to `StorageList<E>` is enough to circumvent that constraint: `StorageList<E> list2 = (StorageList<E>) view`. This way, the original `list` can be modified by modifying `list2` and it would not be safe to export `view`, since it is a Trojan horse for the modification of `list`. With method `view()`, the problem does not arise, since the cast `StorageList<E> list2 = (StorageList<E>) list.view()` fails: method `view()` actually returns another list object without modification methods. The same is true for method `snapshot()` that, moreover, yields a frozen view of the original list. These same considerations hold for the other Takamaka collections that we will see in this chapter.

Next section shows an example of use for `StorageLinkedList`.

## A Gradual Ponzi Contract

**[See project `ponzi_gradual` inside the `hotmoka_tutorial` repository]**

Consider our previous Ponzi contract again. It is somehow irrealistic, since an investor gets its investment back in full. In a more realistic scenario, the investor will receive the investment back gradually, as soon as new investors arrive. This is more complex to program, since the Ponzi contract must take note of all investors that invested up to now, not just of the current one as in `SimplePonzi.java`. This requires a list of investors, of unbounded size. An implementation of this gradual Ponzi contract is reported below and has been inspired by a similar Ethereum contract from Iyer and Dannen, shown at page 150 of [IyerD08]. Write its code inside package `it.takamaka.ponzi` of the `ponzi` project, as a new class `GradualPonzi.java`:

```
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.util.StorageLinkedList;
import io.takamaka.code.util.StorageList;
```

```
public class GradualPonzi extends Contract {
  public final BigInteger MINIMUM_INVESTMENT = BigInteger.valueOf(1_000L);

  /**
   * All investors up to now. This list might contain the same investor many times,
   * which is important to pay him back more than investors who only invested once.
   */
  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_INVESTMENT) >= 0,
      () -> "you must invest at least " + MINIMUM_INVESTMENT);
    BigInteger eachInvestorGets = amount.divide(BigInteger.valueOf(investors.size()));
    investors.stream().forEachOrdered(investor -> investor.receive(eachInvestorGets));
    investors.add((PayableContract) caller());
  }
}
```

The constructor of `GradualPonzi` is annotated as `@FromContract`, hence it can only be called by a contract, that gets added, as first investor, in the `io.takamaka.code.util.StorageLinkedList` held in field `investors`. This list, that implements an unbounded list of objects, is a storage object, as long as only storage objects are added inside it. `PayableContract`s are storage objects, hence its use is correct here. Subsequently, other contracts can invest by calling method `invest()`. A minimum investment is required, but this remains constant over time. The `amount` invested gets split by the number of the previous investors and sent back to each of them. Note that Takamaka allows programmers to use Java 8 lambdas and streams. Old fashioned Java programmers, who don't feel at home with such treats, can exploit the fact that storage lists are iterable and replace the single-line `forEachOrdered()` call with a more traditional (but gas-hungrier):

```
for (PayableContract investor: investors)
  investor.receive(eachInvestorGets);
```

It is instead **highly discouraged** to iterate the list as if it were an array. Namely, **do not write**

```
for (int pos = 0; pos < investors.size(); pos++)
  investors.get(i).receive(eachInvestorGets);
```

since linked lists are not random-access data structures and the complexity of the last loop is quadratic in the size of the list. This is not a novelty: the same occurs with many traditional Java lists, that do not implement `java.util.RandomAccess` (a notable example is `java.util.LinkedList`). In Takamaka, code execution costs gas and computational complexity does matter, more than in other programming contexts.

## A Note on Re-entrancy

The `GradualPonzi.java` class pays back previous investors immediately: as soon as a new investor invests something, his investment gets split and forwarded to all previous investors. This should make Solidity programmers uncomfortable, since the same approach, in Solidity, might

lead to the infamous re-entrancy attack, when the contract that receives his investment back has a fallback function redefined in such a way to re-enter the paying contract and re-execute the distribution of the investment. As it is well known, such an attack has made some people rich and other desperate. You can find more detail at page 173 of [AntonopoulosW19]. Even if such a frightening scenario does not occur, paying back previous investors immediately is discouraged in Solidity also for other reasons. Namely, the contract that receives his investment back might have a redefined fallback function that consumes too much gas or does not terminate. This would hang the loop that pays back previous investors, actually locking the money inside the `GradualPonzi` contract. Moreover, paying back a contract is a relatively expensive operation in Solidity, even if the fallback function is not redefined, and this cost is paid by the new investor that called `invest()`, in terms of gas. The cost is linear in the number of investors that must be paid back.

As a solution to these problems, Solidity programmers do not pay previous investors back immediately, but let the `GradualPonzi` contract take note of the balance of each investor, through a map. This map is updated as soon as a new investor arrives, by increasing the balance of every previous investor. The cost of updating the balances is still linear in the number of previous investors, but it is cheaper (in Solidity) than sending money back to each of them, which requires costy inter-contract calls that trigger new subtransactions. With this technique, previous investors are now required to withdraw their balance explicitly and voluntarily, through a call to some function, typically called `widthdraw()`. This leads to the *withdrawal pattern*, widely used for writing Solidity contracts.

We have not used the withdrawal pattern in `GradualPonzi.java`. In general, there is no need for such pattern in Takamaka, at least not for simple contracts like `GradualPonzi.java`. The reason is that the `receive()` methods of a payable contract (corresponding to the fallback function of Solidity) are `final` in Takamaka and very cheap in terms of gas. In particular, inter-contract calls are not especially expensive in Takamaka, since they are just a method invocation in Java bytecode (one bytecode instruction). They are *not* inner transactions. They are actually cheaper than updating a map of balances. Moreover, avoiding the `widthdraw()` transactions means reducing the overall number of transactions; without using the map supporting the withdrawal pattern, Takamaka contracts consume less gas and less storage. Hence, the withdrawal pattern is both useless in Takamaka and more expensive than paying back previous contracts immediately.

## Running the Gradual Ponzi Contract

Let us play with the `GradualPonzi` contract now. Run, inside that `ponzi` project, the command `mvn package`. A file `ponzi-0.0.1.jar` should appear inside `target`. We can now start by installing that jar in the node:

```
$ cd hotmoka_tutorial   # if not already there
$ moka install ponzi/target/ponzi-0.0.1.jar
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io

Please specify the password of the payer account: chocolate
Do you really want to spend up to 697300 gas units to install the jar [Y/N] Y

ponzi/target/ponzi-0.0.1.jar has been installed at
64f405c480a8058546d101629819e63463ce7da0e25c6edbb94a4413d99e5c27
```

We create two more accounts now, letting our first account pay:

```
$ moka create-account
    10000000
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io

Please specify the password of the payer account: chocolate
Please specify the password of the new account: orange
Do you really want to spend up to 200000 gas units to create a new account [Y/N] Y
Total gas consumed: 44574
  for CPU: 401
  for RAM: 1352
  for storage: 42821
  for penalty: 0

A new account 26b240580489d5a00e241db547fe2ae756a0209ae87fc6a17e4a06f36f1e7ff0#0
has been created.
Its entropy has been saved into the file
"26b240580489d5a00e241db547fe2ae756a0209ae87fc6a17e4a06f36f1e7ff0#0.pem".
Please take note of the following passphrase of 36 words...

$ moka create-account
    10000000
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io

Please specify the password of the payer account: chocolate
Please specify the password of the new account: apple
Do you really want to spend up to 200000 gas units to create a new account [Y/N] Y
Total gas consumed: 44574
  for CPU: 401
  for RAM: 1352
  for storage: 42821
  for penalty: 0

A new account 48ef7306af5e86adbe01dd7807e1c7bf30fbd3781a63e770245ac72743c5fd1a#0
has been created.
Its entropy has been saved into the file
"48ef7306af5e86adbe01dd7807e1c7bf30fbd3781a63e770245ac72743c5fd1a#0.pem".
Please take note of the following passphrase of 36 words...
```

We let our first account create an instance of `GradualPonzi` in the node now and become the first investor of the contract:

```
$ moka create
    io.takamaka.ponzi.GradualPonzi
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --classpath 64f405c480a8058546d101629819e63463ce7da0e25c6edbb94a4413d99e5c27
    --url panarea.hotmoka.io

Please specify the password of the payer account: chocolate
Do you really want to spend up to 500000 gas units to call
@FromContract(PayableContract.class) public GradualPonzi() ? [Y/N] Y

The new object has been allocated at
84d1edde36aaab618d46742d89647148e413add6d9a77eeb0ddd9130046552ad#0
```

We let the other two players invest, in sequence, in the `GradualPonzi` contract:

```
$ moka call
    84d1edde36aaab618d46742d89647148e413add6d9a77eeb0ddd9130046552ad#0
    invest 5000
    --payer 26b240580489d5a00e241db547fe2ae756a0209ae87fc6a17e4a06f36f1e7ff0#0
    --url panarea.hotmoka.io

Please specify the password of the payer account: orange
Do you really want to spend up to 500000 gas units to call
public void invest(java.math.BigInteger) ? [Y/N] Y

$ moka call
    84d1edde36aaab618d46742d89647148e413add6d9a77eeb0ddd9130046552ad#0
    invest 15000
    --payer 48ef7306af5e86adbe01dd7807e1c7bf30fbd3781a63e770245ac72743c5fd1a#0
    --url panarea.hotmoka.io

Please specify the password of the payer account: apple
Do you really want to spend up to 500000 gas units to call
public void invest(java.math.BigInteger) ? [Y/N] Y
```

We let the first player try to invest again in the contract, this time with a too small investment, which leads to an exception, since the code of the contract requires a minimum investment:

```
$ moka call
    84d1edde36aaab618d46742d89647148e413add6d9a77eeb0ddd9130046552ad#0
    invest 500
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io

Please specify the password of the payer account: chocolate
Do you really want to spend up to 500000 gas units to call
public void invest(java.math.BigInteger) ? [Y/N] Y

io.hotmoka.beans.TransactionException:
io.takamaka.code.lang.RequirementViolationException:
you must invest at least 1000@GradualPonzi.java:46
```

This exception states that a transaction failed because the last investor invested less than 1000 units of coin. Note that the exception message reports the cause (a `require` failed) and includes the source program line of the contract where the exception occurred: line 46 of `GradualPonzi.java`, that is

```
require(amount.compareTo(MINIMUM_INVESTMENT) >= 0,
  () -> "you must invest at least " + MINIMUM_INVESTMENT);
```

Finally, we can check the state of the contract:

```
$ moka state 84d1edde36aaab618d46742d89647148e413add6d9a77eeb0ddd9130046552ad#0
    --url panarea.hotmoka.io

This is the state of object
84d1edde36aaab618d46742d89647148e413add6d9a77eeb0ddd9130046552ad#0
@panarea.hotmoka.io

class io.takamaka.ponzi.GradualPonzi (from jar installed at
    64f405c480a8058546d101629819e63463ce7da0e25c6edbb94a4413d99e5c27)

  MINIMUM_INVESTMENT:java.math.BigInteger = 1000
  investors:io.takamaka.code.util.StorageList
    = 84d1edde36aaab618d46742d89647148e413add6d9a77eeb0ddd9130046552ad#1
  balance:java.math.BigInteger = 0 (inherited from io.takamaka.code.lang.Contract)
  balanceRed:java.math.BigInteger = 0 (inherited from io.takamaka.code.lang.Contract)
```

You can see that the contract keeps no balance. Moreover, its `investors` field is bound to an object, whose state can be further investigated:

```
$ moka state 84d1edde36aaab618d46742d89647148e413add6d9a77eeb0ddd9130046552ad#1
    --url panarea.hotmoka.io

This is the state of object
84d1edde36aaab618d46742d89647148e413add6d9a77eeb0ddd9130046552ad#1
@panarea.hotmoka.io

class io.takamaka.code.util.StorageLinkedList (from jar installed at
    5fd6ae9fe7dbd499621f56814c1f6f1e30718ca9aea69b427dee8c16b9f6c665)

  first:io.takamaka.code.util.StorageLinkedList$Node
    = 84d1edde36aaab618d46742d89647148e413add6d9a77eeb0ddd9130046552ad#2
  last:io.takamaka.code.util.StorageLinkedList$Node
    = 0a314615f98a1e800c0e63628911d6bccd6f5c9c4b99b229319cf4ca4860565a#0
  size:int = 3
```

As you can see, it is a `StorageLinkedList` of size three, since it contains our three accounts that interacted with the `GradualPonzi` contract instance.

## Storage Arrays

Arrays are an ordered sequence of elements, with constant-time access to such elements, both for reading and for writing. The size of the arrays is typically fixed, although there are programming languages with limited forms of dynamic arrays.

Java has native arrays, of type `E[]`, where `E` is the type of the elements of the array. They can be used in Takamaka, but not as fields of storage classes. For that, Takamaka provides class `io.takamaka.code.util.StorageTreeArray<E>`. Its instances are storage objects and can consequently be held in fields of storage classes and can be stored in the store of a Hotmoka node, *as long as only storage objects are added to the array*. Their size is fixed and decided at time of construction. Although we consider `StorageTreeArray<E>` as the storage replacement for Java arrays, it must be stated that the complexity of accessing their elements is logarithmic in the size of the array, which is a significant deviation from the standard definition of arrays. Nevertheless, logarithmic complexity is much better than the linear complexity for accessing elements of a `StorageLinkedList<E>` that, instead, has the advantage of being dynamic in size.

We refer to the JavaDoc of `StorageTreeArray<E>` for a full list of its methods. They include methods for adding elements, for accessing and removing elements, for iterating on an array and for building a Java array `E[]` with the elements of a `StorageTreeArray<E>`. Figure 25 shows the hierarchy of the `StorageTreeArray<E>` class. It implements the interface `StorageArray<E>`, that defines the methods that modify an array. That interface extends the interface `StorageArrayView<E>` that, instead, defines the methods that read data from an array, but do not modify it. This distinction between the *read-only* interface and the *modification* interface is identical to what we have seen for lists in the previous sections. Arrays have methods `snapshot()` and `view()` as well, like lists. They yield `@Exported` storage arrays, both in contants time. All constructors of the `StorageTreeArray<E>` class require to specify the immutable size of the array. Moreover, it is possible to specify a default value for the elements of the array, that can be explicit or given as a supplier, possibly indexed.

Next section shows an example of use for `StorageTreeArray<E>`.


*Figure 25. The hierarchy of storage arrays.*

## A Tic-Tac-Toe Contract

**[See project `tictactoe` inside the `hotmoka_tutorial` repository]**

Tic-tac-toe is a game where two players place, alternately, a cross and a circle on a 3x3 board, initially empty. The winner is the player who places three crosses or three circles on the same row, column or diagonal. For instance, in Figure 26 the player of the cross wins.

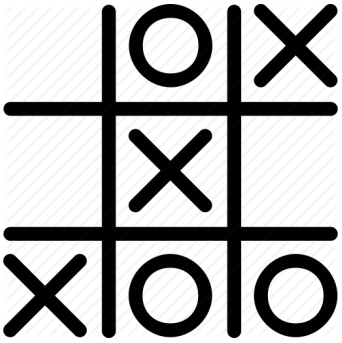


*Figure 26. Cross wins.*

There are games that end up in a draw, when the board is full but nobody wins, as in Figure 27.

*Figure 27. A draw.*

A natural representation of the tic-tac-toe board is a bidimensional array where indexes are distributed as shown in Figure 28.

This can be implemented as a `StorageTreeArray<StorageTreeArray<Tile>>`, where `Tile` is an enumeration of the three possible tiles (empty, cross, circle). This is possible but overkill. It is simpler and cheaper (also in terms of gas) to use the previous diagram as a conceptual representation of the board shown to the users, but use, internally, a monodimensional array of nine tiles, distributed as in Figure 29. This monodimensional array can be implemented as a `StorageTreeArray<Tile>`. There will be functions for translating the conceptual representation into the internal one.

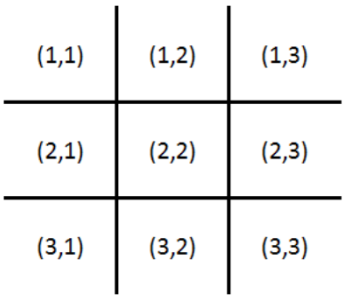



*Figure 28. A bidimensional representation of the game.*

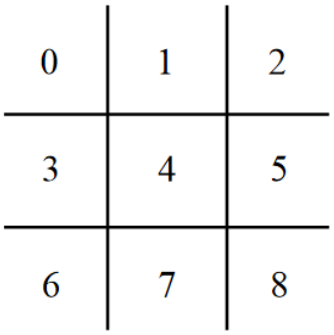



*Figure 29. A linear representation of the game.*

Create hence in Eclipse a new Maven Java 11 (or later) project named `tictactoe`. You can do this by duplicating the project `family` (make sure to store the project inside the `hotmoka_tutorial` directory, as a sibling of `family`, `ponzi` and `runs`). Use the following `pom.xml`:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
    http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>io.hotmoka</groupId>
  <artifactId>tictactoe</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>1.0.7</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>
```

and the following `module-info.java`:

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

Create package `io.takamaka.tictactoe` inside `src/main/java` and add the following `TicTacToe.java` source inside that package:

```
package io.takamaka.tictactoe;

import static io.takamaka.code.lang.Takamaka.require;
import static java.util.stream.Collectors.joining;
import static java.util.stream.IntStream.rangeClosed;

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;
import io.takamaka.code.util.StorageArray;
import io.takamaka.code.util.StorageTreeArray;

public class TicTacToe extends Contract {

  public enum Tile {
    EMPTY, CROSS, CIRCLE;

    @Override
    public String toString() {
      switch (this) {
      case EMPTY: return " ";
```

```
    case CROSS: return "X";
    default: return "O";
    }
  }

  private Tile nextTurn() {
    return this == CROSS ? CIRCLE : CROSS;
  }
}

private final StorageArray<Tile> board = new StorageTreeArray<>(9, Tile.EMPTY);
private PayableContract crossPlayer, circlePlayer;
private Tile turn = Tile.CROSS; // cross plays first
private boolean gameOver;

public @View Tile at(int x, int y) {
  require(1 <= x && x <= 3 && 1 <= y && y <= 3,
    "coordinates must be between 1 and 3");
  return board.get((y - 1) * 3 + x - 1);
}

private void set(int x, int y, Tile tile) {
  board.set((y - 1) * 3 + x - 1, tile);
}

public @Payable @FromContract(PayableContract.class)
    void play(long amount, int x, int y) {

  require(!gameOver, "the game is over");
  require(1 <= x && x <= 3 && 1 <= y && y <= 3,
    "coordinates must be between 1 and 3");
  require(at(x, y) == Tile.EMPTY, "the selected tile is not empty");

  PayableContract player = (PayableContract) caller();

  if (turn == Tile.CROSS)
    if (crossPlayer == null)
      crossPlayer = player;
    else
      require(player == crossPlayer, "it's not your turn");
  else
    if (circlePlayer == null) {
      require(crossPlayer != player, "you cannot play against yourself");
      long previousBet = balance().subtract(BigInteger.valueOf(amount)).longValue();
      require(amount >= previousBet,
        () -> "you must bet at least " + previousBet + " coins");
      circlePlayer = player;
    }
    else
      require(player == circlePlayer, "it's not your turn");

  set(x, y, turn);
  if (isGameOver(x, y))
```

```
      player.receive(balance());
    else
      turn = turn.nextTurn();
  }

  private boolean isGameOver(int x, int y) {
    return gameOver =
      rangeClosed(1, 3).allMatch(_y -> at(x, _y) == turn) || // column x
      rangeClosed(1, 3).allMatch(_x -> at(_x, y) == turn) || // row y
      (x == y && rangeClosed(1, 3).allMatch(_x -> at(_x, _x) == turn)) || // 1st diagonal
      (x + y == 4 && rangeClosed(1, 3).allMatch(_x -> at(_x, 4 - _x) == turn)); // 2nd
  }

  @Override
  public @View String toString() {
    return rangeClosed(1, 3)
      .mapToObj(y -> rangeClosed(1, 3)
                     .mapToObj(x -> at(x, y).toString())
                     .collect(joining("|")))
      .collect(joining("\n-----\n"));
  }
}
```

The internal enumeration `Tile` represents the three alternatives that can be put in the tic-tac-toe board. It overrides the default `toString()` implementation, to yield the usual representation for such alternatives; its `nextTurn()` method alternates between cross and circle.

> There is no need to make the `Tile` enumeration `static`, to save gas, since enumerations are always implicitly `static` in Java.

The board of the game is represented as a `new StorageTreeArray<>(9, Tile.EMPTY)`, whose elements are indexed from 0 to 8 (inclusive) and are initialized to `Tile.EMPTY`. It is also possible to construct the array as `new StorageTreeArray<>(9)`, but then its elements would hold the default value `null` and the array would need to be initialized inside a constructor for `TicTacToe`:

```
public TicTacToe() {
  rangeClosed(0, 8).forEachOrdered(index -> board.set(index, Tile.EMPTY));
}
```

Methods `at()` and `set()` read and set the board element at indexes (x,y), respectively. They transform the bidimensional conceptual representation of the board into its internal monodimensional representation. Since `at()` is `public`, we defensively check the validity of the indexes there.

Method `play()` is the heart of the contract. It is called by the accounts that play the game, hence it is annotated as `@FromContract`. It is also annotated as `@Payable(PayableContract.class)` since players must bet money for taking part in the game, at least for the first two moves, and receive money if they win. The first contract that plays is registered as `crossPlayer`. The second contract that plays is registered as `circlePlayer`. Subsequent moves must come, alternately, from `crossPlayer` and `circlePlayer`. The contract uses a `turn` variable to keep track of the current turn.

Note the extensive use of `require()` to check all error situations:

1. It is possible to play only if the game is not over yet.
2. A move must be inside the board and identify an empty tile.
3. Players must alternate correctly.
4. The second player must bet at least as much as the first player.
5. It is not allowed to play against oneself.

The `play()` method ends with a call to `gameOver()` that checks if the game is over. In that case, the winner receives the full jackpot. Note that the `gameOver()` method receives the coordinates where the current player has moved. This allows it to restrict the check for game over: the game is over only if the row or column where the player moved contain the same tile; if the current player played on a diagonal, the method checks the diagonals as well. It is of course possible to check all rows, columns and diagonals, always, but our solution is gas-thriftier.

The `toString()` method yields a string representation of the current board, such as

```
X|O|
-----
 |X|O
-----
 |X|
```

For those who do not appreciate Java 8 streams, the same result can be obtained with a more traditional (and gas-hungrier) code:

```
@Override
public @View String toString() {
  String result = "";
  for (int y = 0; y < 3; y++) {
    for (int x = 0; x < 3; x++) {
      result += at(x, y);
      if (x < 2)
        result += "|";
    }
    if (y < 2)
      result += "\n-----\n"
  }

  return result;
}
```

## A More Realistic Tic-Tac-Toe Contract

**[See project `tictactoe_improved` inside the `hotmoka_tutorial` repository]**

The `TicTacToe.java` code implements the rules of a tic-tac-toe game, but has a couple of drawbacks that make it still incomplete. Namely:

1. The creator of the game must spend gas to call its constructor, but has no direct incentive in doing so. He must be a benefactor, or hope to take part in the game after creation, if he is faster than any other potential player.

2. If the game ends in a draw, money gets stuck in the `TicTacToe` contract instance, for ever and ever.

Replace hence the previous version of `TicTacToe.java` with the following improved version. This new version solves both problems at once. The policy is very simple: it imposes a minimum bet, in order to avoid free games; if a winner emerges, then the game forwards him only 90% of the jackpot; the remaing 10% goes to the creator of the `TicTacToe` contract. If, instead, the game ends in a draw, it forwards the whole jackpot to the creator. Note that we added a `@FromContract` constructor, that takes note of the `creator` of the game:

```
package io.takamaka.tictactoe;

import static io.takamaka.code.lang.Takamaka.require;
import static java.util.stream.Collectors.joining;
import static java.util.stream.IntStream.rangeClosed;

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;
import io.takamaka.code.util.StorageArray;
import io.takamaka.code.util.StorageTreeArray;

public class TicTacToe extends Contract {

  public enum Tile {
    EMPTY, CROSS, CIRCLE;

    @Override
    public String toString() {
      switch (this) {
      case EMPTY: return " ";
      case CROSS: return "X";
      default: return "O";
      }
    }

    private Tile nextTurn() {
      return this == CROSS ? CIRCLE : CROSS;
    }
  }

  private final static long MINIMUM_BET = 100L;

  private final StorageArray<Tile> board = new StorageTreeArray<>(9, Tile.EMPTY);
  private final PayableContract creator;
  private PayableContract crossPlayer, circlePlayer;
  private Tile turn = Tile.CROSS; // cross plays first
  private boolean gameOver;

  public @FromContract(PayableContract.class) TicTacToe() {
```

```
  creator = (PayableContract) caller();
}

public @View Tile at(int x, int y) {
  require(1 <= x && x <= 3 && 1 <= y && y <= 3,
    "coordinates must be between 1 and 3");
  return board.get((y - 1) * 3 + x - 1);
}

private void set(int x, int y, Tile tile) {
  board.set((y - 1) * 3 + x - 1, tile);
}

public @Payable @FromContract(PayableContract.class)
    void play(long amount, int x, int y) {

  require(!gameOver, "the game is over");
  require(1 <= x && x <= 3 && 1 <= y && y <= 3,
    "coordinates must be between 1 and 3");
  require(at(x, y) == Tile.EMPTY, "the selected tile is not empty");

  PayableContract player = (PayableContract) caller();

  if (turn == Tile.CROSS)
    if (crossPlayer == null) {
      require(amount >= MINIMUM_BET,
        () -> "you must bet at least " + MINIMUM_BET + " coins");
      crossPlayer = player;
    }
    else
      require(player == crossPlayer, "it's not your turn");
  else
    if (circlePlayer == null) {
      require(crossPlayer != player, "you cannot play against yourself");
      long previousBet = balance().subtract(BigInteger.valueOf(amount)).longValue();
      require(amount >= previousBet,
        () -> "you must bet at least " + previousBet + " coins");
      circlePlayer = player;
  }
  else
    require(player == circlePlayer, "it's not your turn");

  set(x, y, turn);
  if (isGameOver(x, y)) {
    // 90% goes to the winner
    player.receive(balance().multiply(BigInteger.valueOf(9L))
                            .divide(BigInteger.valueOf(10L)));
    // the rest goes to the creator of the game
    creator.receive(balance());
  }
  else if (isDraw())
    // everything goes to the creator of the game
    creator.receive(balance());
```

```
    else
      turn = turn.nextTurn();
  }

  private boolean isGameOver(int x, int y) {
    return gameOver =
      rangeClosed(1, 3).allMatch(_y -> at(x, _y) == turn) || // column x
      rangeClosed(1, 3).allMatch(_x -> at(_x, y) == turn) || // row y
      (x == y && rangeClosed(1, 3).allMatch(_x -> at(_x, _x) == turn)) || // 1st diagonal
      (x + y == 4 && rangeClosed(1, 3).allMatch(_x -> at(_x, 4 - _x) == turn)); // 2nd
  }

  private boolean isDraw() {
    return rangeClosed(0, 8).mapToObj(board::get).noneMatch(Tile.EMPTY::equals);
  }

  @Override
  public @View String toString() {
    return rangeClosed(1, 3)
      .mapToObj(y -> rangeClosed(1, 3)
                     .mapToObj(x -> at(x, y).toString())
                     .collect(joining("|")))
      .collect(joining("\n-----\n"));
  }
}
```

We have chosen to allow a `long amount` in the `@Payable` method `play()` since it is unlikely that users will want to invest huge quantities of money in this game. This gives us the opportunity to discuss why the computation of the previous bet has been written as `long previousBet = balance().subtract(BigInteger.valueOf(amount)).longValue()` instead of the simpler `long previousBet = balance().longValue() - amount`. The reason is that, when that line is executed, both players have aleady paid their bet, that accumulates in the balance of the `TicTacToe` contract. Each single bet is a `long`, but their sum could overflow the size of a `long`. Hence, we have to deal with a computation on `BigInteger`. The same situation occurs later, when we have to compute the 90% that goes to the winner: the jackpot might be larger than a `long` and we have to compute over `BigInteger`. As a final remark, note that in the line: `balance().multiply(BigInteger.valueOf(9L)).divide(BigInteger.valueOf(10L))` we first multiply by 9 and **then** divide by 10. This reduces the approximation inherent to integer division. For instance, if the jackpot (`balance()`) were 209, we have (with Java's left-to-right evaluation) `209*9/10=1881/10=188` while `209/10*9=20*9=180`.

## Running the Tic-Tac-Toe Contract

Let us play with the `TicTacToe` contract. Go inside the `tictactoe` project and run the `mvn package` command. A file `tictactoe-0.0.1.jar` should appear inside `target`. Let us start by installing that jar in the node:

```
$ moka install tictactoe/target/tictactoe-0.0.1.jar
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io

Please specify the password of the payer account: chocolate
Do you really want to spend up to 870600 gas units to install the jar [Y/N] Y

tictactoe/target/tictactoe-0.0.1.jar has been installed
at e74e78ee4ee2cfa8a14c5a77f2112484702e225143468bc0c7ed3bf2ed4f8a11
```

Then we create an instance of the contract in the node:

```
$ moka create
    io.takamaka.tictactoe.TicTacToe
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --classpath e74e78ee4ee2cfa8a14c5a77f2112484702e225143468bc0c7ed3bf2ed4f8a11
    --url panarea.hotmoka.io

Please specify the password of the payer account: chocolate
Do you really want to spend up to 500000 gas units to call
@FromContract(PayableContract.class) public TicTacToe() ? [Y/N] Y

The new object has been allocated at
702da7b9404d8391cca09a0dcc46250af711eeee8e22de43387284404565e957#0
```

We use two of our accounts now, that we have already created in the previous section, to interact with the contract: they will play, alternately, until the first player wins. We will print the `toString` of the contract after each move.

The first player starts, by playing at (1,1), and bets 100:

```
$ moka call
    702da7b9404d8391cca09a0dcc46250af711eeee8e22de43387284404565e957#0
    play 100 1 1
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io

$ moka call
    702da7b9404d8391cca09a0dcc46250af711eeee8e22de43387284404565e957#0
    toString
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io

X| |
-----
 | |
-----
 | |
```

The second player plays after, at (2,1), betting 100:

```
$ moka call
    702da7b9404d8391cca09a0dcc46250af711eeee8e22de43387284404565e957#0
    play 100 2 1
    --payer 26b240580489d5a00e241db547fe2ae756a0209ae87fc6a17e4a06f36f1e7ff0#0
    --url panarea.hotmoka.io

$ moka call
    702da7b9404d8391cca09a0dcc46250af711eeee8e22de43387284404565e957#0
    toString
    --payer 26b240580489d5a00e241db547fe2ae756a0209ae87fc6a17e4a06f36f1e7ff0#0
    --url panarea.hotmoka.io

X|O|
-----
 | |
-----
 | |
```

The first player replies, playing at (1,2):

```
$ moka call
    702da7b9404d8391cca09a0dcc46250af711eeee8e22de43387284404565e957#0
    play 0 1 2
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io

$ moka call
    702da7b9404d8391cca09a0dcc46250af711eeee8e22de43387284404565e957#0
    toString
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io

X|O|
-----
X| |
-----
 | |
```

Then the second player plays at (2,2):

```
$ moka call
    702da7b9404d8391cca09a0dcc46250af711eeee8e22de43387284404565e957#0
    play 0 2 2
    --payer 26b240580489d5a00e241db547fe2ae756a0209ae87fc6a17e4a06f36f1e7ff0#0
    --url panarea.hotmoka.io

$ moka call
    702da7b9404d8391cca09a0dcc46250af711eeee8e22de43387284404565e957#0
    toString
    --payer 26b240580489d5a00e241db547fe2ae756a0209ae87fc6a17e4a06f36f1e7ff0#0
    --url panarea.hotmoka.io

X|O|
-----
X|O|
-----
 | |
```

The first player wins by playing at (1,3):

```
$ moka call
    702da7b9404d8391cca09a0dcc46250af711eeee8e22de43387284404565e957#0
    play 0 1 3
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io

$ moka call
    702da7b9404d8391cca09a0dcc46250af711eeee8e22de43387284404565e957#0
    toString
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io

X|O|
-----
X|O|
-----
X| |
```

We can verify that the game is over now:

```
$ moka state 702da7b9404d8391cca09a0dcc46250af711eeee8e22de43387284404565e957#0
    --url panarea.hotmoka.io

This is the state of object
702da7b9404d8391cca09a0dcc46250af711eeee8e22de43387284404565e957#0
@panarea.hotmoka.io

class io.takamaka.tictactoe.TicTacToe (from jar installed at
    e74e78ee4ee2cfa8a14c5a77f2112484702e225143468bc0c7ed3bf2ed4f8a11)

  ...
  gameOver:boolean = true
  balance:java.math.BigInteger = 0 (inherited from io.takamaka.code.lang.Contract)
  ...
```

As you can see, the balance of the contract is zero since it has been distributed to the winner and to the creator of the game (that actually coincide, in this specific run).

If the second player attempts to play now, the transaction will be rejected, since the game is over:

```
$ moka call
    702da7b9404d8391cca09a0dcc46250af711eeee8e22de43387284404565e957#0
    play 0 2 3
    --payer 26b240580489d5a00e241db547fe2ae756a0209ae87fc6a17e4a06f36f1e7ff0#0
    --url panarea.hotmoka.io

io.hotmoka.beans.TransactionException:
io.takamaka.code.lang.RequirementViolationException:
the game is over@TicTacToe.java:79
```

## Specialized Storage Array Classes

The `StorageTreeArray<E>` class is very general, since it can be used to hold any type `E` of storage values. Since it uses generics, primitive values cannot be held in a `StorageTreeArray<E>`, directly. For instance, `StorageTreeArray<byte>` is not legal syntax in Java. Instead, one could think to use `StorageTreeArray<Byte>`, where `Byte` is the Java wrapper class `java.lang.Byte`. However, that class is not currently allowed in storage, hence `StorageTreeArray<Byte>` will not work either. One should hence define a new wrapper class for `byte`, that extends `Storage`. That is possible, but highly discouraged: the use of wrapper classes introduces a level of indirection and requires the instantiation of many small objects, which costs gas. Instead, Takamaka provides specialized storage classes implementing arrays of bytes, without wrappers. The rationale is that such arrays arise naturally when dealing, for instance, with hashes or encrypted data (see next section for an example) and consequently deserve a specialized and optimized implementation. Such specialized array classes can have their length specified at construction time, or fixed to a constant (for best optimization and minimal gas consumption).

Figure 30 shows the hierarchy of the specialized classes for arrays of bytes, available in Takamaka. The interface `StorageByteArrayView` defines the methods that read data from an array of bytes, while the interface `StorageByteArray` defines the modification methods. Class `StorageTreeByteArray` allows one to create byte arrays of any length, specified at construction

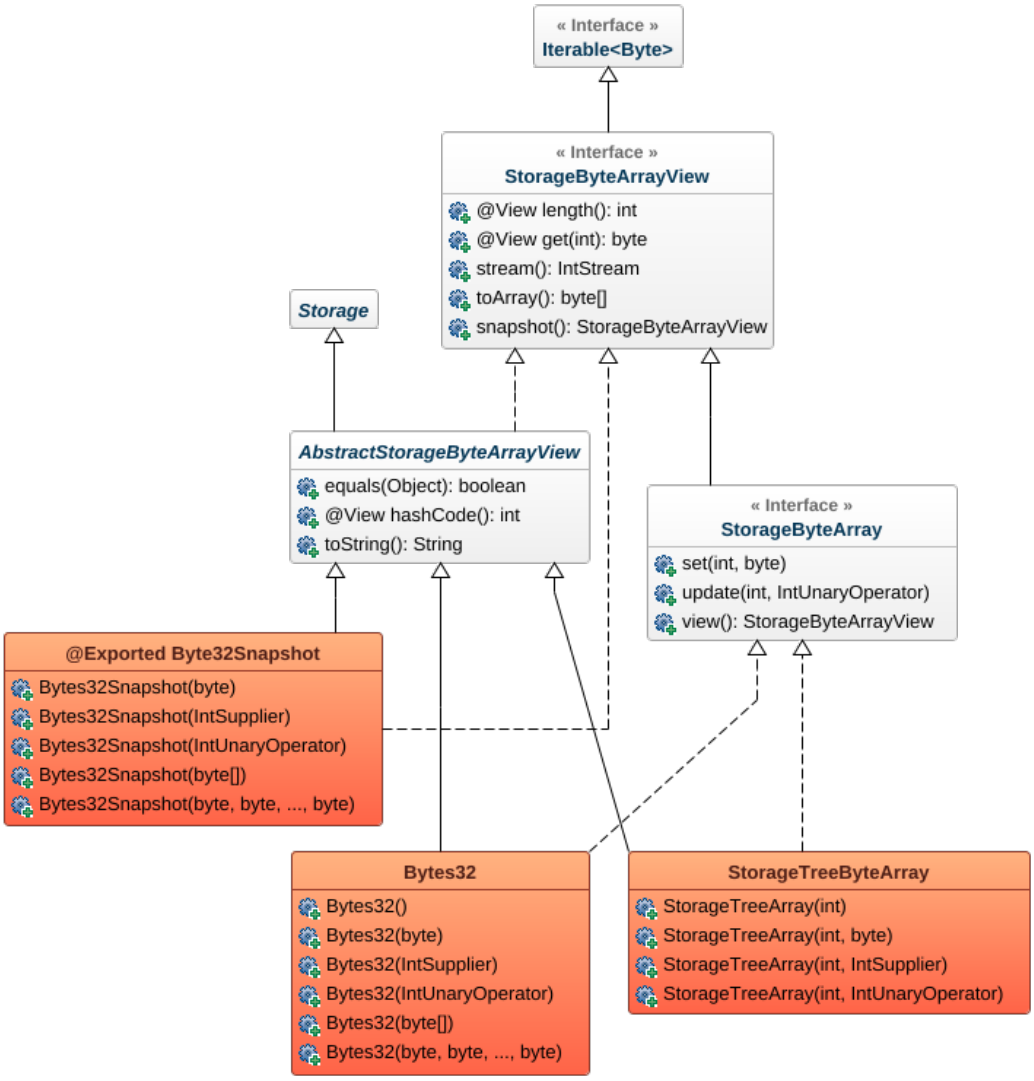



*Figure 30. Specialized byte array classes.*

time. Classes `Bytes32` and `Bytes32Snapshot` have, instead, fixed length of 32 bytes; their constructors include one that allows one to specify such 32 bytes, which is useful for calling the constructor from outside the node, since `byte` is a storage type. While a `Bytes32` is modifiable, instances of class `Bytes32Snapshot` are not modifiable after being created and are `@Exported`. There are sibling classes for different, fixed sizes, such as `Bytes64` and `Bytes8Snaphot`. For a full description of the methods of these classes and interfaces, we refer to their JavaDoc.

## Storage Maps

Maps are dynamic associations of objects to objects. They are useful for programming smart contracts, as their extensive use in Solidity proves. However, most such uses are related to the withdrawal pattern, that is not needed in Takamaka. Nevertheless, there are still situations when maps are useful in Takamaka code, as we show below.

Java has many implementations of maps, that can be used in Takamaka. However, they are not storage objects and consequently cannot be stored in a Hotmoka node. This section describes the `io.takamaka.code.util.StorageTreeMap<K,V>` class, that extends `Storage` and whose instances can then be held in the store of a node, if keys `K` and values `V` can be stored in a node as well.

We refer to the JavaDoc of `StorageTreeMap` for a full description of its methods, that are similar to those of traditional Java maps. Here, we just observe that a key is mapped into a value by calling method `void put(K key, V value)`, while the value bound to a key is retrieved by calling `V get(Object key)`. It is possible to yield a default value when a key is not in the map, by calling `V getOrDefault(Object key, V _default)` or its sibling `V getOrDefault(Object key, Supplier<? extends V> _default)`, that evaluates the default value only if needed. Method `V putIfAbsent(K key, V value)`, binds the key to the value only if the key is unbound. Similarly for its sibling `V computeIfAbsent(K key, Supplier<? extends V> value)` that, however, evaluates the new value only if needed (these two methods differ for their returned value, as in Java maps. Please refer to their JavaDoc).

Instances of `StorageTreeMap<K,V>` keep keys in increasing order. Namely, if type `K` has a natural order, that order is used. Otherwise, keys (that must be storage objects) are kept ordered by increasing storage reference. Consequently, methods `K min()` and `K max()` yield the minimal and the maximal key of a map. Method `List<K> keyList()` yields the ordered list of the keys of a map; method `Stream<K> keys()` yields the same, as an ordered stream; method `Stream<Entry<K,V>> stream()` yields the ordered stream of the entries (ie., key/value pairs) of a map; method `Stream<V> values()` yields the ordered stream of the values bound to the keys of the map.

> Compare this with Solidity, where maps do not know the set of their keys nor the set of their values.

Figure 31 shows the hierarchy of the `StorageTreeMap<K,V>` class. It implements the interface `StorageMap<K,V>`, that defines the methods that modify a map. That interface extends the interface `StorageMapView<K,V>` that, instead, defines the methods that read data from a map, but do not modify it. Methods `snapshot()` and `view()` return an `@Exported StorageMapView<K,V>`, in constant time.

There are also specialized map classes, optimized for specific primitive types of keys, such as

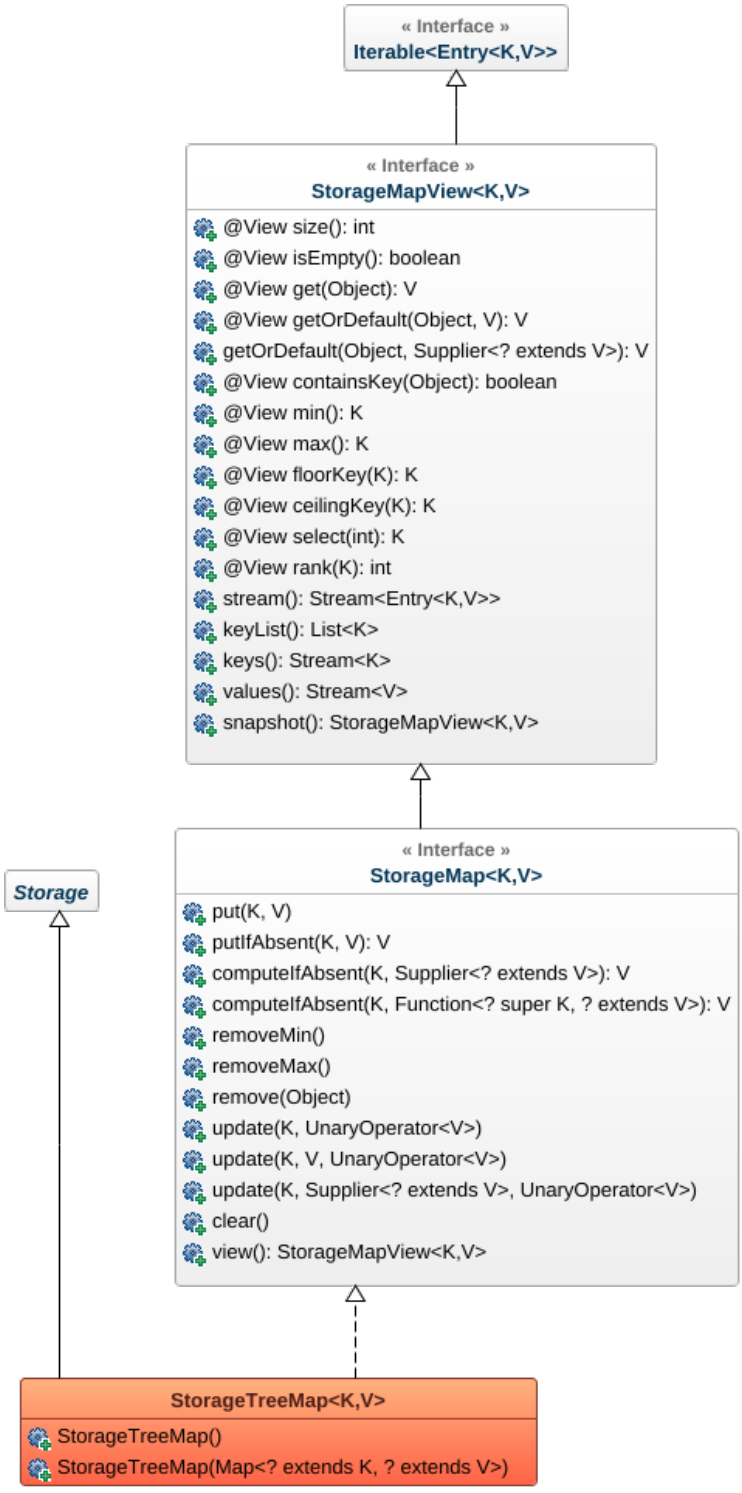



*Figure 31. The hierarchy of storage maps.*

`StorageTreeIntMap<V>`, whose keys are `int` values. We refer to their JavaDoc for further information.

## A Blind Auction Contract

**[See project `auction` inside the `hotmoka_tutorial` repository]**

This section exemplifies the use of class `StorageTreeMap` for writing a smart contract that implements a *blind auction*. That contract allows a *beneficiary* to sell an item to the buying contract that offers the highest bid. Since data in blockchain is public, in a non-blind auction it is possible that bidders eavesdrop the offers of other bidders in order to place an offer that is only slightly higher than the current best offer. A blind auction, instead, uses a two-phases mechanism: in the initial *bidding time*, bidders place bids, hashed, so that they do not reveal their amount. After the bidding time expires, the second phase, called *reveal time*, allows bidders to reveal the real values of their bids and the auction contract to determine the actual winner. This works since, to reveal a bid, each bidder provides the real data of the bid. The auction contract then recomputes the hash from real data and checks if the result matches the hash provided at bidding time. If not, the bid is considered invalid. Bidders can even place fake offers on purpose, in order to confuse other bidders.

Create in Eclipse a new Maven Java 11 (or later) project named `auction`. You can do this by duplicating the project `family` (make sure to store the project inside the `hotmoka_tutorial` directory, as a sibling of `family`, `ponzi`, `tictactoe` and `runs`). Use the following `pom.xml`:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>io.hotmoka</groupId>
  <artifactId>auction</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>1.0.7</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>
```

and the following `module-info.java`:

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

Create package `io.takamaka.auction` inside `src/main/java` and add the following `BlindAuction.java` inside that package. It is a Takamaka contract that implements a blind auction. Since each bidder may place more bids and since such bids must be kept in storage until reveal time, this code uses a map from bidders to lists of bids. This smart contract has been inspired by a similar Solidity contract [BlindAuction]. Please note that this code will not compile yet, since it misses two classes that we will define in the next section.

```
package io.takamaka.auction;

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

import java.math.BigInteger;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Arrays;
import java.util.function.Supplier;

import io.takamaka.code.lang.Contract;
import io.takamaka.code.lang.Exported;
import io.takamaka.code.lang.FromContract;
import io.takamaka.code.lang.Payable;
import io.takamaka.code.lang.PayableContract;
import io.takamaka.code.lang.Storage;
import io.takamaka.code.util.Bytes32Snapshot;
import io.takamaka.code.util.StorageLinkedList;
import io.takamaka.code.util.StorageList;
import io.takamaka.code.util.StorageMap;
import io.takamaka.code.util.StorageTreeMap;

public class BlindAuction extends Contract {

  /**
   * A bid placed by a bidder. The deposit has been paid in full.
   * If, later, the bid will be revealed as fake, then the deposit will
   * be fully refunded. If, instead, the bid will be revealed as real, but for
```

```
 * a lower amount, then only the difference will be refunded.
 */
private static class Bid extends Storage {

  /**
   * The hash that will be regenerated and compared at reveal time.
   */
  private final Bytes32Snapshot hash;

  /**
    * The value of the bid. Its real value might be lower and known
    * at real time only.
    */
  private final BigInteger deposit;

  private Bid(Bytes32Snapshot hash, BigInteger deposit) {
    this.hash = hash;
    this.deposit = deposit;
  }

  /**
   * Recomputes the hash of a bid at reveal time and compares it
   * against the hash provided at bidding time. If they match,
   * we can reasonably trust the bid.
   *
   * @param revealed the revealed bid
   * @param digest the hasher
   * @return true if and only if the hashes match
   */
  private boolean matches(RevealedBid revealed, MessageDigest digest) {
    digest.update(revealed.value.toByteArray());
    digest.update(revealed.fake ? (byte) 0 : (byte) 1);
    digest.update(revealed.salt.toArray());
    return Arrays.equals(hash.toArray(), digest.digest());
  }
}

/**
 * A bid revealed by a bidder at reveal time. The bidder shows
 * if the corresponding bid was fake or real, and how much was the
 * actual value of the bid. This might be lower than previously communicated.
 */
@Exported
public static class RevealedBid extends Storage {
  private final BigInteger value;
  private final boolean fake;

  /**
   * The salt used to strengthen the hashing.
   */
  private final Bytes32Snapshot salt;

  public RevealedBid(BigInteger value, boolean fake, Bytes32Snapshot salt) {
```

```
    this.value = value;
    this.fake = fake;
    this.salt = salt;
  }
}

/**
 * The beneficiary that, at the end of the reveal time, will receive the highest bid.
 */
private final PayableContract beneficiary;

/**
 * The bids for each bidder. A bidder might place more bids.
 */
private final StorageMap<PayableContract, StorageList<Bid>> bids = new StorageTreeMap<>();

/**
 * The time when the bidding time ends.
 */
private final long biddingEnd;

/**
 * The time when the reveal time ends.
 */
private final long revealEnd;

/**
 * The bidder with the highest bid, at reveal time.
 */
private PayableContract highestBidder;

/**
 * The highest bid, at reveal time.
 */
private BigInteger highestBid;

/**
 * Creates a blind auction contract.
 *
 * @param biddingTime the length of the bidding time
 * @param revealTime the length of the reveal time
 */
public @FromContract(PayableContract.class) BlindAuction(int biddingTime, int revealTime) {
  require(biddingTime > 0, "Bidding time must be positive");
  require(revealTime > 0, "Reveal time must be positive");

  this.beneficiary = (PayableContract) caller();
  this.biddingEnd = now() + biddingTime;
  this.revealEnd = biddingEnd + revealTime;
}

/**
 * Places a blinded bid with the given hash.
```

```
 * The money sent is only refunded if the bid is correctly
 * revealed in the revealing phase. The bid is valid if the
 * money sent together with the bid is at least "value" and
 * "fake" is not true. Setting "fake" to true and sending
 * not the exact amount are ways to hide the real bid but
 * still make the required deposit. The same bidder can place multiple bids.
 */
public @Payable @FromContract(PayableContract.class) void bid
    (BigInteger amount, Bytes32Snapshot hash) {

  onlyBefore(biddingEnd);
  bids.computeIfAbsent((PayableContract) caller(), (Supplier<StorageList<Bid>>) StorageLinkedList::new)
    .add(new Bid(hash, amount));
}

/**
 * Reveals a bid of the caller. The caller will get a refund for all correctly
 * blinded invalid bids and for all bids except for the totally highest.
 *
 * @param revealed the revealed bid
 * @throws NoSuchAlgorithmException if the hashing algorithm is not available
 */
public @FromContract(PayableContract.class) void reveal
    (RevealedBid revealed) throws NoSuchAlgorithmException {

  onlyAfter(biddingEnd);
  onlyBefore(revealEnd);
  PayableContract bidder = (PayableContract) caller();
  StorageList<Bid> bids = this.bids.get(bidder);
  require(bids != null && bids.size() > 0, "No bids to reveal");
  require(revealed != null, () -> "The revealed bid cannot be null");

  // any other hashing algorithm will do, as long as
  // both bidder and auction contract use the same
  MessageDigest digest = MessageDigest.getInstance("SHA-256");
  // by removing the head of the list, it makes it impossible
  // for the caller to re-claim the same deposits
  bidder.receive(refundFor(bidder, bids.removeFirst(), revealed, digest));
}

/**
 * Ends the auction and sends the highest bid to the beneficiary.
 *
 * @return the highest bidder
 */
public PayableContract auctionEnd() {
  onlyAfter(revealEnd);
  PayableContract winner = highestBidder;

  if (winner != null) {
    beneficiary.receive(highestBid);
    event(new AuctionEnd(winner, highestBid));
    highestBidder = null;
```

```
  }

  return winner;
}

/**
 * Checks how much of the deposit should be refunded for a given bid.
 *
 * @param bidder the bidder that placed the bid
 * @param bid the bid, as was placed at bidding time
 * @param revealed the bid, as was revealed later
 * @param digest the hashing algorithm
 * @return the amount to refund
 */
private BigInteger refundFor(PayableContract bidder, Bid bid,
    RevealedBid revealed, MessageDigest digest) {

  if (!bid.matches(revealed, digest))
    // the bid was not actually revealed: no refund
    return BigInteger.ZERO;
  else if (!revealed.fake && bid.deposit.compareTo(revealed.value) >= 0
      && placeBid(bidder, revealed.value))
    // the bid was correctly revealed and is the best up to now:
    // only the difference between promised and provided is refunded;
    // the rest might be refunded later if a better bid will be revealed
    return bid.deposit.subtract(revealed.value);
  else
    // the bid was correctly revealed and is not the best one:
    // it is fully refunded
    return bid.deposit;
}

/**
 * Takes note that a bidder has correctly revealed a bid for the given value.
 *
 * @param bidder the bidder
 * @param value the value, as revealed
 * @return true if and only if this is the best bid, up to now
 */
private boolean placeBid(PayableContract bidder, BigInteger value) {
  if (highestBid != null && value.compareTo(highestBid) <= 0)
    // this is not the best bid seen so far
    return false;

  // if there was a best bidder already, its bid is refunded
  if (highestBidder != null)
    // Refund the previously highest bidder
    highestBidder.receive(highestBid);

  // take note that this is the best bid up to now
  highestBid = value;
  highestBidder = bidder;
  event(new BidIncrease(bidder, value));
```

```
    return true;
  }

  private static void onlyBefore(long when) {
    require(now() < when, "Too late");
  }

  private static void onlyAfter(long when) {
    require(now() > when, "Too early");
  }
}
```

Let us discuss this (long) code, by starting from the inner classes.

Class `Bid` represents a bid placed by a contract that takes part to the auction. This information will be stored in blockchain at bidding time, hence it is known to all other participants. An instance of `Bid` contains the `deposit` paid at time of placing the bid. This is not necessarily the real value of the offer but must be at least as large as the real offer, or otherwise the bid will be considered as invalid at reveal time. Instances of `Bid` contain a `hash` consisting of 32 bytes. As already said, this will be recomputed at reveal time and matched against the result. Since arrays cannot be stored in blockchain, we use the storage class `io.takamaka.code.util.Bytes32Snapshot` here, a library class that holds 32 bytes, as a traditional array (see Specialized Storage Array Classes). It is well possible to use a `StorageArray` of a wrapper of `byte` here, but `Bytes32Snapshot` is much more compact and its methods consume less gas.

Class `RevealedBid` describes a bid revealed after bidding time. It contains the real value of the bid, the salt used to strengthen the hashing algorithm and a boolean `fake` that, when true, means that the bid must be considered as invalid, since it was only placed in order to confuse other bidders. It is possible to recompute and check the hash of a revealed bid through method `Bid.matches()`, that uses a given hashing algorithm (`digest`, a Java `java.security.MessageDigest`) to hash value, fake mark and salt into bytes, finally compared against the hash provided at bidding time.

The `BlindAuction` contract stores the `beneficiary` of the auction. It is the contract that created the auction and is consequently initialized, in the constructor of `BlindAuction`, to its caller. The constructor must be annotated as `@FromContract` because of that. The same constructor receives the length of bidding time and reveal time, in milliseconds. This allows the contract to compute tha absolute ending time for the bidding phase and for the reveal phase, stored into fields `biddingEnd` and `revealEnd`, respectively. Note, in the contructor of `BlindAuction`, the use of the static method `io.takamaka.code.lang.Takamaka.now()`, that yields the current time, as with the traditional `System.currentTimeMillis()` of Java (that instead cannot be used in Takamaka code). Method `now()`, in a blockchain, yields the time of creation of the block of the current transaction, as seen by its miner. That time is reported in the block and hence is independent from the machine that runs the contract, which guarantees determinism.

Method `bid()` allows a caller (the bidder) to place a bid during the bidding phase. An instance of `Bid` is created and added to a list, specific to each bidder. Here is where our map comes to help. Namely, field `bids` holds a `StorageTreeMap<PayableContract, StorageList<Bid>>`, that can be held in the store of a node since it is a storage map between storage keys and storage values. Method `bid()` computes an empty list of bids if it is the first time that a bidder places a bid. For that, it uses method `computeIfAbsent()` of `StorageMap`. If it used method `get()`, it would run into a null-pointer exception the first time a bidder places a bid. That is, storage maps default to

`null`, as all Java maps. (But differently to Solidity maps, that provide a new value automatically when undefined.)

Method `reveal()` is called by each bidder during the reveal phase. It accesses the `bids` placed by the bidder during the bidding time. The method matches each revealed bid against the corresponding list of bids for the player, by calling method `refundFor()`, that determines how much of the deposit must be refunded to the bidder. Namely, if a bid was fake or was not the best bid, it must be refunded in full. If it was the best bid, it must be partially refunded if the apparent `deposit` turns out to be higher than the actual value of the revealed bid. While bids are refunded, method `placeBid` updates the best bid information.

Method `auctionEnd()` is meant to be called after the reveal phase. If there is a winner, it sends the highest bid to the beneficiary.

Note the use of methods `onlyBefore()` and `onlyAfter()` to guarantee that some methods are only run at the right moment.

## Events

**[See project `auction_events` inside the `hotmoka_tutorial` repository]**

The code in the previous section does not compile since it misses two classes `BidIncrease.java` and `AuctionEnd.java`, that we report below. Namely, the code of the blind auction contract contains some lines that generate *events*, such as:

```
event(new AuctionEnd(winner, highestBid));
```

Events are milestones that are saved in the store of a Hotmoka node. From outside the node, it is possible to subscribe to specific events and get notified as soon as an event of that kind occurs, to trigger actions when that happens. In terms of the Takamaka language, events are generated through the `io.takamaka.code.lang.Takamaka.event(Event event)` method, that receives a parameter of type `io.takamaka.code.lang.Event`. The latter is simply an abstract class that extends `Storage`. Hence, events will be stored in the node as part of the transaction that generated that event. The constructor of class `Event` is annotated as `FromContract`, which allows one to create events from the code of contracts only. The creating contract is available through method `creator()` of class `Event`.

In our example, the `BlindAuction` class uses two events, that you can add to the `io.takamaka.auction` package and are defined as follows:

```
package io.takamaka.auction;

import java.math.BigInteger;

import io.takamaka.code.lang.FromContract;
import io.takamaka.code.lang.Event;
import io.takamaka.code.lang.PayableContract;
import io.takamaka.code.lang.View;

public class BidIncrease extends Event {
  public final PayableContract bidder;
  public final BigInteger amount;

  @FromContract BidIncrease(PayableContract bidder, BigInteger amount) {
```

```
    this.bidder = bidder;
    this.amount = amount;
  }

  public @View PayableContract getBidder() {
    return bidder;
  }

  public @View BigInteger getAmount() {
    return amount;
  }
}
```

and

```
package io.takamaka.auction;

import java.math.BigInteger;

import io.takamaka.code.lang.FromContract;
import io.takamaka.code.lang.Event;
import io.takamaka.code.lang.PayableContract;
import io.takamaka.code.lang.View;

public class AuctionEnd extends Event {
  public final PayableContract highestBidder;
  public final BigInteger highestBid;

  @FromContract AuctionEnd(PayableContract highestBidder, BigInteger highestBid) {
    this.highestBidder = highestBidder;
    this.highestBid = highestBid;
  }

  public @View PayableContract getHighestBidder() {
    return highestBidder;
  }

  public @View BigInteger getHighestBid() {
  return highestBid;
  }
}
```

Now that all classes have been completed, the project should compile. Go inside the `auction` project and run `mvn package`. A file `auction-0.0.1.jar` should appear inside `target`.

## Running the Blind Auction Contract

**[See project `runs` inside the `hotmoka_tutorial` repository]**

Go to the `runs` Eclipse project and add the following class inside that package:

```
package runs;

import static io.hotmoka.beans.Coin.panarea;
```

```
import static io.hotmoka.beans.types.BasicTypes.BOOLEAN;
import static io.hotmoka.beans.types.BasicTypes.BYTE;
import static io.hotmoka.beans.types.BasicTypes.INT;
import static io.hotmoka.beans.types.ClassType.BIG_INTEGER;
import static io.hotmoka.beans.types.ClassType.BYTES32_SNAPSHOT;
import static io.hotmoka.beans.types.ClassType.PAYABLE_CONTRACT;

import java.math.BigInteger;
import java.nio.file.Files;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.security.KeyPair;
import java.security.MessageDigest;
import java.util.ArrayList;
import java.util.List;
import java.util.Random;
import java.util.stream.Stream;

import io.hotmoka.beans.SignatureAlgorithm;
import io.hotmoka.beans.references.TransactionReference;
import io.hotmoka.beans.requests.ConstructorCallTransactionRequest;
import io.hotmoka.beans.requests.InstanceMethodCallTransactionRequest;
import io.hotmoka.beans.requests.JarStoreTransactionRequest;
import io.hotmoka.beans.requests.SignedTransactionRequest;
import io.hotmoka.beans.requests.SignedTransactionRequest.Signer;
import io.hotmoka.beans.signatures.CodeSignature;
import io.hotmoka.beans.signatures.ConstructorSignature;
import io.hotmoka.beans.signatures.MethodSignature;
import io.hotmoka.beans.signatures.NonVoidMethodSignature;
import io.hotmoka.beans.signatures.VoidMethodSignature;
import io.hotmoka.beans.types.ClassType;
import io.hotmoka.beans.values.BigIntegerValue;
import io.hotmoka.beans.values.BooleanValue;
import io.hotmoka.beans.values.ByteValue;
import io.hotmoka.beans.values.IntValue;
import io.hotmoka.beans.values.StorageReference;
import io.hotmoka.beans.values.StorageValue;
import io.hotmoka.beans.values.StringValue;
import io.hotmoka.crypto.Account;
import io.hotmoka.crypto.SignatureAlgorithmForTransactionRequests;
import io.hotmoka.helpers.GasHelper;
import io.hotmoka.helpers.NonceHelper;
import io.hotmoka.helpers.SignatureHelper;
import io.hotmoka.nodes.Node;
import io.hotmoka.remote.RemoteNode;
import io.hotmoka.remote.RemoteNodeConfig;

public class Auction {
  // change this with your accounts' storage references
  private final static String[] ADDRESSES =
   { "75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0",
     "26b240580489d5a00e241db547fe2ae756a0209ae87fc6a17e4a06f36f1e7ff0#0",
     "48ef7306af5e86adbe01dd7807e1c7bf30fbd3781a63e770245ac72743c5fd1a#0" };
```

```
public final static int NUM_BIDS = 10; // number of bids placed
public final static int BIDDING_TIME = 130_000; // in milliseconds
public final static int REVEAL_TIME = 170_000; // in milliseconds

private final static BigInteger _500_000 = BigInteger.valueOf(500_000);
private final static ClassType BLIND_AUCTION
  = new ClassType("io.takamaka.auction.BlindAuction");
private final static ConstructorSignature CONSTRUCTOR_BLIND_AUCTION
  = new ConstructorSignature(BLIND_AUCTION, INT, INT);
private final static ConstructorSignature CONSTRUCTOR_BYTES32_SNAPSHOT
  = new ConstructorSignature(BYTES32_SNAPSHOT,
      BYTE, BYTE, BYTE, BYTE, BYTE, BYTE, BYTE, BYTE,
      BYTE, BYTE, BYTE, BYTE, BYTE, BYTE, BYTE, BYTE,
      BYTE, BYTE, BYTE, BYTE, BYTE, BYTE, BYTE, BYTE,
      BYTE, BYTE, BYTE, BYTE, BYTE, BYTE, BYTE, BYTE);
private final static ConstructorSignature CONSTRUCTOR_REVEALED_BID
  = new ConstructorSignature(
      new ClassType("io.takamaka.auction.BlindAuction$RevealedBid"),
      BIG_INTEGER, BOOLEAN, BYTES32_SNAPSHOT);
private final static MethodSignature BID = new VoidMethodSignature
   (BLIND_AUCTION, "bid", BIG_INTEGER, BYTES32_SNAPSHOT);
private final static MethodSignature REVEAL = new VoidMethodSignature
   (BLIND_AUCTION, "reveal",
   new ClassType("io.takamaka.auction.BlindAuction$RevealedBid"));
private final static MethodSignature AUCTION_END = new NonVoidMethodSignature
   (BLIND_AUCTION, "auctionEnd", PAYABLE_CONTRACT);

// the hashing algorithm used to hide the bids
private final MessageDigest digest = MessageDigest.getInstance("SHA-256");

private final Path auctionPath = Paths.get("../auction/target/auction-0.0.1.jar");
private final TransactionReference takamakaCode;
private final StorageReference[] accounts;
private final Signer[] signers;
private final String chainId;
private final long start;  // the time when bids started being placed
private final Node node;
private final TransactionReference classpath;
private final StorageReference auction;
private final List<BidToReveal> bids = new ArrayList<>();
private final GasHelper gasHelper;
private final NonceHelper nonceHelper;

public static void main(String[] args) throws Exception {
  RemoteNodeConfig config = new RemoteNodeConfig.Builder()
    .setURL("panarea.hotmoka.io")
    .build();

  try (Node node = RemoteNode.of(config)) {
    new Auction(node);
  }
}
```

```
/**
 * Class used to keep in memory the bids placed by each player,
 * that will be revealed at the end.
 */
private class BidToReveal {
  private final int player;
  private final BigInteger value;
  private final boolean fake;
  private final byte[] salt;

  private BidToReveal(int player, BigInteger value, boolean fake, byte[] salt) {
    this.player = player;
    this.value = value;
    this.fake = fake;
    this.salt = salt;
  }

  /**
   * Creates in store a revealed bid corresponding to this object.
   *
   * @return the storage reference to the freshly created revealed bid
   */
  private StorageReference intoBlockchain() throws Exception {
    StorageReference bytes32 = node.addConstructorCallTransaction(new ConstructorCallTransactionReques
     (signers[player], accounts[player],
     nonceHelper.getNonceOf(accounts[player]), chainId, _500_000,
     panarea(gasHelper.getSafeGasPrice()), classpath, CONSTRUCTOR_BYTES32_SNAPSHOT,
     new ByteValue(salt[0]), new ByteValue(salt[1]),
     new ByteValue(salt[2]), new ByteValue(salt[3]),
     new ByteValue(salt[4]), new ByteValue(salt[5]),
     new ByteValue(salt[6]), new ByteValue(salt[7]),
     new ByteValue(salt[8]), new ByteValue(salt[9]),
     new ByteValue(salt[10]), new ByteValue(salt[11]),
     new ByteValue(salt[12]), new ByteValue(salt[13]),
     new ByteValue(salt[14]), new ByteValue(salt[15]),
     new ByteValue(salt[16]), new ByteValue(salt[17]),
     new ByteValue(salt[18]), new ByteValue(salt[19]),
     new ByteValue(salt[20]), new ByteValue(salt[21]),
     new ByteValue(salt[22]), new ByteValue(salt[23]),
     new ByteValue(salt[24]), new ByteValue(salt[25]),
     new ByteValue(salt[26]), new ByteValue(salt[27]),
     new ByteValue(salt[28]), new ByteValue(salt[29]),
     new ByteValue(salt[30]), new ByteValue(salt[31])));

    return node.addConstructorCallTransaction(new ConstructorCallTransactionRequest
      (signers[player], accounts[player],
      nonceHelper.getNonceOf(accounts[player]), chainId,
      _500_000, panarea(gasHelper.getSafeGasPrice()), classpath, CONSTRUCTOR_REVEALED_BID,
      new BigIntegerValue(value), new BooleanValue(fake), bytes32));
  }
}
```

```
private Auction(Node node) throws Exception {
  this.node = node;
  takamakaCode = node.getTakamakaCode();
  accounts = Stream.of(ADDRESSES).map(StorageReference::new)
    .toArray(StorageReference[]::new);
  SignatureAlgorithm<SignedTransactionRequest> signature
    = SignatureAlgorithmForTransactionRequests.mk
      (node.getNameOfSignatureAlgorithmForRequests());
  signers = Stream.of(accounts).map(this::loadKeys)
    .map(keys -> Signer.with(signature, keys)).toArray(Signer[]::new);
  gasHelper = new GasHelper(node);
  nonceHelper = new NonceHelper(node);
  chainId = getChainId();
  classpath = installJar();
  auction = createContract();
  start = System.currentTimeMillis();

  StorageReference expectedWinner = placeBids();
  waitUntilEndOfBiddingTime();
  revealBids();
  waitUntilEndOfRevealTime();
  StorageValue winner = askForWinner();

  // show that the contract computes the correct winner
  System.out.println("expected winner: " + expectedWinner);
  System.out.println("actual winner: " + winner);
}

private StorageReference createContract() throws Exception {
  System.out.println("Creating contract");

  return node.addConstructorCallTransaction
    (new ConstructorCallTransactionRequest(signers[0], accounts[0],
    nonceHelper.getNonceOf(accounts[0]), chainId, _500_000,
    panarea(gasHelper.getSafeGasPrice()),
    classpath, CONSTRUCTOR_BLIND_AUCTION,
    new IntValue(BIDDING_TIME), new IntValue(REVEAL_TIME)));
}

private String getChainId() throws Exception {
  return ((StringValue) node.runInstanceMethodCallTransaction
    (new InstanceMethodCallTransactionRequest
    (accounts[0], // payer
    BigInteger.valueOf(50_000), // gas limit
    takamakaCode, // class path for the execution of the transaction
    CodeSignature.GET_CHAIN_ID, // method
    node.getManifest()))) // receiver of the method call
    .value;
}

private TransactionReference installJar() throws Exception {
  System.out.println("Installing jar");
```

```
    return node.addJarStoreTransaction(new JarStoreTransactionRequest
      (signers[0], // an object that signs with the payer's private key
      accounts[0], // payer
      nonceHelper.getNonceOf(accounts[0]), // payer's nonce
      chainId, // chain identifier
      BigInteger.valueOf(1_000_000), // gas limit: enough for this very small jar
      gasHelper.getSafeGasPrice(), // gas price: at least the current gas price of the network
      takamakaCode, // class path for the execution of the transaction
      Files.readAllBytes(auctionPath), // bytes of the jar to install
      takamakaCode)); // dependency
  }

  private StorageReference placeBids() throws Exception {
    BigInteger maxBid = BigInteger.ZERO;
    StorageReference expectedWinner = null;
    Random random = new Random();

    int i = 1;
    while (i <= NUM_BIDS) { // generate NUM_BIDS random bids
      System.out.println("Placing bid " + i);
      int player = 1 + random.nextInt(accounts.length - 1);
      BigInteger deposit = BigInteger.valueOf(random.nextInt(1000));
      BigInteger value = BigInteger.valueOf(random.nextInt(1000));
      boolean fake = random.nextBoolean();
      byte[] salt = new byte[32];
      random.nextBytes(salt); // random 32 bytes of salt for each bid

      // create a Bytes32 hash of the bid in the store of the node
      StorageReference bytes32 = codeAsBytes32(player, value, fake, salt);

      // keep note of the best bid, to verify the result at the end
      if (!fake && deposit.compareTo(value) >= 0)
        if (expectedWinner == null || value.compareTo(maxBid) > 0) {
          maxBid = value;
          expectedWinner = accounts[player];
        }
        else if (value.equals(maxBid))
          // we do not allow ex aequos, since the winner
          // would depend on the fastest player to reveal
          continue;

      // keep the explicit bid in memory, not yet in the node,
      // since it would be visible there
      bids.add(new BidToReveal(player, value, fake, salt));

      // place a hashed bid in the node
      node.addInstanceMethodCallTransaction
        (new InstanceMethodCallTransactionRequest
        (signers[player], accounts[player],
        nonceHelper.getNonceOf(accounts[player]), chainId,
        _500_000, panarea(gasHelper.getSafeGasPrice()), classpath, BID,
        auction, new BigIntegerValue(deposit), bytes32));
```

```
      i++;
    }

    return expectedWinner;
  }

  private void revealBids() throws Exception {
    // we create the revealed bids in blockchain; this is safe now, since the bidding time is over
    int counter = 1;
    for (BidToReveal bid: bids) {
      System.out.println("Revealing bid " + counter++ + " out of " + bids.size());
      int player = bid.player;
      StorageReference bidInBlockchain = bid.intoBlockchain();
      node.addInstanceMethodCallTransaction(new InstanceMethodCallTransactionRequest
        (signers[player], accounts[player],
        nonceHelper.getNonceOf(accounts[player]), chainId, _500_000,
        panarea(gasHelper.getSafeGasPrice()),
        classpath, REVEAL, auction, bidInBlockchain));
    }
  }

  private StorageReference askForWinner() throws Exception {
    StorageValue winner = node.addInstanceMethodCallTransaction
      (new InstanceMethodCallTransactionRequest
      (signers[0], accounts[0], nonceHelper.getNonceOf(accounts[0]),
      chainId, _500_000, panarea(gasHelper.getSafeGasPrice()),
      classpath, AUCTION_END, auction));

    // the winner is normally a StorageReference,
    // but it could be a NullValue if all bids were fake
    return winner instanceof StorageReference ? (StorageReference) winner : null;
  }

  private void waitUntilEndOfBiddingTime() {
    waitUntil(BIDDING_TIME + 5000);
  }

  private void waitUntilEndOfRevealTime() {
    waitUntil(BIDDING_TIME + REVEAL_TIME + 5000);
  }

  /**
   * Waits until a specific time after start.
   */
  private void waitUntil(long duration) {
    try {
      Thread.sleep(start + duration - System.currentTimeMillis());
    }
    catch (InterruptedException e) {}
  }

  /**
   * Hashes a bid and put it in the store of the node, in hashed form.
```

```
 */
private StorageReference codeAsBytes32
  (int player, BigInteger value, boolean fake, byte[] salt) throws Exception {

  digest.reset();
  digest.update(value.toByteArray());
  digest.update(fake ? (byte) 0 : (byte) 1);
  digest.update(salt);
  byte[] hash = digest.digest();
  return createBytes32(player, hash);
}

/**
 * Creates a Bytes32Snapshot object in the store of the node.
 */
private StorageReference createBytes32(int player, byte[] hash) throws Exception {
  return node.addConstructorCallTransaction
    (new ConstructorCallTransactionRequest(
    signers[player],
    accounts[player],
    nonceHelper.getNonceOf(accounts[player]), chainId,
    _500_000, panarea(gasHelper.getSafeGasPrice()),
    classpath, CONSTRUCTOR_BYTES32_SNAPSHOT,
    new ByteValue(hash[0]), new ByteValue(hash[1]),
    new ByteValue(hash[2]), new ByteValue(hash[3]),
    new ByteValue(hash[4]), new ByteValue(hash[5]),
    new ByteValue(hash[6]), new ByteValue(hash[7]),
    new ByteValue(hash[8]), new ByteValue(hash[9]),
    new ByteValue(hash[10]), new ByteValue(hash[11]),
    new ByteValue(hash[12]), new ByteValue(hash[13]),
    new ByteValue(hash[14]), new ByteValue(hash[15]),
    new ByteValue(hash[16]), new ByteValue(hash[17]),
    new ByteValue(hash[18]), new ByteValue(hash[19]),
    new ByteValue(hash[20]), new ByteValue(hash[21]),
    new ByteValue(hash[22]), new ByteValue(hash[23]),
    new ByteValue(hash[24]), new ByteValue(hash[25]),
    new ByteValue(hash[26]), new ByteValue(hash[27]),
    new ByteValue(hash[28]), new ByteValue(hash[29]),
    new ByteValue(hash[30]), new ByteValue(hash[31])));
}

private KeyPair loadKeys(StorageReference account) {
  try {
    String password;
    if (account.toString().equals(ADDRESSES[0]))
      password = "chocolate";
    else if (account.toString().equals(ADDRESSES[1]))
      password = "orange";
    else
      password = "apple";

    return new Account(account, "..")
      .keys(password, new SignatureHelper(node).signatureAlgorithmFor(account));
```

```
    }
    catch (Exception e) {
      throw new RuntimeException(e);
    }
  }
}
```

This test class is relatively long and complex. Let us start from its beginning. The code specifies that the test will place 10 random bids, that the bidding phase lasts 130 seconds and that the reveal phase lasts 170 seconds (these timings are fine on a blockchain that creates a block every five seconds; shorter block creation times allow shorter timings):

```
public final static int NUM_BIDS = 10;
public final static int BIDDING_TIME = 130_000;
public final static int REVEAL_TIME = 170_000;
```

Some constant signatures follow, that simplify the calls to methods and constructors later. Method `main()` connects to a remote node and passes it as a parameter to the constructor of class `Auction`, that installs `auction-0.0.1.jar` inside it. It stores the node in field `node`. Then the constructor of `Auction` creates an `auction` contract in the node and calls method `placeBids()` that uses the inner class `BidToReveal` to keep track of the bids placed during the test, in clear. Initially, bids are kept in memory, not in the store of the node, where they could be publicly accessed. Only their hashes are stored in the node. Method `placeBids()` generates `NUM_BIDS` random bids on behalf of the `accounts.length -1` players (the first element of the `accounts` array is the creator of the auction):

```
int i = 1;
while (i <= NUM_BIDS) {
  int player = 1 + random.nextInt(accounts.length - 1);
  BigInteger deposit = BigInteger.valueOf(random.nextInt(1000));
  BigInteger value = BigInteger.valueOf(random.nextInt(1000));
  boolean fake = random.nextBoolean();
  byte[] salt = new byte[32];
  random.nextBytes(salt);
  ...
}
```

Each random bid is hashed (including a random salt) and a `Bytes32Snapshot` object is created in the store of the node, containing that hash:

```
StorageReference bytes32 = codeAsBytes32(player, value, fake, salt);
```

The bid, in clear, is added to a list `bids` that, at the end of the loop, will contain all bids:

```
bids.add(new BidToReveal(player, value, fake, salt));
```

The hash is used instead to place a bid in the node:

```
node.addInstanceMethodCallTransaction(new InstanceMethodCallTransactionRequest
  (signers[player], accounts[player],
  nonceHelper.getNonceOf(accounts[player]), chainId,
  _500_000, panarea(gasHelper.getSafeGasPrice()), classpath, BID,
  auction, new BigIntegerValue(deposit), bytes32));
```

The loop takes also care of keeping track of the best bidder, that placed the best bid, so that it

can be compared at the end with the best bidder computed by the smart contract (they should coincide):

```
if (!fake && deposit.compareTo(value) >= 0)
  if (expectedWinner == null || value.compareTo(maxBid) > 0) {
    maxBid = value;
    expectedWinner = accounts[player];
  }
  else if (value.equals(maxBid))
    continue;
```

As you can see, the test above avoids generating a bid that is equal to the best bid seen so far. This avoids having two bidders that place the same bid: the smart contract will consider as winner the first bidder that reveals its bids. To avoid this tricky case, we prefer to assume that the best bid is unique. This is just a simplification of the testing code, since the smart contract deals perfectly with that case.

After all bids have been placed, the constructor of `Auction` waits until the end of the bidding time:

```
waitUntilEndOfBiddingTime();
```

Then the constructor of `Auction` calls method `revealBids()`, that reveals the bids to the smart contract, in plain. It creates in the store of the node a data structure `RevealedBid` for each elements of the list `bids`, by calling `bid.intoBlockchain()`. This creates the bid in clear in the store of the node, but this is safe now, since the bidding time is over and they cannot be used to guess a winning bid anymore. Then method `revealBids()` reveals the bids by calling method `reveal()` of the smart contract:

```
for (BidToReveal bid: bids) {
  int player = bid.player;
  StorageReference bidInBlockchain = bid.intoBlockchain();
  node.addInstanceMethodCallTransaction(new InstanceMethodCallTransactionRequest
    (signers[player], accounts[player],
    nonceHelper.getNonceOf(accounts[player]), chainId, _500_000,
    panarea(gasHelper.getSafeGasPrice()),
    classpath, REVEAL, auction, bidInBlockchain));
}
```

Note that this is possible since the inner class `RevealedBid` of the smart contract has been annotated as `@Exported` (see its code in section A Blind Auction Contract), hence its instances can be passed as argument to calls from outside the blockchain.

Subsequently, the constructor of `Auction` waits until the end of the reveal phase:

```
waitUntilEndOfRevealTime();
```

After that, method `askForWinner()` signals to the smart contract that the auction is over and asks about the winner:

```
StorageValue winner = node.addInstanceMethodCallTransaction
  (new InstanceMethodCallTransactionRequest
  (signers[0], accounts[0], nonceHelper.getNonceOf(accounts[0]),
  chainId, _500_000, panarea(gasHelper.getSafeGasPrice()),
  classpath, AUCTION_END, auction));
```

The final two `System.out.println()`'s in the constructor of `Auction` allow one to verify that the smart contract actually computes the right winner, since they will always print the identical storage object (different at each run, in general), such as:

```
expected winner: 48ef7306af5e86adbe01dd7807e1c7bf30fbd3781a63e770245ac72743c5fd1a#0
actual winner: 48ef7306af5e86adbe01dd7807e1c7bf30fbd3781a63e770245ac72743c5fd1a#0
```

You can run class `Auction` from Eclipse. Please remember that the execution of this test will take a few minutes. Moreover, remember to put your accounts at the beginning of `Auction.java` and ensure that they have enough balance for this long execution. Its execution should print something like this on the console:

```
Installing jar
Creating contract
Placing bid 1
Placing bid 2
Placing bid 3
...
Placing bid 10
Revealing bid 1 out of 20
Revealing bid 2 out of 20
Revealing bid 3 out of 20
...
Revealing bid 10 out of 10
expected winner: 26b240580489d5a00e241db547fe2ae756a0209ae87fc6a17e4a06f36f1e7ff0#0
actual winner: 26b240580489d5a00e241db547fe2ae756a0209ae87fc6a17e4a06f36f1e7ff0#0
```

## Listening to Events

**[See project runs inside the `hotmoka_tutorial` repository]**

The `BlindAuction` contract generates events during its execution. If an external tool, such as a wallet, wants to listen to such events and trigger some activity when they occur, it is enough for it to subscribe to the events of a node that is executing the contract, by providing a handler that gets executed each time a new event gets generated. Subscription requires to specify the creator of the events that should be forwarded to the handler. In our case, this is the `auction` contract. Thus, clone the `Auction.java` class into `Events.java` and modify its constructor as follows:

```
...
import io.hotmoka.nodes.Node.Subscription;
...
auction = createAuction();
start = System.currentTimeMillis();

try (Subscription subscription = node.subscribeToEvents(auction,
  (creator, event) -> System.out.println
    ("Seen event of class " + node.getClassTag(event).clazz.name
     + " created by contract " + creator))) {

  StorageReference expectedWinner = placeBids();
  waitUntilEndOfBiddingTime();
  revealBids();
```

```
  waitUntilEndOfRevealTime();
  StorageValue winner = askForWinner();

  System.out.println("expected winner: " + expectedWinner);
  System.out.println("actual winner: " + winner);
}
...
```

The event handler, in this case, simply prints on the console the class of the event and its creator (that will coincide with `auction`).

If you run the `Events` class, you should see something like this on the console:

```
Seen event of class io.takamaka.auction.BidIncrease
  created by contract 310d241d1f5dbe955f25ede96be324ade...#0
Seen event of class io.takamaka.auction.BidIncrease
  created by contract 310d241d1f5dbe955f25ede96be324ade...#0
Seen event of class io.takamaka.auction.BidIncrease
  created by contract 310d241d1f5dbe955f25ede96be324ade...#0
Seen event of class io.takamaka.auction.AuctionEnd
  created by contract 310d241d1f5dbe955f25ede96be324ade...#0
```

The `subscribeToEvents()` method returns a `Subscription` object that should be closed when it is not needed anymore, in order to reduce the overhead on the node. Since it is an `AutoCloseable` resource, the recommended technique is to use a try-with-resource construct, as shown in the previous example.

In general, event handlers can perform arbitrarily complex operations and even access the event object in the store of the node, from its storage reference, reading its fields or calling its methods. Please remember, however, that event handlers are run in a thread of the node. Hence, they should be fast and shouldn't hang. It is good practice to let event handlers add events in a queue, in a non-blocking way. A consumer thread, external to the node, then retrieves the events from the queue and process them in turn.

It is possible to subscribe to *all* events generated by a node, by using `null` as creator in the `subscribeToEvents()` method. Think twice before doing that, since your handler will be notified of *all* events generated by *any* application installed in the node. It might be a lot.

Chapter 6

# Hotmoka Nodes

A Hotmoka node is a device that implements an interface for running Java code remotely. It can be any kind of device, such as a device of an IoT network, but also a node of a blockchain. We have already used instances of Hotmoka nodes, namely, instances of `RemoteNode`. But there are other examples of nodes, that we will describe in this chapter.

The interface `io.hotmoka.nodes.Node` is shown in the topmost part of Figure 32. That interface can be split in five parts:

1. A `get` part, that includes methods for querying the state of the node and for accessing the objects contained in its store.
2. An `add` part, that expands the store of the node with the result of a transaction.
3. A `run` part, that allows one to run transactions that execute `@View` methods and hence do not expand the store of the node.
4. A `post` part, that expands the store of the node with the result of a transaction, without waiting for its result; instead, a future is returned.
5. A `subscribe` part, that allows users to subscribe listeners of events generated during the execution of the transactions.

If a node belongs to a blockchain, then all nodes of the blockchain have the same vision of the state, so that it is equivalent to call a method of a node or of any other node of the network. The only method that is out of consensus, since it contains information specific to each node, is `getNodeInfo`. It reports for instance the type of the node and its version.

Looking at Figure 32, it is possible to see that the `Node` interface has many implementations, that we describe below.

### Local Implementations

*Local Implementations* are actual nodes that run on the machine where they have been started. For instance, they can be a node of a larger blockchain network. Among them, `TendermintBlockchain` implements a node of a Tendermint blockchain and will be presented in Tendermint Nodes. `MemoryBlockchain` implements a single-node blockchain in RAM, hence its content is partially lost after it is turned off. It is useful for debugging, testing and learning, since it allows one to inspect the content of blocks, transactions and store. It will be presented in Memory Nodes.

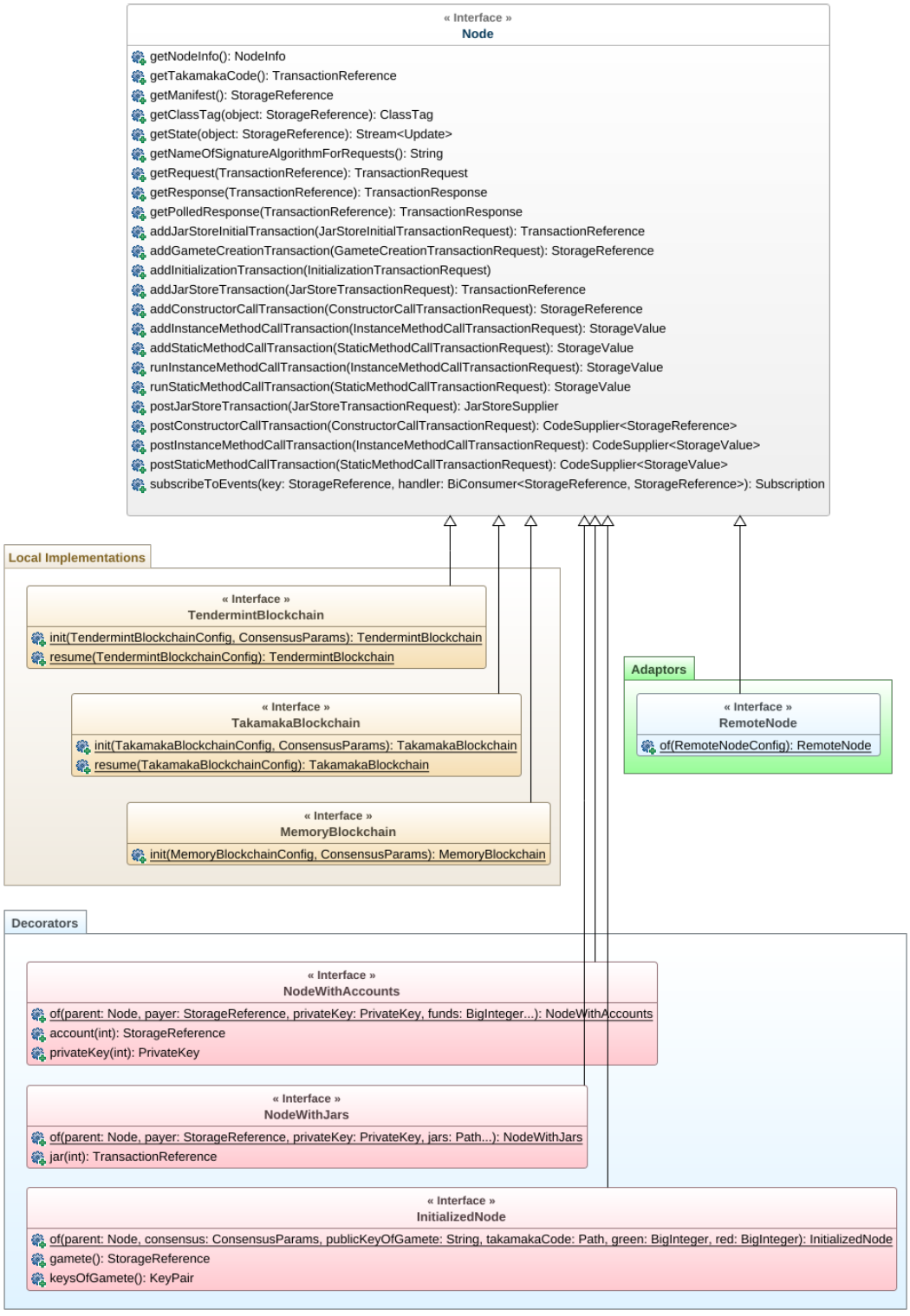



*Figure 32. The hierarchy of Hotmoka nodes.*

`TakamakaBlockchain` implements a node for the Takamaka blockchain developed by Ailia SA. Local nodes can be instantiated through the static factory method `init()` of their implementing interface. That method requires to specify parameters that are global to the network of nodes (`ConsensusParams`) and must be the same for all nodes in the network, and parameters that are specific to the given node of the network that is being started and can be different from node to node (`TendermintBlockchainConfig` and similar). Some implementations have to ability to *resume*. This means that they recover the state at the end of a previous execution, reconstruct the consensus parameters from that state and resume the execution from there, downloading and verifying blocks already processed by the network. In order to resume a node from an already existing state, the static `resume` method of its implementing interface must be used.

### Decorators

The `Node` interface is implemented by some decorators as well. Typically, these decorators run some transactions on the decorated node, to simplify some tasks, such as the initialization of a node, the installation of jars into a node or the creation of accounts in a node. These decorators are views of the decorated node, in the sense that any method of the `Node` interface, invoked on the decorator, is forwarded to the decorated node. We will discuss them in Node Decorators.

### Adaptors

Very often, one wants to *publish* a node online, so that we (and other programmers who need its service) can use it concurrently. This should be possible for all implementations of the `Node` interface, such as `MemoryBlockchain`, `TendermintBlockchain` and all present and future implementations. In other words, we would like to publish *any* Hotmoka node as a service, accessible through the internet. This will be the subject of Publishing a Hotmoka Node Online. Conversely, once a Hotmoka node has been published at some URL, say `http://my.company.com`, it will be accessible through some network API, through the SOAP or REST protocol, and also through websockets, for subscription to events. This complexity might make it awkward, for a programmer, to use the published node. In that case, we can create an instance of `Node` that operates as a proxy to the network service, helping programmers integrate their software to the service in a seamless way. This *remote* node still implements the `Node` interface, but simply forwards all its calls to the remote service. By programming against the same `Node` interface, it becomes easy for a programmer to swap a local node with a remote node, or vice versa. This mechanism is described in Remote Nodes, where the adaptor interface `RemoteNode` in Figure 32 is presented.

## Tendermint Nodes

Tendermint [Tendermint] is a Byzantine-fault tolerant engine for building blockchains, that replicates a finite-state machine on a network of nodes across the world. The finite-state machine is often referred to as a *Tendermint app*. The nice feature of Tendermint is that it takes care of all issues related to networking and consensus, leaving to the developer only the task to develop the Tendermint app.

There is a Hotmoka node that implements such a Tendermint app, for programming in Takamaka over Tendermint. In order to use it, the Tendermint executable must be installed in our machine, or our experiments will fail. The Hotmoka node works with Tendermint version 0.34.15, that can be downloaded in executable form from https://github.com/tendermint/tendermint/releases/tag/v0.34.15. Be sure that you download that executable and install it at a place that is part of the

command-line path of your computer. This means that, if you run the following command in a shell:

```
$ tendermint version
```

the answer must be

```
0.34.15
```

or similar, as long as the version starts with 0.34.15. Our Hotmoka node built on Tendermint is known to work on Windows, MacOs and Linux machines.

Before starting a local node of a Hotmoka blockchain based on Tendermint, you need to create the Tendermint configuration file. For instance, in order to run a single validator node with no non-validator nodes, you can create its configuration as follows:

```
$ tendermint testnet --v 1 --n 0
```

This will create a directory `mytestnet/node0` for a single Tendermint node, that includes the configuration of the node and its private and public validator keys.

Once this is done, you can create a key pair for the gamete of the node that you are going to start. You perform this with `moka`:

```
$ moka create-key

Please specify the password of the new key: king
A new key FaHYC1TxCJBcpgz8FrXy2bidwNBgPjPg1L7GEHaDHwmZ has been created.
Its entropy has been saved into the file
  "./FaHYC1TxCJBcpgz8FrXy2bidwNBgPjPg1L7GEHaDHwmZ.pem".
```

The entropy is a representation of the key pair. The name of the file is the Base58-encoded public key of the pair. While the entropy and the password are secret information that you should not distribute, the public key can be used to create a new node. Namely, you can start a Hotmoka node based on Tendermint, making reference to the Tendermint configuration directory that you have just created, by using the `moka init-tendermint` command. You also need to specify how much coins are minted for the gamete and where is the jar of the runtime of Takamaka, that will be stored inside the node as `takamakaCode`: we use the local Maven's cache for that:

```
$ moka init-tendermint 100000000000000
    --tendermint-config mytestnet/node0
    --takamaka-code ~/.m2/repository/io/hotmoka/io-takamaka-code/
                        1.0.7/io-takamaka-code-1.0.7.jar
    --key-of-gamete FaHYC1TxCJBcpgz8FrXy2bidwNBgPjPg1L7GEHaDHwmZ

Do you really want to start a new node at this place
  (old blocks and store will be lost) [Y/N] Y

The following node has been initialized:
  takamakaCode: b23118aa95fa436f951bdc78d5ffea99a7bd72cf1512bef3df2ea12993f18a70
  manifest: 21d375ae9bac3b74d1a54a6418f7c70c2c107665fb2066a94dbf65cb3db9cdc6#0
    chainId: chain-btmZzq
    ...
    signature: ed25519
    gamete: d2fc1b34d6e4b2d2d80f7665d5ef4d5eb81e927cebe2240aec4dda7c1173542b#0
      balance: 100000000000000
      maxFaucet: 0
      ...
    ...

The Hotmoka node has been published at localhost:8080
Try for instance in a browser: http://localhost:8080/get/manifest

The owner of the key of the gamete can bind it to its address now:
  moka bind-key FaHYC1TxCJBcpgz8FrXy2bidwNBgPjPg1L7GEHaDHwmZ
    --url url_of_this_node
or
  moka bind-key FaHYC1TxCJBcpgz8FrXy2bidwNBgPjPg1L7GEHaDHwmZ
    --reference d2fc1b34d6e4b2d2d80f7665d5ef4d5eb81e927cebe2240aec4dda7c1173542b#0

Press enter to exit this program and turn off the node
```

This command has done a lot! It has created an instance of `TendermintBlockchain`; it has stored the `io-takamaka-code` jar inside it, at a reference called `takamakaCode`; it has created a Java object, called manifest, that contains other objects, including an externally-owned account named `gamete`, whose public key is that provided after `--key-of-gamete`; it has initialized the balance of the gamete to the value passed after `moka init-tendermint`. Finally, this command has published an internet service at localhost, that exports the API of the node. For instance, you can open the suggested URL

```
http://localhost:8080/get/manifest
```

in a browser, to see the JSON answer:


```
{"transaction":{"type":"local",
    "hash":"21d375ae9bac3b74d1a54a6418f7c70c2c107665fb2066a94dbf65cb3db9cdc6"},
  "progressive":"0"}
```


You can also try

```
http://localhost:8080/get/nameOfSignatureAlgorithmForRequests
```

and see the following response in your browser:

```
{"algorithm":"ed25519"}
```

In order to use the gamete, you should bind its key to its actual storage reference in the node, on your local machine. Open another shell, move inside the directory holding the keys of the gamete and digit:

```
$ moka bind-key FaHYC1TxCJBcpgz8FrXy2bidwNBgPjPg1L7GEHaDHwmZ

A new account d2fc1b34d6e4b2d2d80f7665d5ef4d5eb81e927cebe2240aec4dda7c1173542b#0
  has been created.
Its entropy has been saved into the file
  "./d2fc1b34d6e4b2d2d80f7665d5ef4d5eb81e927cebe2240aec4dda7c1173542b#0.pem".
```

This operation has created a pem file whose name is that of the storage reference of the gamete. With this file, it is possiblke to run transactions on behalf of the gamete.

Your computer exports a Hotmoka node now, running on Tendermint. If your computer is reachable at some address `my.machine`, anybody can contact your node at `http://my.machine:8080`, query your node and run transactions on your node. However, what has been created is a Tendermint node where all initial coins are inside the gamete. By using the gamete, *you* can fill the node with objects and accounts now, and in general run all transactions you want. However, other users, who do not know the keys of the gamete, will not be able to run any non-`@View` transaction on your node. If you want to open a faucet, so that other users can gain droplets of coins, you must add the `--open-unsigned-faucet` option to the `moka init-tendermint` command above. Then, *in another shell* (since the previous one is busy with the execution of the node), in a directory holding the keys of the gamete, you can type:

```
$ moka faucet 5000000

Please specify the password of the gamete account: king
```

which specifies the maximal amount of coins that the faucet is willing to give away at each request (its *flow*). You can re-run the `moka faucet` command many times, in order to change the flow of the faucet, or close it completely. Needless to say, only the owner of the keys of the gamete can run the `moka faucet` command, which is why the file with the entropy of the gamete must be in the directory where you run `moka faucet`.

After opening a faucet with a sufficient flow, anybody can re-run the examples of the previous chapters by replacing `panarea.hotmoka.io` with `my.machine:8080`: your computer will serve the requests and run the transactions.

If you turn off your Hotmoka node based on Tendermint, its state remains saved inside the `chain` directory: the `chain/blocks` subdirectory is where Tendermint stores the blocks of the chain; while `chain/store` is where the Tendermint app for smart contracts in Takamaka stores its state, consisting of the storage objects created in blockchain. Later, you can resume the node from that state, by typing:

```
$ moka resume-tendermint --tendermint-config mytestnet/node0
...
Press enter to exit this program and turn off the node
```

There is a log file that can be useful to check the state of our Hotmoka-Tendermint app. Namely, `tendermint.log` contains the log of Tendermint itself. It can be interesting to inspect which blocks are committed and when:

```
I[2021-05-05|11:46:00.113] Version info, software=0.34.15 block=10 p2p=7
I[2021-05-05|11:46:00.248] Starting Node, impl=Node
I[2021-05-05|11:46:00.364] Started node, nodeInfo=
  "{ProtocolVersion:{P2P:7 Block:10 App:0}
   ID_:6615dcd76f7ecd1bde824c45f316c719b6bfe55c  ListenAddr:tcp://0.0.0.0:26656
   Network:chain-btmZzq  Version:0.34.15  Channels:4020212223303800
   Moniker:filicudi  Other:{TxIndex:on RPCAddress:tcp://127.0.0.1:26657}}"
I[2021-05-05|11:46:04.597] Executed block, height=1 validTxs=1 invalidTxs=0
I[2021-05-05|11:46:04.657] Committed state, height=1 txs=1 appHash=E83360...
I[2021-05-05|11:46:05.377] Executed block, height=2 validTxs=1 invalidTxs=0
I[2021-05-05|11:46:05.441] Committed state, height=2 txs=1 appHash=C923A1...
...
I[2021-05-05|11:46:15.501] Executed block, height=9 validTxs=3 invalidTxs=0
I[2021-05-05|11:46:15.568] Committed state, height=9 txs=3 appHash=4876BD...
...
```

Note how the block height increases and that the application hash changes whenever a block contains transactions (`validTxs`>0), reflecting the fact that the state has been modified.

## Memory Nodes

The Tendermint nodes of the previous section form a real blockchain. They are perfect for deploying a blockchain where we can program smart contracts in Takamaka. Nevertheless, they are slow for debugging: transactions are committed every second, by default. Hence, if we want to see the result of a transaction, we have to wait for one second at least. Moreover, Tendermint does not allow one to see the effects of each single transaction, in a simple way. For testing, debugging and didactical purposes, it would be simpler to have a light node that behaves like a blockchain, allows access to blocks and transactions as text files, but is not a blockchain. This is the goal of the `MemoryBlockchain` nodes. They are not part of an actual blockchain since they do not duplicate transactions in a peer-to-peer network, where consensus is imposed. But they are very handy because they allow one to inspect, very easily, the requests sent to the node and the corresponding responses.

You can start a memory Hotmoka node, with an open faucet, exactly as you did, in the previous section, for a Tendermint node, but using the `moka init-memory` command instead of `moka init-tendermint`. You do not need any Tendermint configuration this time:

```
$ moka init-memory 10000000000000000000000
    --open-unsigned-faucet
    --takamaka-code ~/.m2/repository/io/hotmoka/io-takamaka-code/
                        1.0.7/io-takamaka-code-1.0.7.jar
    --key-of-gamete FaHYC1TxCJBcpgz8FrXy2bidwNBgPjPg1L7GEHaDHwmZ

Do you really want to start a new node at this place
  (old blocks and store will be lost) [Y/N] Y

The following node has been initialized:
  takamakaCode: b23118aa95fa436f951bdc78d5ffea99a7bd72cf1512bef3df2ea12993f18a70
  manifest: ff7855ed728c2f323341d493a6a7b33218e4844b512c3dd86220e05fd0af7847#0
    chainId:
    ...
    signature: ed25519
    gamete: ee7a549a9419f6178efea6291121535efd71aa6c98233c89a4a0fae700a6efcc#0
      balance: 10000000000000000000000
      maxFaucet: 0
      ...
    ...

The Hotmoka node has been published at localhost:8080
Try for instance in a browser: http://localhost:8080/get/manifest

The owner of the key of the gamete can bind it to its address now:
  moka bind-key FaHYC1TxCJBcpgz8FrXy2bidwNBgPjPg1L7GEHaDHwmZ --url url_of_this_node
or
  moka bind-key FaHYC1TxCJBcpgz8FrXy2bidwNBgPjPg1L7GEHaDHwmZ
    --reference ee7a549a9419f6178efea6291121535efd71aa6c98233c89a4a0fae700a6efcc#0

Press enter to exit this program and turn off the node
```

Then, in another shell, move in the directory holding the keys of the gamete, bind the gamete to the keys and open the faucet:

```
$ moka bind-key FaHYC1TxCJBcpgz8FrXy2bidwNBgPjPg1L7GEHaDHwmZ

A new account ee7a549a9419f6178efea6291121535efd71aa6c98233c89a4a0fae700a6efcc#0
  has been created.
Its entropy has been saved into the file
  "./ee7a549a9419f6178efea6291121535efd71aa6c98233c89a4a0fae700a6efcc#0.pem".

$ moka faucet 5000000000000000

Please specify the password of the gamete account: queen
```

You won't notice any real difference with Tendermint, but for the fact that this node is faster, its chain identifier is the empty string and it has no validators. Blocks and transactions are inside the `chain` directory, that this time contains a nice textual representation of requests and responses:

```
$ tree chain
chain
  b0
    0-b23118aa95fa436f951bdc78d5ffea99a7bd72cf1512bef3df2ea12993f18a70
      request.txt
      response.txt
    1-ee7a549a9419f6178efea6291121535efd71aa6c98233c89a4a0fae700a6efcc
      request.txt
      response.txt
    2-3c0ba38654c78476d488a4bfedcc1debf2c33c2c79979ee044ca583d68c2d4d0
      request.txt
      response.txt
    3-951f3cb034c5b8ac9b7d40c4693ee73c46ae3f9b50120a3548f6b782474dc972
      request.txt
      response.txt
    4-ff7855ed728c2f323341d493a6a7b33218e4844b512c3dd86220e05fd0af7847
      request.txt
      response.txt
  b1
    0-bc7b704430a5f683ee3c8d1df303d74172a3590526ec9ed738ebd3fa017b46ee
      request.txt
      response.txt
    1-02cda6840f83b19a9e02884d80fda9f721575c35800c7ab9e005b9a80a5c9696
      request.txt
      response.txt
```

The exact ids and the number of these transactions will be different in your computer.

There are two blocks, `b0` and `b1`, each containing up to five transactions. Each transaction is reported with its id and the pair request/response that the node has computed for it. They are text files, that you can open to understand what is happening inside the node.

The transactions shown above are those that have initialized the node and opened the faucet. The last transaction inside each block is a *reward* transaction, that distributes the earnings of the block to the (zero) validators and increases block height and number of transactions in the manifest.

Spend some time looking at the `request.txt` and `response.txt` files. In particular, the last transaction inside `b1` should be that triggered by your `moka faucet` command. Open its `request.txt` file. It should read like this:

```
InstanceMethodCallTransactionRequest:
  caller: ee7a549a9419f6178efea6291121535efd71aa6c98233c89a4a0fae700a6efcc#0
  nonce: 3
  gas limit: 100000
  gas price: 100
  class path: b23118aa95fa436f951bdc78d5ffea99a7bd72cf1512bef3df2ea12993f18a70
  method: void io.takamaka.code.lang.Gamete.setMaxFaucet(BigInteger, BigInteger)
  actuals:
    5000000000000000
    0
  receiver: ee7a549a9419f6178efea6291121535efd71aa6c98233c89a4a0fae700a6efcc#0
  chainId:
  signature: 6934f9b1b614ff1fb5cc0e84929b60a0fa4ca5f292c8946b796e3afae3e1b2d07...
```

You can clearly see that the `moka faucet` command is actually calling the `setMaxFaucet` method of the gamete, passing `5000000000000000` as new value for the flow of the faucet. The caller (payer) and the receiver of the method invocation coincide, since they are both the gamete. The signature has been generated with the keys of the gamete.

If you check the corresponding `response.txt`, you will see something like:

```
VoidMethodCallTransactionSuccessfulResponse:
  gas consumed for CPU execution: 329
  gas consumed for RAM allocation: 1196
  gas consumed for storage consumption: 9590
  updates:

    <ee7a549a9419f6178efea6291121535efd71aa6c98233c89a4a0fae700a6efcc#0|
      io.takamaka.code.lang.Contract.balance:java.math.BigInteger|
      99999999999999999999999999999999998888500>

    <ee7a549a9419f6178efea6291121535efd71aa6c98233c89a4a0fae700a6efcc#0|
      io.takamaka.code.lang.ExternallyOwnedAccount.nonce:java.math.BigInteger|
      4>

    <ee7a549a9419f6178efea6291121535efd71aa6c98233c89a4a0fae700a6efcc#0|
      io.takamaka.code.lang.Gamete.maxFaucet:java.math.BigInteger|
      5000000000000000>

  events:
```

The response states clearly the cost of the transaction. Moreover, responses typically report a set of *updates*. These are the side-effects on the state of the node, induced by the transaction. Each update is a triple, that specifies a change in the value of a field of a storage object. In this case, the three updates state that the balance of the gamete has been decreased (because it paid for the transaction); that its nonce has been increased to four (since it ran the transaction); and that the `maxFaucet` field of the gamete has been set to 5000000000000000.

## Logs

All Hotmoka nodes generate a `hotmoka.log` log file, that reports which transactions have been processed and potential errors. Its content, in the case of a Tendermint node, looks like:

```
INFO: No roots found: the database is empty [05-05-2021 11:45:58]
INFO: Exodus environment created: chain/state [05-05-2021 11:45:58]
INFO: The Tendermint process is up and running [05-05-2021 11:46:00]
INFO: a18c0a...: posting (JarStoreInitialTransactionRequest) [05-05-2021 11:46:00]
INFO: a18c0a...: checking start [05-05-2021 11:46:00]
INFO: a18c0a...: checking success [05-05-2021 11:46:00]
INFO: a18c0a...: delivering start [05-05-2021 11:46:01]
INFO: a18c0a...: delivering success [05-05-2021 11:46:04]
INFO: 3cbaa2...: posting (GameteCreationTransactionRequest)
      [05-05-2021 11:46:04]
INFO: 3cbaa2...: checking start [05-05-2021 11:46:04]
INFO: 3cbaa2...: checking success [05-05-2021 11:46:04]
INFO: 3cbaa2...: checking start [05-05-2021 11:46:05]
INFO: 3cbaa2...: checking success [05-05-2021 11:46:05]
INFO: 3cbaa2...: delivering start [05-05-2021 11:46:06]
INFO: 3cbaa2...: delivering success [05-05-2021 11:46:06]
INFO: 6ed545...: posting (ConstructorCallTransactionRequest) [05-05-2021 11:46:07]
...
INFO: Store get cache hit rate: 0.0% [05-05-2021 11:46:15]
INFO: Exodus log cache hit rate: 36.7% [05-05-2021 11:46:15]
INFO: Time spent in state procedures: 138ms [05-05-2021 11:46:15]
INFO: Time spent checking requests: 8ms [05-05-2021 11:46:15]
INFO: Time spent delivering requests: 2213ms [05-05-2021 11:46:15]
INFO: The Tendermint process has been shut down [05-05-2021 11:46:15]
```

If you want to follow in real time what is happening inside your node, you can run for instance:

```
$ tail -f hotmoka.log
```

This will hang and print the new log entries as they are generated. Assuming that you have a local node running in your machine, try for instance in another shell

```
$ moka info
```

You will see in the log all new entries related to the execution of the methods to access the information on the node printed by `moka info`.

## Node Decorators

**[See project `runs` inside the `hotmoka_tutorial` repository]**

There are some frequent actions that can be performed on a Hotmoka node. Typically, these actions consist in a sequence of transactions. A few examples are:

1. The creation of an externally owned account. This requires the creation of its private and public keys and the instantiation of an `io.takamaka.code.lang.ExternallyOwnedAccount`. It is not a difficult procedure, but it is definitely tedious and occurs frequently.
2. The installation of a jar in a node. This requires a transaction for installing code in the node. It requires also to parse the jar into bytes and identify the number of gas units for the transaction, depending on the size of the jar.
3. The initialization of a node. Namely, local nodes start empty, that is, their store does not contain anything at the beginning, not even their manifest object. This initialization is rather technical and detail might change in future versions of Hotmoka. Performing this initialization by hand leads to fragile and error-prone code.

In all these examples, Hotmoka provides decorators, that is, implementation of the `Node` interface built from an existing `Node` object. The decorator is just an alias to the decorated node, but adds some functionality or performs some action on it. Figure 32 shows that there are decorators for each of the three situations enumerated above.

In order to understand the use of node decorators and appreciate their existence, let us write a Java class that creates a `MemoryBlockchain` node, hence initially empty; then it initializes the node; subsequently it installs our `family-0.0.1.jar` file in the node and finally creates two accounts in the node. We stress the fact that these actions can be performed in code by using calls to the node interface (Figure 32); they can also be performed through the `moka` tool. Here, however, we want to perform them in code, simplified by using node decorators.

Create the following `Decorators.java` class inside the `runs` package of the `runs` project:

```
package runs;

import java.math.BigInteger;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.security.KeyPair;

import io.hotmoka.constants.Constants;
import io.hotmoka.crypto.Base58;
import io.hotmoka.crypto.Entropy;
import io.hotmoka.crypto.SignatureAlgorithmForTransactionRequests;
import io.hotmoka.helpers.InitializedNode;
import io.hotmoka.helpers.NodeWithAccounts;
import io.hotmoka.helpers.NodeWithJars;
import io.hotmoka.memory.MemoryBlockchain;
import io.hotmoka.memory.MemoryBlockchainConfig;
import io.hotmoka.nodes.ConsensusParams;
import io.hotmoka.nodes.Node;

public class Decorators {
  public final static BigInteger GREEN_AMOUNT = BigInteger.valueOf(1_000_000_000);
  public final static BigInteger RED_AMOUNT = BigInteger.ZERO;

  public static void main(String[] args) throws Exception {
    MemoryBlockchainConfig config = new MemoryBlockchainConfig.Builder().build();
    ConsensusParams consensus = new ConsensusParams.Builder().build();

    // the path of the runtime Takamaka jar, inside Maven's cache
```

```
    Path takamakaCodePath = Paths.get
      (System.getProperty("user.home") +
      "/.m2/repository/io/hotmoka/io-takamaka-code/" + Constants.VERSION +
      "/io-takamaka-code-" + Constants.VERSION + ".jar");

    // the path of the user jar to install
    Path familyPath = Paths.get("../family/target/family-0.0.1.jar");

    // create a key pair for the gamete and compute the Base58-encoding of its public key
    var signature = SignatureAlgorithmForTransactionRequests.ed25519();
    Entropy entropy = new Entropy();
    KeyPair keys = entropy.keys("password", signature);
    var publicKeyBase58 = Base58.encode(signature.encodingOf(keys.getPublic()));

    try (Node node = MemoryBlockchain.init(config, consensus)) {
      // first view: store the io-takamaka-code jar and create manifest and gamete
      InitializedNode initialized = InitializedNode.of
        (node, consensus, publicKeyBase58, takamakaCodePath, GREEN_AMOUNT, RED_AMOUNT);

      // second view: store family-0.0.1.jar: the gamete will pay for that
      NodeWithJars nodeWithJars = NodeWithJars.of
        (node, initialized.gamete(), keys.getPrivate(), familyPath);

      // third view: create two accounts, the first with 10,000,000 units of coin
      // and the second with 20,000,000 units of coin; the gamete will pay
      NodeWithAccounts nodeWithAccounts = NodeWithAccounts.of
        (node, initialized.gamete(), keys.getPrivate(),
        BigInteger.valueOf(10_000_000), BigInteger.valueOf(20_000_000));

      System.out.println("manifest: " + node.getManifest());
      System.out.println("family-0.0.1.jar: " + nodeWithJars.jar(0));
      System.out.println("account #0: " + nodeWithAccounts.account(0) +
                         "\n  with private key " + nodeWithAccounts.privateKey(0));
      System.out.println("account #1: " + nodeWithAccounts.account(1) +
                         "\n  with private key " + nodeWithAccounts.privateKey(1));
    }
  }
}
```

Run class `Decorators` from Eclipse. It should print something like this on the console:

```
manifest: 5f1ebc34f4aef10e2c2eeac3558aae7d4df97f676f29ba9d7e28d0d1713c5ad5#0
family-0.0.1.jar: 7d6b33133647f0c84cc9550cc0010eab35329e0822df9706...
account #0: 64fd4337475541ed2aeb3d49149603142b5ec275d41bfc9ec29555c41739ea8e#0
  with private key Ed25519 Private Key [ab:69:96:b0:9c:24:6d:a2:d2:d9:97:b4:...]
    public data: 4e1d5299f31e19315e4f59c3ade35a8b8f1d1bf5feb9b042c349cc5e051e8e55

account #1: f0840b73741d3fceefc4e87a4d055a7044dbcbdeb8213636c0d810eba4cf60cc#0
  with private key Ed25519 Private Key [cb:a5:ce:79:9b:98:25:3c:4d:44:7b:93:...]
    public data: 46d9cbcbad683d1d21079558a20fbfb7c1feb6f9c07e33c0288d939df5...
```

You can see that the use of decorators has avoided us the burden of programming transaction

requests, explicitly, and makes our code more robust, since future versions of Hotmoka will update the implementation of the decorators, while their interface will remain untouched, protecting our code from modifications.

As we have already said, decorators are views of the same node, just seen through different lenses (Java interfaces). Hence, further transactions can be run on `node` or `initialized` or `nodeWithJars` or `nodeWithAccounts`, with the same effects. Moreover, it is not necessary to close all such nodes: closing `node` at the end of the try-with-resource will actually close all of them, since they are the same node.

## Publishing a Hotmoka Node Online

**[See project `runs` inside the `hotmoka_tutorial` repository]**

This section shows how we can publish a Hotmoka node online, so that it becomes a network service that can be used, concurrently, by many remote users. Namely, we will show how to publish a blockchain node based on Tendermint, but the code is similar if you want to publish a node based on a memory blockchain or any other Hotmoka node.

Remember that we have already published our nodes online, by using the `moka init-tendermint` and `moka init-memory` commands. Here, however, we want to do the same operation in code.

Create a class `Publisher.java` inside package `runs` of the `runs` project, whose code is the following:

```
package runs;

import static java.math.BigInteger.ZERO;

import java.math.BigInteger;
import java.nio.file.Path;
import java.nio.file.Paths;
import java.security.KeyPair;

import io.hotmoka.constants.Constants;
import io.hotmoka.crypto.Base58;
import io.hotmoka.crypto.Entropy;
import io.hotmoka.crypto.SignatureAlgorithmForTransactionRequests;
import io.hotmoka.helpers.InitializedNode;
import io.hotmoka.nodes.ConsensusParams;
import io.hotmoka.nodes.Node;
import io.hotmoka.service.NodeService;
import io.hotmoka.service.NodeServiceConfig;
import io.hotmoka.tendermint.TendermintBlockchain;
import io.hotmoka.tendermint.TendermintBlockchainConfig;

public class Publisher {
  public final static BigInteger GREEN_AMOUNT = BigInteger.valueOf(100_000_000);
  public final static BigInteger RED_AMOUNT = ZERO;

  public static void main(String[] args) throws Exception {
    TendermintBlockchainConfig config = new TendermintBlockchainConfig.Builder().build();
    ConsensusParams consensus = new ConsensusParams.Builder().build();
```

```
    NodeServiceConfig serviceConfig = new NodeServiceConfig.Builder().build();
    // the path of the runtime Takamaka jar, inside Maven's cache
    Path takamakaCodePath = Paths.get
      (System.getProperty("user.home") +
      "/.m2/repository/io/hotmoka/io-takamaka-code/" + Constants.VERSION +
      "/io-takamaka-code-" + Constants.VERSION + ".jar");

    // create a key pair for the gamete and compute the Base58-encoding of its public key
    var signature = SignatureAlgorithmForTransactionRequests.ed25519();
    Entropy entropy = new Entropy();
    KeyPair keys = entropy.keys("password", signature);
    var publicKeyBase58 = Base58.encode(signature.encodingOf(keys.getPublic()));

    try (Node original = TendermintBlockchain.init(config, consensus);
         // remove the next three lines if you want to publish an uninitialized node
         //InitializedNode initialized = InitializedNode.of
         //  (original, consensus, publicKeyBase58,
         //  takamakaCodePath, GREEN_AMOUNT, RED_AMOUNT);
         NodeService service = NodeService.of(serviceConfig, original)) {

        System.out.println("\nPress ENTER to turn off the server and exit this program");
        System.in.read();
    }
  }
}
```

We have already seen that `original` is a Hotmoka node based on Tendermint. The subsequent line makes the feat:

```
NodeService service = NodeService.of(serviceConfig, original);
```

Variable `service` holds a Hotmoka *service*, that is, an actual network service that adapts the `original` node to a web API that is published at localhost, at port 8080 (another port number can be selected through the `serviceConfig` object, if needed). The service is an `AutoCloseable` object: it starts when it is created and gets shut down when its `close()` method is invoked, which occurs, implicitly, at the end of the scope of the try-with-resources. Hence, this service remains online until the user presses the ENTER key and terminates the service (and the program).

Run class `Publisher` from Eclipse. It should run for a few seconds and then start waiting for the ENTER key. Do not press such key yet! Instead, enter the following URL into a browser running in your machine:

```
http://localhost:8080/get/nameOfSignatureAlgorithmForRequests
```

You should see the following response in your browser:

```
{"algorithm":"ed25519"}
```

What we have achieved, is to call the method `getNameOfSignatureAlgorithmForRequests()` of `original`, accessible through the network service.

Let us try to ask for the storage address of the manifest of the node. Again, insert the following URL in a browser on your local machine:

```
http://localhost:8080/get/manifest
```

This time, the response is negative:

```
{"message":"no manifest set for this node",
 "exceptionClassName":"java.util.NoSuchElementException"}
```

We have called the method `getManifest()` of `original`, through the network service. Since `original` is not initialized yet, it has no manifest and no gamete. Its store is just empty at the moment. Hence the negative response.

Thus, let us initialize the node before publishing it, so that it is already initialized when published. Press ENTER to terminate the service, then modify the `Publisher.java` class by uncommenting the use of the `InitializedNode` decorator, whose goal is to create manifest and gamete of the node and install the basic classes of Takamaka inside the node.

Note that we have published `original`:

```
NodeService service = NodeService.of(serviceConfig, original);
```

We could have published `initialized` instead:

```
NodeService service = NodeService.of(serviceConfig, initialized);
```

The result would be the same, since both are views of the same node object. Moreover, note that we have initialized the node inside the try-with-resource, before publishing the service as the last of the three resources. This ensures that the node, when published, is already initialized. In principle, publishing an uninitialized node, as done previously, exposes to the risk that somebody else might initialize the node, hence taking its control since he will set the keys of the gamete.

If you re-run class `Publisher` and re-enter the last URL in a browser on your local machine, the response will be positive this time:

```
{
  "transaction":
  {
    "type":"local",
    "hash":"f9ac8849f7ee484d73fd84470652582cf93da97c379fee9ccc66bd5e2ffc9867"
  },
  "progressive":"0"
}
```

This means that the manifest is allocated, in the store of `original`, at the storage reference `f9ac8849f7ee484d73fd84470652582cf93da97c379fee9ccc66bd5e2ffc9867#0`.

A Hotmoka node, once published, can be accessed by many users, *concurrently*. This is not a problem, since Hotmoka nodes are thread-safe and can be used in parallel by many users. Of course, this does not mean that there are no race conditions at the application level. As a simple example, if two users operate with the same paying externally owned account, their wallets might suffer from race conditions on the nonce of the account and they might see requests rejected because of an incorrect nonce. The situation is the same here as in Ethereum, for instance. In practice, each externally owned account should be controlled by a single wallet at a time.

## Publishing a Hotmoka Node on Amazon EC2

We have published a Hotmoka node on our machine (the localhost). This might not be the best place where the node should be published, since our machine might not allow external connections from the internet and since we might want to turn it off after we stop working with it. In reality, a node should be published on a machine that can receive external connections and that is always on, at least for a long period. There are many solutions for that. Here, we describe the simple technique of using a rented machine from Amazon EC2's computing cloud [EC2]. This service offers a micro machine for free, while more powerful machines require one to pay for their use. Since the micro machine is enough for our purposes, EC2 is a good candidate for experimentation.

Let us publish a Tendermint node then. We will show how to do it with the `moka` tool, although it is possible to do the same in code as well.

Perform then the following steps in order to publish a node online with Amazon EC2:

1. Turn on an Amazon EC2 machine from the AWS console.
2. Edit the inbound rules of the security group of the machine, from the AWS console, so that its port 8080 is open for every incoming TCP connection.
3. Log into the EC2 machine (use the address of your machine below):

```
$ ssh -i your.pem ubuntu@ec2-99-80-8-84.eu-west-1.compute.amazonaws.com
```

4. Install the Java Runtime Environment in the machine, at least version 11.
5. Install Tendermint in the machine, so that it can be reached from the command-path, as explained in Tendermint Nodes.
6. Run the `screen` command in the EC2 machine. It will allow you later to exit the remote shell and leave the `moka` process running in the background.
7. Install Hotmoka in the EC2 machine, as in Installation of Hotmoka.
8. Start the server in the Amazon machine, as done in Tendermint Nodes. Add the `--open-unsigned-faucet` option if you want to be able to open a free faucet of coins. Use a public key for the gamete, that you have created on another machine, so that the private key will never be inside the machine where the node is running.
9. Leave the server running in the background and exit the Amazon machine:

```
$ CTRL-a d
$ exit
```

10. In the other machine, that you use for controlling the node, bind the key of the gamete to the storage reference of the gamete:

```
$ bind-key key --url ec2-99-80-8-84.eu-west-1.compute.amazonaws.com:8080
```

You can verify that the EC2 server is accessible from outside if you direct your local browser to your machine (use the address of your machine below):

```
http://ec2-99-80-8-84.eu-west-1.compute.amazonaws.com:8080/get/manifest
```

The response should be something like:

```
{"transaction":{"type":"local",
    "hash":"21d375ae9bac3b74d1a54a6418f7c70c2c107665fb2066a94dbf65cb3db9cdc6"},
  "progressive":"0"}
```

# Remote Nodes

**[See project `runs` inside the `hotmoka_tutorial` repository]**

We have seen how a service can be published and its methods called through a browser. This has been easy for methods such as `getManifest()` and `getNameOfSignatureAlgorithmForRequest()` of the interface `Node`. However, it becomes harder if we want to call methods of `Node` that need parameters, such as `getState()` or the many `add/post/run` methods for scheduling transactions on the node. Parameters should be passed as JSON payload of the http connection, in a format that is hard to remember, easy to get wrong and possibly changing in the future. Moreover, the JSON responses must be parsed back. In principle, this can be done by hand or through software that builds the requests for the server and interprets its responses. Nevertheless, it is not the suggested way to proceed.

A typical solution to this problem is to provide a software SDK, that is, a library that takes care of serializing the requests into JSON and deserializing the responses from JSON. Roughly speaking, this is the approach taken in Hotmoka. More precisely, as this section will show, we can forget about the details of the JSON serialization and deserialization of requests and responses and only program against the `Node` interface, by using an adaptor of a published Hotmoka service into a `Node`. This adaptor is called a *remote* Hotmoka node.

We have used remote nodes from the very beginning of this tutorial. Namely, if you go back to Installation of the Jar in a Node, you will see that we have built a Hotmoka node from a remote service:

```
RemoteNodeConfig config = new RemoteNodeConfig.Builder()
  .setURL("panarea.hotmoka.io")
  .build();

try (Node node = RemoteNode.of(config)) {
  ...
}
```

The `RemoteNode.of(...)` method adapts a remote service into a Hotmoka node, so that we can call all methods of that (Figure 32).

By default, a remote node connects to a service by using the HTTP protocol, but handles event notification by using web sockets. This is automatic and you do not need to understand the details of this connection. It is possible to use web sockets for *all* communications, also those of the many `get/add/post/run` methods of the `Node` interface. For that, you can set a flag in the configuration of the remote node, as follows:

```
RemoteNodeConfig config = new RemoteNodeConfig.Builder()
  .setURL("panarea.hotmoka.io")
  .setWebSockets(true)
  .build();
```

Nevertheless, there is currently no actual benefit in using web sockets for all communications. Thus, we suggest to stick to the default configuration, that uses web sockets only for event notification to the subscribed event handlers.

## Creating Sentry Nodes

We have seen that a `Node` can be published as a Hotmoka service: on a machine `my.validator.com` we can execute:

```
TendermintBlockchainConfig config = new TendermintBlockchainConfig.Builder().build();
ConsensusParams consensus = new ConsensusParams.Builder().build();
NodeServiceConfig serviceConfig = new NodeServiceConfig.Builder().build();

try (Node original = TendermintBlockchain.init(config, consensus);
     NodeService service = NodeService.of(serviceConfig, original)) {
  ...
}
```

The service will be available on the internet as

```
http://my.validator.com:8080
```

Moreover, on another machine `my.sentry.com`, that Hotmoka service can be adapted into a remote node that, itself, can be published on that machine:

```
NodeServiceConfig serviceConfig = new NodeServiceConfig.Builder().build();
RemoteNodeConfig config = new RemoteNodeConfig.Builder()
  .setURL("my.validator.com:8080")
  .build();

try (Node validator = RemoteNode.of(config);
     NodeService service = NodeService.of(serviceConfig, validator)) {
  ...
}
```

The service will be available at

```
http://my.sentry.com:8080
```

We can continue this process as much as we want, but let us stop at this point. Programmers can connect to the service published at `http://my.sentry.com:8080` and send requests to it. That

service is just a bridge that forwards everything to the service at `http://my.validator.com:8080`. It might not be immediately clear why this intermediate step could be useful or desirable. The motivation is that we could keep the (precious) validator machine under a firewall that allows connections with `my.sentry.com` only. As a consequence, in case of DOS attacks, the sentry node will receive the attack and possibly crash, while the validator continues to operate as usual: it will continue to interact with the other validators and take part in the validation of blocks. Moreover, since many sentries can be connected to a single validator, the latter remains accessible through the other sentries, if needed. This is an effective way to mitigate the problem of DOS attacks to validator nodes.

The idea of sentry nodes against DOS attacks is not new and is used, for instance, in Cosmos networks [Sentry]. However, note how it is easy, with Hotmoka, to build such a network architecture by using network services and remote nodes.

## Signatures and Quantum-Resistance

Hotmoka is agnostic wrt. the algorithm used for signing requests. This means that it is possible to deploy Hotmoka nodes that sign requests with distinct signature algorithms. Of course, if nodes must re-execute the same transactions, such as in the case of a blockchain, then all nodes of the blockchain must use the same algorithm for the transactions signed by each given account, or otherwise they will not be able to reach consensus. Yet, any algorithm can be chosen for the blockchain. In principle, it is even possible to use an algorithm that does not sign the transactions, if the identity of the callers of the transactions needn't be verified. However, this might be sensible in private networks only.

The default signature algorithm used by a node is specified at construction time, as a configuration parameter. For instance, the code

```
TendermintBlockchainConfig config = new TendermintBlockchainConfig.Builder()
                                        .signRequestsWith("ed25519")
                                        .build();

try (Node node = TendermintBlockchain.of(config, consensus)) {
  ...
}
```

starts a Tendermint-based blockchain node that uses the ed25519 signature algorithm as default signature algorithm for the requests. Requests sent to that node can be signed as follows:

```
// recover the algorihm used by the node
SignatureAlgorithm<SignedTransactionRequest> signature
  = SignatureAlgorithmForTransactionRequests.mk
       (node.getNameOfSignatureAlgorithmForRequests());

// create a key pair for that algorithm
KeyPair keys = signature.getKeyPair();

// create a signer object with the private key of the key pair
Signer signer = Signer.with(signature, keys.getPrivate());

// create an account having public key keys.getPublic()
....
```

```
// create a transaction request on behalf of the account
ConstructorCallTransactionRequest request
  = new ConstructorCallTransactionRequest(signer, account, ...);

// send the request to the node
node.addConstructorCallTransaction(request);
```

In the example above, we have explicitly specified to use ed25519 as default signature algorithm. That is what is chosen if nothing is specified at configuration-time. Consequently, there is no need to specify that algorithm in the configuration object and that is why we never did it in the previous chapters. It is possible to configure nodes with other default signature algorithms. For instance:

```
TendermintBlockchainConfig config = new TendermintBlockchainConfig.Builder()
                                        .signRequestsWith("sha256dsa")
                                        .build();
```

configures a node that uses sha256dsa as default signature algorithm, while

```
TendermintBlockchainConfig config = new TendermintBlockchainConfig.Builder()
                                        .signRequestsWith("empty")
                                        .build();
```

configures a node that uses the empty signature as default signature algorithm; it is an algorithm that accepts all signatures, in practice disabling any signature checking.

It is possible to specify a quantum-resistant signature algorithm as default, that is, one that belongs to a family of algorithms that are expected to be immune from attacks performed through a quantistic computer. For instance,

```
TendermintBlockchainConfig config = new TendermintBlockchainConfig.Builder()
                                        .signRequestsWith("qtesla1")
                                        .build();
```

configures a node that uses the quantum-resistant qtesla-p-I algorithm as default signature algorihtm, while

```
TendermintBlockchainConfig config = new TendermintBlockchainConfig.Builder()
                                        .signRequestsWith("qtesla3")
                                        .build();
```

configures a node that uses the quantum-resistant qtesla-p-III algorithm as default signature algorihtm, that is expected to be more resistant than qtesla-p-I but has larger signatures than qtesla-p-I.

Quantum-resistance is an important aspect of future-generation blockchains. However, at the time of this writing, a quantum attack is mainly a theoretical possibility, while the large size of quantum-resistant keys and signatures is already a reality and a node using a qtesla signature algorithm *as default* might exhaust the disk space of your computer very quickly. In practice, it is better to use a quantum-resistant signature algorithm only for a subset of the transactions, whose quantum-resistance is deemed important. Instead, one should use a lighter algorithm (such as the default ed25519) for all other transactions. This is possible because Hotmoka nodes allow one to mix transactions signed with distinct algorithms. Namely, one can use ed25519 as default algorithm, for all transactions signed

by instances of `ExternallyOwnedAccounts`, with the exception of those transactions that are signed by instances of `AccountQTESLA1`, such as `ExternallyOwnedAccountQTESLA1`, or of `AccountQTESLA3`, such as `ExternallOwnedAccountQTESLA3`, or of `AccountSHA256DSA`, such as `ExternallOwnedAccountSHA256DSA` (see Figure 23). Namely, if the caller of a transaction is an `AccountQTESLA1`, then the request of the transaction must be signed with the qtesla-p-I algorithm. If the caller of a transaction is an `AccountQTESLA3`, then the request of the transaction must be signed with the qtesla-p-III algorithm. If the caller of a transaction is an `AccountSHA256DSA`, then the request of the transaction must be signed with the sha256dsa algorithm. If the caller of a transaction is an `AccountED25519`, then the request of the transaction must be signed with the ed25519 algorithm. In practice, this allows specific transactions to override the default signature algorithm for the node.

For instance, let us create an account using the default signature algorithm for the node. We charge its creation to the faucet of the node:

```
$ moka create-account 1000000000000 --payer faucet --url panarea.hotmoka.io

Please specify the password of the new account: game
...
A new account 43d9e576b03706079ac617ff62430ab3691c75f247d264e33c2a9a0986507c64#0
has been created
```

You can check the class of the new account with the `moka state` command:

```
$ moka state 43d9e576b03706079ac617ff62430ab3691c75f247d264e33c2a9a0986507c64#0
    --url panarea.hotmoka.io

...
class io.takamaka.code.lang.ExternallyOwnedAccountED25519 ...
  publicKey:java.lang.String =
    "shA7+XygJ5Wc+WPccFPis6TLbWxqWVnR3eNTJradf5c="
  balance:java.math.BigInteger = 1000000000000
...
```

As you can see, an account has been created, that uses the default `ed25519` signature algorithm of the node. Assume that we want to create an account now, that *always* uses the `sha256dsa` signature algorithm, regardless of the default signature algorithm of the node. We can specify that to `moka create-account`:

```
$ moka create-account 1000000000000
    --payer faucet --signature sha256dsa --url panarea.hotmoka.io

Please specify the password of the new account: play
...
A new account 1215c77ebec338cc31767b129095708837645fb25e491b98c60a2f6415995d4a#0
has been created
```

This creation has been more expensive, because the public key of the sha256dsa algorithm is much longer than that for the ed25519 algorithm. You can verify this with the `moka state` command:

```
$ moka state 1215c77ebec338cc31767b129095708837645fb25e491b98c60a2f6415995d4a#0
    --url panarea.hotmoka.io

...
class io.takamaka.code.lang.ExternallyOwnedAccountSHA256DSA ...
  publicKey:java.lang.String = "MIIDQjCCAjUGByqGSM44BAEwggIoAo..."
  balance:java.math.BigInteger = 1000000000000000
  ...
```

Note that the class of the account is `ExternallyOwnedAccountSHA256DSA` this time.

Let us create an account that uses the qtesla-p-I signature algorithm now:

```
$ moka create-account 1000000000000
    --payer faucet --signature qtesla1 --url panarea.hotmoka.io

Please specify the password of the new account: quantum1
...
Total gas consumed: 5294043
...
A new account a7e878c6965109d1bf8ec4c9635b3513bfc2055a298e840be5badf8bd166d335#0
has been created
```

The creation of this account has been very expensive, since quantum-resistant keys are very large. Again, you can use the `moka state` command to verify that it has class `ExternallyOwnedAccountQTESLA1`.

Finally, let us use the previous qtesla-p-I account to create a qtesla-p-III account:

```
$ moka create-account 100000
    --payer a7e878c6965109d1bf8ec4c9635b3513bfc2055a298e840be5badf8bd166d335#0
    --signature qtesla3 --url panarea.hotmoka.io

Please specify the password of the payer account: quantum1
Please specify the password of the new account: quantum3
...
Total gas consumed: 5294170
...
A new account 9437ba8c5c29edb48e9fd0bba1a54e0286a84609d2c13391a8fdf5d9b25bff68#0
has been created
```

Note, again, the extremely high gas cost of this creation.

Regardless of the kind of account, their use it always the same. The only difference is to use the right signature algorithm when signing a transaction, since it must match that of the caller account. This is automatic, if we use the `moka` tool. For instance, let us use our qtesla-p-I account to install the `family-0.0.1.jar` code in the node:

```
$ cd hotmoka_tutorial
$ moka install family/target/family-0.0.1.jar
    --payer a7e878c6965109d1bf8ec4c9635b3513bfc2055a298e840be5badf8bd166d335#0
    --url panarea.hotmoka.io

Please specify the password of the payer account: quantum1
Do you really want to spend up to 696900 gas units to install the jar [Y/N] Y
family/target/family-0.0.1.jar has been installed
at 24681fa7eb8aa247e184ec6e9490625becb80b9c8604e12670481ea169da0ce2
...
```

The `moka` tool has understood that the payer is an account that signs with the qtesla-p-I algorithm and has signed the request accordingly.

Chapter 7

# Tokens

A popular class of smart contracts implement a dynamic ledger of coin transfers between accounts. These coins are not native tokens, but rather new, derived tokens. In some sense, tokens are programmed money, whose rules are specified by a smart contract and enforced by the underlying blockchain.

> In this context, the term *token* is used for the smart contract that tracks coin transfers, for the single coin units and for the category of similar coins. This is sometime confusing.

Native and derived tokens can be categorized in many ways [OliveiraZBS18,Freni20,Tapscott20]. The most popular classification is between *fungible* and *non-fungible* tokens. Fungible tokens are interchangeable with each other, since they have an identical nominal value that does not depend on each specific token instance. Native tokens and traditional (*fiat*) currencies are both fungible tokens. Their main application is in the area of crowdfunding and initial coin offers to support startups. On the contrary, non-fungible tokens have a value that depends on their specific instance. Hence, in general, they are not interchangeable. Their main application is currently in the art market, where they represent a written declaration of author's rights concession to the holder.

A few standards have emerged for such tokens, that should guarantee correctness, accessibility, interoperability, management and security of the smart contracts that run the tokens. Among them, the Ethereum Requests for Comment #20 (ERC-20, see https://eips.ethereum.org/EIPS/eip-20) and #721 (ERC-721, see https://eips.ethereum.org/EIPS/eip-721) are the most popular, also outside Ethereum. They provide developers with a list of rules required for the correct integration of tokens with other smart contracts and with applications external to the blockchain, such as wallets, block explorers, decentralized finance protocols and games.

The most popular implementations of the ERC-20 and ERC-721 standards are in Solidity, by OpenZeppelin (see https://docs.openzeppelin.com/contracts/2.x/erc20 and https://docs.openzeppelin.com/contracts/2.x/erc721), a team of programmers in the Ethereum community who deliver useful and secure smart contracts and libraries, and by ConsenSys, later deprecated in favor of OpenZeppelin's. OpenZeppelin extends ERC-20 with snapshots, that is, immutable views of the state of a token contract, that show its ledger at a specific instant of

time. They are useful to investigate the consequences of an attack, to create forks of the token and to implement mechanisms based on token balances such as weighted voting.

## Fungible Tokens (ERC20)

A fungible token ledger is a ledger that binds owners (contracts) to the numerical amount of tokens they own. With this very high-level description, it is an instance of the `IERC20View` interface in Figure 33. The `balanceOf` method tells how many tokens an `account` holds and the method `totalSupply` provides the total number of tokens in circulation. The `UnsignedBigInteger` class is a Takamaka library class that wraps a `BigInteger` and guarantees that its value is never negative. For instance, the subtraction of two `UnsignedBigInteger`s throws an exception when the second is larger than the first.

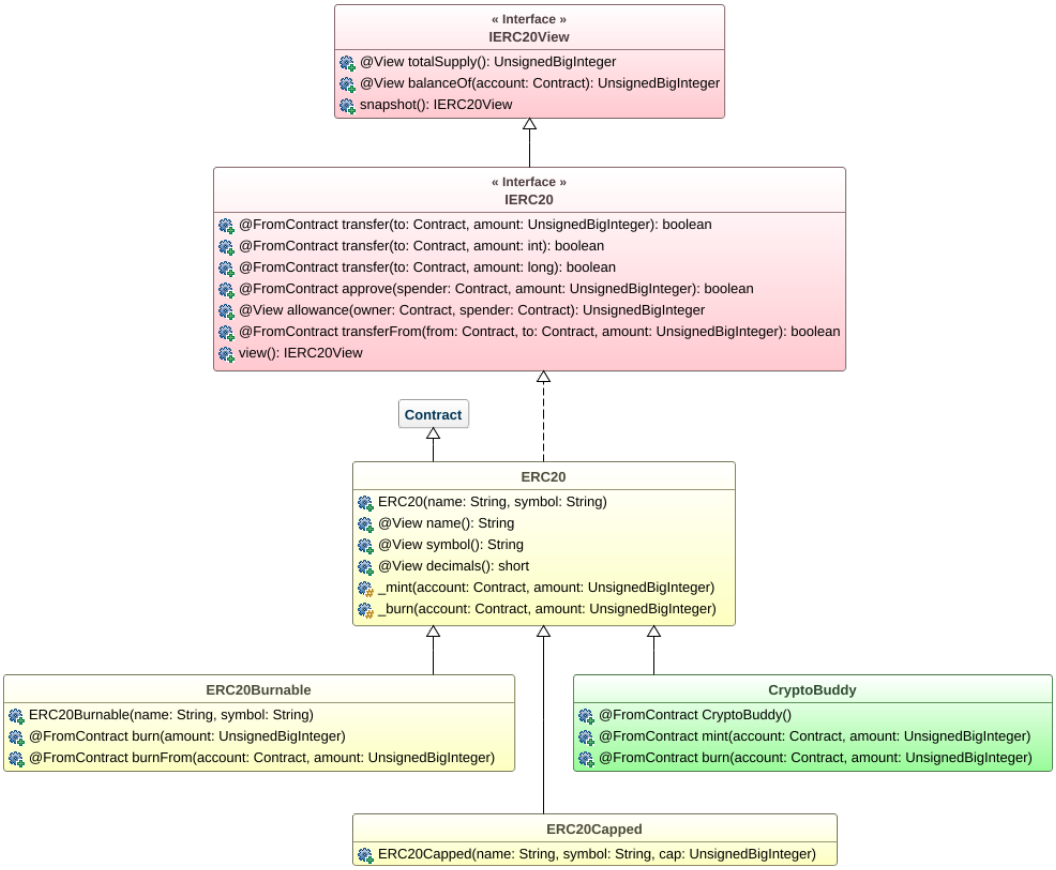



*Figure 33. The hierarchy of the ERC20 token implementations.*

The `snapshot` method, as already seen for collection classes, yields a read-only, frozen view of the token ledger. Since it is defined in the topmost interface, all token classes can be snapshotted. Snapshots are computable in constant time and their construction does not affect other users of the ledger.

In the original ERC20 standard and implementation in Ethereum, only specific subclasses allow snapshots, since their creation adds gas costs to all operations, also for token owners that never performed any snapshot. See the arguments and comparison in [CrosaraOST21].

An ERC20 ledger is typically modifiable. Namely, owners can sell tokens to other owners and can delegate trusted contracts to transfer tokens on their behalf. Of course, these operations must be legal, in the sense that a owner cannot sell more tokens than it owns and delegated contracts cannot transfer more tokens than the cap to their delegation. These modification operations are defined in the `IERC20` interface in Figure 33. They are identical to the same operations in the ERC20 standard for Ethereum, hence we refer to that standard for further detail. The `view()` method is used to yield a *view* of the ledger, that is, an object that reflects the current state of the original ledger, but without any modification operation.

The `ERC20` implementation provides a standard implementation for the functions defined in the `IERC20View` and `IERC20` interfaces. Moreover, it provides metadata information such as the name, symbol and number of decimals for the specific token implementation. There are protected implementations for methods that allow one to mint or burn an amount of tokens for a given owner (`account`). These are protected since one does not want to allow everybody to print or burn money. Instead, subclasses can call into these methods in their constructor, to implement an initial distribution of tokens, and can also allow subsequent, controlled mint or burns. For instance, the `ERC20Burnable` class is an `ERC20` implementation that allows a token owner to burn its tokens only, or those it has been delegated to transfer, but never those of another owner.

The `ERC20Capped` implementation allows the specification of a maximal cap to the number of tokens in circulation. When new tokens get minted, it checks that the cap is not exceeded and throws an exception otherwise.

## Implementing Our Own ERC20 Token

**[See project `erc20` inside the `hotmoka_tutorial` repository]**

Let us define a token ledger class that allows only its creator the mint or burn tokens. We will call it `CryptoBuddy`. As Figure 33 shows, we plug it below the `ERC20` implementation, so that we inherit that implementation and do not need to reimplement the methods of the `ERC20` interface.

Create in Eclipse a new Maven Java 11 (or later) project named `erc20`. You can do this by duplicating the project `family` (make sure to store the project inside the `hotmoka_tutorial` directory, as a sibling of `family`, `ponzi`, `tictactoe` and so on). Use the following `pom.xml`:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>io.hotmoka</groupId>
  <artifactId>erc20</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>1.0.7</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>
```

and the following `module-info.java`:

```
module erc20 {
  requires io.takamaka.code;
}
```

Create package `io.takamaka.erc20` inside `src/main/java` and add the following `CryptoBuddy.java` inside that package:

```
package io.takamaka.erc20;

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

import io.takamaka.code.lang.Contract;
import io.takamaka.code.lang.FromContract;
import io.takamaka.code.math.UnsignedBigInteger;
import io.takamaka.code.tokens.ERC20;

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 mint(Contract account, UnsignedBigInteger amount) {
    require(caller() == owner, "Lack of permission");
    _mint(account, amount);
  }

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

The constructor of `CryptoBuddy` initializes the total supply by minting a very large number of tokens. They are initially owned by the creator of the contract, that is saved as `owner`. Methods `mint` and `burn` check that the owner is requesting the mint or burn and call the inherited protected methods in that case.

You can generate the `erc20-0.0.1.jar` file:

```
$ cd erc20
$ mvn package
```

Then you can install that jar in the node, by letting our first account pay:

```
$ cd ..
$ moka install erc20/target/erc20-0.0.1.jar
    --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
    --url panarea.hotmoka.io

Please specify the password of the payer account: chocolate
Do you really want to spend up to 504100 gas units to install the jar [Y/N] Y
erc20/target/erc20-0.0.1.jar has been installed at
ff3c79e01bdfd37afdd7b9bec052caf5f012e7a58f6a83931cfdfb88d42cb6af
Total gas consumed: 244568
  for CPU: 262
  for RAM: 1303
  for storage: 243003
  for penalty: 0
```

Finally, you can create an instance of the token class, by always letting our first account pay for that:

```
$ moka create
  io.takamaka.erc20.CryptoBuddy
  --payer 75af93866a41581c0aa2dd0ab33ac8790637c6dfc759a7bbd8cf97a43ca32be0#0
  --classpath ff3c79e01bdfd37afdd7b9bec052caf5f012e7a58f6a83931cfdfb88d42cb6af
  --url panarea.hotmoka.io

Please specify the password of the payer account: chocolate
Do you really want to spend up to 500000 gas units to call CryptoBuddy() ? [Y/N] Y
The new object has been allocated at
bb3c76396f27d8402e9252a9194c69ed635f7a2de2385a7f89c0344e6083c8ba#0
Total gas consumed: 129369
  for CPU: 1314
  for RAM: 2843
  for storage: 125212
  for penalty: 0
```

The new ledger instance is installed in the storage of the node now, at the address `bb3c76396f27d8402e9252a9194c69ed635f7a2de2385a7f89c0344e6083c8ba#0`. It is possible to start interacting with that ledger instance, by trasferring tokens between accounts. For instance, this can be done with the `moka call` command, that allows one to invoke the `transfer` or `transferFrom` methods of the ledger. It is possible to show the state of the ledger with the `moka state` command, although specific utilities will provide a more user-friendly view of the ledger in the future.

## Richer than Expected

Every owner of ERC20 tokens can decide to send some of its tokens to another contract $C$, that will become an owner itself, if it was not already. This means that the ledger inside an `ERC20` implementation gets modified and some tokens get registered for the new owner $C$. However, $C$ is not notified in any way of this transfer. This means that our contracts could be richer than we expect, if somebody has sent tokens to them, possibly inadvertently. In theory, we could scan the whole memory of a Hotmoka node, looking for implementations of the `IERC20` interface, and check if our contracts are registered inside them. Needless to say, this is computationally irrealistic. Moreover, even if we know that one of our contracts is waiting to receive some tokens, we don't know immediately when this happens, since the contract does not get notified of any transfer of tokens.

This issue is inherent to the definition of the ERC20 standard in Ethereum and the implementation in Takamaka inherits this limitation, since it wants to stick as much as possible to the Ethereum standard. A solution to the problem would be to restrict the kind of owners that are allowed in Figure 33. Namely, instead of allowing all `Contract`s, the signature of the methods could be restricted to owners of some interface type `IERC20Receiver`, with a single method `onReceive` that gets called by the `ERC20` implementation, every time tokens get transferred to an `IERC20Receiver`. In this way, owners of ERC20 tokens get notified when they receive new tokens. This solution has never been implemented for ERC20 tokens in Ethereum, while it has been used in the ERC721 standard for non-fungible tokens, as we will show in the next section.

## Non-Fungible Tokens (ERC721)

A non-fungible token is implemented as a ledger that maps each token identifier to its owner. Ethereum provides the ERC721 specification for non-fungible tokens. There, a token identifier is an array of bytes. Takamaka uses, more generically, a `BigInteger`. Note that a `BigInteger` can be constructed from an array of bytes by using the constructor of class `BigInteger` that receives an array of bytes. In the ERC721 specification, token owners are contracts, although the implementation will check that only contracts implementing the `IERC721Receiver` interface are added to an `IERC721` ledger, or externally owned accounts.

> The reason for allowing externally owned accounts is probably a simplification, since Ethereum users own externally owned accounts and it is simpler for them to use such accounts directly inside an ERC721 ledger, instead of creating contracts of type `IERC721Receiver`. In any case, no other kind of contracts is allowed in ERC721 implementations.

The hierarchy of the Takamaka classes for the ERC721 standard is shown in Figure 34.

As in the case of the ERC20 tokens, the interface `IERC721View` contains the read-only operations that implement a ledger of non-fungible tokens: the `ownerOf` method yields the owner of a given token and the `balanceOf` method returns the number of tokens held by a given `account`. The `snapshot()` method allows one to create a frozen read-only view of a ledger.

The `IERC721` subinterface adds methods for token transfers. Please refer to their description given by the Ethereum standard. We just say here that the `transferFrom` method moves a given token from its previous owner to a new owner. The caller of this method can be the owner of the token, but it can also be another contract, called *operator*, as long as the latter has been previously approved by the token owner, by using the `approve` method. It is also possible to approve an operator for all one's tokens (or remove such approval), through the `setApprovalForAll` method. The `getApproved` method tells who is the operator approved for a given token (if any) and the `isApprovedForAll` method tells if a given operator has been approved to transfer all tokens of a given `owner`. The `view` method yields a read-only view of the ledger, that reflects all future changes to the ledger.

The implementation `ERC721` provides standard implementations for all methods of `IERC721View` and `IERC721`, adding metadata information about the name and the symbol of the token and protected methods for minting and burning new tokens. These are meant to be called in subclasses, such as `ERC721Burnable`. Namely, the latter adds a `burn` method that allows the owner of a token (or its approved operator) to burn the token.

As we have already said previously, the owners of the tokens are declared as contracts in the `IERC721View` and `IERC721` interfaces, but the `ERC721` implementation actually requires them to be `IERC71Receiver`s or externally owned accounts. Otherwise, the methods of `ERC721` will throw an exception. Moreover, token owners that implement the `IERC721Receiver` interface get their `onReceive` method called whenever new tokens are transferred to them.

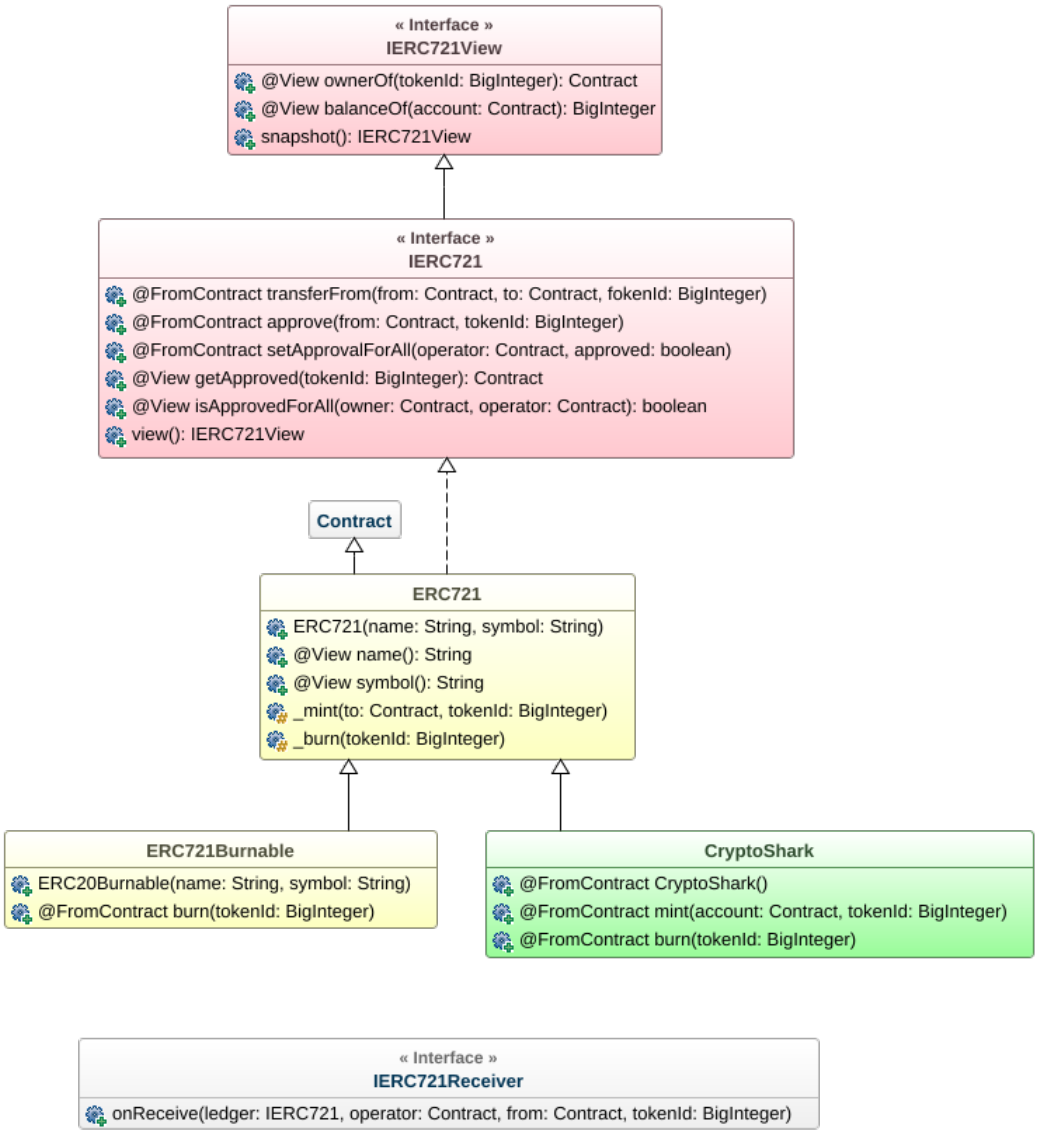



*Figure 34. The hierarchy of the ERC721 token implementations.*

The ERC721 standard requires `onReceive` to return a special message, in order to prove that the contract actually executed that method. This is a very technical necessity of Solidity, whose first versions allowed one to call non-existent methods without getting an error. It is a sort of security measure, since Solidity has no `instanceof` operator and cannot check in any reliable way that the token owners are actually instances of the interface `IERC721Receiver`. The implementation in Solidity uses the ERC165 standard for interface detection, but that standard is not a reliable replacement of `instanceof`, since a contract can always pretend to belong to any contract type. Takamaka is Java and can use the `instanceof` operator, that works correctly. As a consequence, the `onReceive` method in Takamaka needn't return any value.

## Implementing Our Own ERC721 Token

**[See project `erc721` inside the `hotmoka_tutorial` repository]**

Let us define a ledger for non-fungible tokens that only allows its creator the mint or burn tokens. We will call it `CryptoShark`. As Figure 34 shows, we plug it below the `ERC721` implementation, so that we inherit that implementation and do not need to reimplement the methods of the `ERC721` interface. The code is almost identical to that for the `CryptoBuddy` token defined in [Implementing Our Own ERC20 Token].

Create in Eclipse a new Maven Java 11 (or later) project named `erc721`. You can do this by duplicating the project `erc20` (make sure to store the project inside the `hotmoka_tutorial` directory, as a sibling of `family`, `ponzi`, `tictactoe` and so on). Use the following `pom.xml`:

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
                        http://maven.apache.org/xsd/maven-4.0.0.xsd">

  <modelVersion>4.0.0</modelVersion>
  <groupId>io.hotmoka</groupId>
  <artifactId>erc721</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>1.0.7</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>
```

and the following `module-info.java`:

```
module erc721 {
  requires io.takamaka.code;
}
```

Create package `io.takamaka.erc721` inside `src/main/java` and add the following `CryptoShark.java` inside that package:

```
package io.takamaka.erc721;

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.tokens.ERC721;

public class CryptoShark extends ERC721 {
  private final Contract owner;

  public @FromContract CryptoShark() {
    super("CryptoShark", "SHK");
    owner = caller();
  }

  public @FromContract void mint(Contract account, BigInteger tokenId) {
    require(caller() == owner, "Lack of permission");
    _mint(account, tokenId);
  }

  public @FromContract void burn(BigInteger tokenId) {
    require(caller() == owner, "Lack of permission");
    _burn(tokenId);
  }
}
```

The constructor of `CryptoShark` takes note of the creator of the token. That creator is the only that is allowed to mint or burn tokens, as you can see in methods `mint` and `burn`.

You can generate the `erc721-0.0.1.jar` file:

```
$ cd erc721
$ mvn package
```

Then you can install that jar in the node and create an instance of the token exactly as we did for the `CryptoBuddy` ERC20 token before.

Chapter 8

# Code Verification

Code verification checks that code complies with some constraints, that should guarantee that its execution does not run into errors. Modern programming languages apply more or less extensive code verification, since this helps programmers write reliable code. This can both occur at run time and at compile time. Run-time (*dynamic*) code verification is typically stronger, since it can exploit exact information on run-time values flowing through the code. However, compile-time (*static*) code verification has the advantage that it runs only once, at compilation time or at jar installation time, and can prove, once and for all, that some errors will never occur, regardless of the execution path that will be followed at run time.

Hotmoka nodes apply a combination of static and dynamic verification to the Takamaka code that is installed inside their store. Static verification runs only once, when a node installs a jar in its store, or when classes are loaded for the first time at run time. Dynamic verification runs every time some piece of code gets executed.

## JVM Bytecode Verification

Takamaka code is written in Java, compiled into Java bytecode, instrumented and run inside the Java Virtual Machine (*JVM*). Hence, all code verifications executed by the JVM apply to Takamaka code as well. In particular, the JVM verifies some structural and dynamic constraints of class files, including their type correctness. Moreover, the JVM executes run-time checks as well: for instance, class casts are checked at run time, as well as pointer dereferences and array stores. Violations result in exceptions. For a thorough discussion, we refer the interested reader to the official documentation about Java bytecode class verification [JVM-Verification].

## Takamaka Bytecode Verification

Hotmoka nodes verify extra constraints, that are not checked as part of the standard JVM bytecode verification. Such extra constraints are mainly related to the correct use of Takamaka annotations and contracts, and are in part static and in part dynamic. Static constraints are checked when a jar is installed into the store of a node, hence only once for each node of a network. If a static constraint is violated, the transaction that tries to install the jar fails with

an exception. Dynamic constraints, instead, are checked every time a piece of code is run. If a dynamic constraint is violated, the transaction that runs the code fails with an exception.

Below, remember that `@FromContract` is shorthand for `@FromContract(Contract.class)`. Moreover, note that the constraints related to overridden methods follow by Liskov's principle [LiskovW94].

Hotmoka nodes verify the following static constraints:

1. The `@FromContract(C.class)` annotation is only applied to constructors of a (non-strict) subclass of `io.takamaka.code.lang.Storage` or to instance methods of a (non-strict) subclass of `io.takamaka.code.lang.Storage` or interface.
2. In every use of the `@FromContract(C.class)` annotation, class `C` is a subclass of the abstract class `io.takamaka.code.lang.Contract`.
3. If a method is annotated as `@FromContract(C.class)` and overrides another method, then the latter is annotated as `@FromContract(D.class)` as well, and `D` is a (non-strict) subclass of `C`.
4. If a method is annotated as `@FromContract(D.class)` and is overridden by another method, then the latter is annotated as `@FromContract(C.class)` as well, and `D` is a (non-strict) subclass of `C`.
5. If a method is annotated as `@Payable` or `@RedPayable`, then it is also annotated as `@FromContract(C.class)` for some `C`.
6. If a method is annotated as `@Payable` or `@RedPayable`, then it has a first formal argument (the paid amount) of type `int`, `long` or `BigInteger`.
7. If a method is annotated as `@Payable` and overrides another method, then the latter is annotated as `@Payable` as well; an identical rule holds for `@RedPayable`.
8. If a method is annotated as `@Payable` and is overridden by another method, then the latter is annotated as `@Payable` as well; an identical rule holds for `@RedPayable`.
9. No method or constructor is annotated with both `@Payable` and `@RedPayable`.
10. The `@Payable` annotation is only applied to constructors of a (non-strict) subclass of `io.takamaka.code.lang.Contract` or to instance methods of a (non-strict) subclass of `io.takamaka.code.lang.Contract` or interface.
11. The `@RedPayable` annotation is only applied to constructors of a (non-strict) subclass of `io.takamaka.code.lang.Contract` or to instance methods of a (non-strict) subclass of `io.takamaka.code.lang.Contract` or interface.
12. Classes that extend `io.takamaka.code.lang.Storage` have instance non-transient fields whose type is primitive (`char`, `byte`, `short`, `int`, `long`, `float`, `double` or `boolean`), or is a class that extends `io.takamaka.code.lang.Storage`, or is an `enum` without instance non-transient fields, or is any of `java.math.BigInteger`, `java.lang.String`, `java.lang.Object` or an interface (see Storage Types and Constraints on Storage Classes).

The choice of allowing, inside a storage type, fields of type `java.lang.Object` can be surprising. After all, any reference value can be stored in such a field, which requires to verify, at run time, if the field actually contains a storage value or not (see the dynamic checks, below). The reason for this choice is to allow generic storage types, such as `StorageTreeMap<K,V>`, whose values are storage values as long as `K` and `V` are replaced with storage types. Since Java implements generics by erasure, the bytecode of such a class ends up having fields of type `java.lang.Object`. An alternative solution would be to bound `K` and `V` from above (`StorageTreeMap<K extends Storage, V extends Storage>`). This second choice will be erased by using `Storage` as static type of the erased fields of the class. However, not all storage reference values extend `Storage`. For instance, this solution would not allow one to write `StorageTreeMap<MyEnum, BigInteger>`, where `MyEnum` is an enumeration type with no instance non-transient fields: both `MyEnum` and `BigInteger` are storage types, but neither extends `Storage`. The fact that fields of type `java.lang.Object` or interface actually hold a storage value at the end of a transaction is checked dynamically (see the dynamic checks below).

13. There are no static initializer methods.

Java runs static initializer methods the first time their defining class is loaded. They are either coded explicitly, inside a `static { ... }` block, or are implicitly generated by the compiler in order to initialize the static fields of the class. The reason for forbidding such static initializers is that, inside Takamaka, they would end up being run many times, at each transaction that uses the class, and reset the static state of a class, since static fields are not kept in blockchain. This is a significant divergence from the expected semantics of Java, that requires static initialization of a class to occur only once during the lifetime of that class. Note that the absence of static initializers still allows a class to have static fields, as long as they are bound to constant primitive or `String` values.

14. There are no finalizers.

A finalizer is a method declared exactly as `public void finalize() { ... }`. It might be called when the JVM garbage collects an object from RAM. The reason for forbidding such finalizers is that their execution is not guaranteed (they might never be called) or might occur at a non-deterministic moment, while code in blockchain must be deterministic.

15. Calls to `caller()` occur only inside `@FromContract` constructors or methods and on `this`.
16. Calls to constructors or methods annotated as `@FromContract` occur only in constructors or instance methods of an `io.takamaka.code.lang.Contract`; moreover, if they occur, syntactically, on `this`, then they occur in a method or constructor that is itself annotated as `@FromContract` (since the `caller()` is preserved in that case).
17. Bytecodes `jsr`, `ret` and `putstatic` are not used; inside constructors and instance methods, bytecodes `astore 0`, `istore 0`, `lstore 0`, `dstore 0` and `fstore 0` are not used.

Local variable 0 is used to hold the `this` reference. Forbidding its modification is important to guarantee that `this` is not reassigned in code, which is impossible in Java but perfectly legal in (unexpected) Java bytecode. The guarantee that `this` is not reassigned is needed, in turn, for checking properties such as point 15 above.

18. There are no exception handlers that may catch unchecked exceptions (that is, instances of `java.lang.RuntimeException` or of `java.lang.Error`).

By forbidding exception handlers for unchecked exceptions, it follows that unchecked exceptions will always make a transaction fail: all object updates up to the exception will be discarded. In practice, transactions failed because of an unchecked exception leave no trace on the store of the node, but for the gas of the caller being consumed. The reason for forbidding exception handlers for unchecked exceptions is that they could occur in unexpected places and leave a contract in an inconsistent state. Consider for instance the following (illegal) code:

```
try {
  this.list.add(x);
  x.flagAsInList();
  this.counter++;
}
catch (Exception e) { // illegal in Takamaka
}
```

Here, the programmer might expect that the size of `this.list` is `this.counter` and the correctness of his code might be based on that invariant. However, if `x` holds `null`, an unchecked `NullPointerException` is raised just before `this.counter` could be incremented, it would be caught and ingnored. The expected invariant would be lost. The contract will remain in blockchain in an inconsistent state, for ever. The situation would be worse if an `OutOfGasError` would be caught: the caller might provide exactly the amount of gas needed to reach the `flagAsInList()` call, and leave the contract in an inconsistent state. Checked exceptions, instead, are explicitly checked by the compiler, which should ring a bell in the head of the programmer.
For a more dangerous example, consider the following Java bytecode:

```
10: goto 10
exception handler for java.lang.Exception: 10 11 10 // illegal in Takamaka
```

This Java bytecode exception handler entails that any `OutOfGasError` thrown by an instruction from line 10 (included) to line 11 (excluded) redirects control to line 10. Hence, this code will exhaust the gas by looping at line 10. Once all gas is consumed, an `OutOfGasError` is thrown, that is redirected to line 10. Hence another `OutOfGasError` will occur, that redirects the executor to line 10, again. And so on, for ever. That is, this code disables the guarantee that Takamaka transactions always terminate, possibly with an `OutOfGasError`. This code could be used for a DOS attack to a Hotmoka node. Although this code cannot be written in Java, it is well possible to write it directly, with a bytecode editor, and submit it to a Hotmoka node, that will reject it, thanks to point 19.

19. If a method or constructor is annotated as `@ThrowsException`, then it is public.
20. If a method is annotated as `@ThrowsException` and overrides another method, then the

latter is annotated as `@ThrowsException` as well.

21. If a method is annotated as `@ThrowsException` and is overridden by another method, then the latter is annotated as `@ThrowsException` as well.
22. Classes installed in a node are not in packages `java.*`, `javax.*` or `io.takamaka.code.*`; packages starting with `io.takamaka.code.*` are however allowed if the node is not initialized yet.

> The goal of the previous constraint is to make it impossible to change the semantics of the Java or Takamaka runtime. For instance, it is not possible to replace class `io.takamaka.code.lang.Contract`, which could thoroughly revolutionize the execution of the contracts. During the initialization of a node, that occurs once at its start-up, it is however permitted to install the runtime of Takamaka (the `io-takamaka-code-1.0.7.jar` archive used in the examples in the previous chapters).

23. All referenced classes, constructors, methods and fields must be white-listed. Those from classes installed in the store of the node are always white-listed by default. Other classes loaded from the Java class path must have been explicitly marked as white-listed in the `io-takamaka-code-whitelisting-1.0.7.jar` archive.

> Hence, for instance, the classes of the support library `io.takamaka.code.lang.Storage` and `io.takamaka.code.lang.Takamaka` are white-listed, since they are inside `io-takamaka-code-1.0.7.jar`, that is typically installed in the store of a node during its initialization. Classes from user jars installed in the node are similarly white-listed. Method `java.lang.System.currentTimeMillis()` is not white-listed, since it is loaded from the Java class path and is not annotated as white-listed in `io-takamaka-code-whitelisting-1.0.7.jar`.

24. Bootstrap methods for the `invokedynamic` bytecode use only standard call-site resolvers, namely, instances of `java.lang.invoke.LambdaMetafactory.metafactory` or of `java.lang.invoke.StringConcatFactory.makeConcatWithConstants`.

> This condition is needed since other call-site resolvers could call any method, depending on their algorithmic implementation, actually side-stepping the white-listing constraints imposed by point 24. Java compilers currently do not generate other call-site resolvers.

25. There are no native methods.
26. There are no `synchronized` methods, nor `synchronized` blocks.

> Takamaka code is single-threaded, to enforce its determinism. Hence, there is no need to use the `synchronized` keyword.

27. Field and method names do not start with a special prefix used for instrumentation, namely they do not start with `§`.

This condition avoids name clashes after instrumentation. That prefix is not legal in Java, hence this constraint does not interfere with programmers. However, it could be used in (unexpected) Java bytecode, that would be rejected thanks to point 27.

28. Packages are not split across different jars in the classpath.

This condition makes it impossible to call `protected` methods outside of subclasses and of the same jar where they are defined. Split packages allow an attacker to define a new jar with the same package name as classes in another jar and call the `protected` methods of objects of those classes. This is dangerous since `protected` methods often access or modify sensitive fields of the objects.

Takamaka verifies the following dynamic constraints:

1. Every `@Payable` or `@RedPayable` constructor or method is passed a non-`null` and non-negative amount of funds.
2. A call to a `@Payable` or `@RedPayable` constructor or method succeeds only if the caller has enough funds to pay for the call (ie., the amount first parameter of the method or constructor).
3. A call to a `@FromContract(C.class)` constructor or method succeeds only if the caller is an instance of `C`.
4. A bytecode instruction is executed only if there is enough gas for its execution.
5. White-listed methods or constructors with white-listing proof obligations are only executed if such proof obligations are satisfied.
6. Non-transient fields of type `java.lang.Object` or of type interface, belonging to some storage object reachable from the actual parameters of a transaction at the end of the transaction, contain `null` or a storage object.

## Command-Line Verification and Instrumentation

**[See project `family_wrong` inside the `hotmoka_tutorial` repository]**

If a jar being installed in a Hotmoka node does not satisfy the static constraints that we have described before, the installation transaction fails with a verification exception, no jar is actually installed but the gas of the caller gets consumed. Hence it is not practical to realize that a static constraint does not hold only by trying to install a jar in a node. Instead, it is desirable to verify all constraints off-line, fix all violations (if any) and only then install the jar in the node. This is possible by using the `moka` command-line interface of Hotmoka. Namely, it provides a subcommand that performs the same identical jar verification that would be executed when a jar is installed in a Hotmoka node.

Create a `family_wrong-0.0.1.jar` containing a wrong version of the `family` project. For that, copy the `family` project into `family_wrong`, change the artifact name in its `pom.xml` into `family_wrong` and modify its `Person` class so that it contains a few errors, as follows:

```
package io.takamaka.family;

import io.takamaka.code.lang.Exported;
import io.takamaka.code.lang.Payable;
```

```
import io.takamaka.code.lang.Storage;

@Exported
public class Person extends Storage {
  private final String name;
  private final int day;
  private final int month;
  private final int year;

  // error: arrays are not allowed in storage
  public final Person[] parents = new Person[2];

  public static int toStringCounter;

  public Person(String name, int day, int month, int year,
                Person parent1, Person parent2) {

    this.name = name;
    this.day = day;
    this.month = month;
    this.year = year;
    this.parents[0] = parent1;
    this.parents[1] = parent2;
  }

  // error: @Payable without @FromContract, missing amount and is not in Contract
  public @Payable Person(String name, int day, int month, int year) {
    this(name, day, month, year, null, null);
  }

  @Override
  public String toString() {
    toStringCounter++; // error (line 37): static update (putstatic) is now allowed
    return name +" (" + day + "/" + month + "/" + year + ")";
  }
}
```

Then generate the `family_wrong-0.0.1.jar` file:

```
$ cd family_wrong
$ mvn package
```

Let us start with the verification of `io-takamaka-code-1.0.7.jar`, taken from Maven's cache:

```
$ cd hotmoka_tutorial
$ moka verify
    ~/.m2/repository/io/hotmoka/io-takamaka-code/1.0.7/io-takamaka-code-1.0.7.jar
    --init
Verification succeeded
```

No error has been issued, since the code does not violate any static constraint. Note that we used

the `--init` switch, since otherwise we would get many errors related to the use of the forbidded `io.takamaka.code.*` package. With that switch, we verify the jar as it would be verified before node initialization, that is, by considering such packages as legal.

We can generate the instrumented jar, exactly as it would be generated during installation in a Hotmoka node. For that, we run:

```
$ mkdir instrumented
$ moka instrument
    ~/.m2/repository/io/hotmoka/io-takamaka-code/1.0.7/io-takamaka-code-1.0.7.jar
    instrumented/io-takamaka-code-1.0.7.jar
    --init
```

The `moka instrument` command verifies and instruments the jar, and then stores its instrumented version inside the `instrumented` directory.

Let us verify and instrument `family-0.0.1.jar` now. As all Takamaka programs, it uses classes from the `io-takamaka-code` jar, hence it depends on it. We specify this with the `--libs` option, that must refer to an already instrumented jar:

```
$ moka instrument
    family/target/family-0.0.1.jar
    instrumented/family-0.0.1.jar
    --libs instrumented/io-takamaka-code-1.0.7.jar
```

Verification succeeds this time as well, and an instrumented `family-0.0.1.jar` appears in the `instrumented` directory. Note that we have not used the `--init` switch this time, since we wanted to simulate the verification as it would occur after the node has been already initialized, when users add their jars to the store of the node.

Let us verify the `family_wrong-0.0.1.jar` archive now, that (we know) contains a few errors. This time, verification will fail and the errors will be printed on the screen:

```
$ moka verify
    family_wrong/target/family_wrong-0.0.1.jar
    --libs instrumented/io-takamaka-code-1.0.7.jar

io/takamaka/family/Person.java field parents:
  type not allowed for a field of a storage class
io/takamaka/family/Person.java method <init>:
  @Payable can only be used in contracts or interfaces
io/takamaka/family/Person.java method <init>:
  a @Payable method must have a first argument for the paid amount,
  of type int, long or BigInteger
io/takamaka/family/Person.java method <init>:
  @Payable can only be applied to a @FromContract method or constructor
io/takamaka/family/Person.java:55:
  static fields cannot be updated

Verification failed because of errors
```

The same failure occurs with the `instrument` command, that will not generate the instrumented jar:

```
$ moka instrument
    family_wrong/target/family_wrong-0.0.1.jar
    instrumented/family_wrong-0.0.1.jar
    --libs instrumented/io-takamaka-code-1.0.7.jar

io/takamaka/family/Person.java field parents:
  type not allowed for a field of a storage class
io/takamaka/family/Person.java method <init>:
  @Payable can only be used in contracts or interfaces
io/takamaka/family/Person.java method <init>:
  a @Payable method must have a first argument for the paid amount,
  of type int, long or BigInteger
io/takamaka/family/Person.java method <init>:
  @Payable can only be applied to a @FromContract method or constructor
io/takamaka/family/Person.java:55:
  static fields cannot be updated

Verification failed because of errors, no instrumented jar was generated
```

Chapter 9

# References


[Antonopoulos17] Antonopoulos, A. M. (2017). Mastering Bitcoin: Programming the Open Blockchain. O'Reilly Media, 2nd edition.

[AntonopoulosW19] Antonopoulos, A. M. and Wood, G. (2019). Mastering Ethereum: Building Smart Contracts and DApps. O'Reilly Media.

[AtzeiBC17] Atzei, N., Bartoletti, M. and Cimoli, T. (2017). A Survey of Attacks on Ethereum Smart Contracts. *6th Internal Conference on Principles of Security and Trust (POST17)* ETAPS 2017.

[BeniniGMS21] Benini, A., Gambini, M., Migliorini, S., Spoto, F. (2021). Power and Pitfalls of Generic Smart Contracts. *3rd International Conference on Blockchain Computing and Applications (BCCA2021)*.

[BlindAuction] https://solidity.readthedocs.io/en/v0.5.9/solidity-by-example.html#id2.

[CrafaPZ19] Crafa, S., Di Pirro, M. and Zucca, E. (2019). Is Solidity Solid Enough? *3rd Workshop on Trusted Smart Contracts (WTSC19)*.

[CrosaraOST21] Crosara, M., Olivieri, L., Spoto, F. and Tagliaferro, F. (2021). An Implementation in Java of ERC-20 with Efficient Snapshots. *3rd International Conference on Blockchain Computing and Applications (BCCA2021)*.

[EC2] Amazon EC2: Secure and Resizable Compute Capacity in the Cloud. https://aws.amazon.com/ec2.

[Freni20] Freni, P., Ferro, E. and Moncada, R. (2020). Tokenization and Blockchain Tokens Classification: A Morphological Framework. *IEEE Symposium on Computers and Communications (ISCC)*, Rennes, France, pages 1-6.

[IyerD08] Iyer, K. and Dannen, C. (2018). Building Games with Ethereum Smart Contracts: Intermediate Projects for Solidity Developers. Apress.

[JVM-Verification] https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-4.html#jvms-4.9.

[LiskovW94] Liskov, B. and Wing, J. M. (1994). A Behavioral Notion of Subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811-1841.

[MakB17] Mak, S. and Bakker, P. (2017). Java 9 Modularity: Patterns and Practices for Developing Maintainable Applications. Oreilly & Associates Inc.

[Nakamoto08] Nakamoto, S. (2008). Bitcoin: A Peer-to-Peer Electronic Cash System. Available at https://bitcoin.org/bitcoin.pdf.

[OliveiraZBS18] Oliveira, L., Zavolokina, L., Bauer, I. and Schwabe, G. (2018). To Token or not to Token: Tools for Understanding Blockchain Tokens. *Proceedings of the International Conference on Information Systems - Bridging the Internet of People, Data, and Things, ICIS 2018*, San Francisco, CA, USA, Association for Information Systems.

[OlivieriST21] Olivieri, L., Spoto, F., Tagliaferro, F. (2021). On-Chain Smart Contract Verification over Tendermint. *5th Workshop on Trusted Smart Contracts (WTSC21)*.

[Sentry] Sentry Node Architecture Overview - Cosmos Forum. https://forum.cosmos.network/t/sentry-node-architecture-overview/454.

[Spoto19] Spoto, F. (2019). A Java Framework for Smart Contracts. *3rd Workshop on Trusted Smart Contracts (WTSC19)*.

[Spoto20] Spoto, F. (2020). Enforcing Determinism of Java Smart Contracts. *4th Workshop on Trusted Smart Contracts (WTSC20)*.

[Tapscott20] Tapscott, D. (2020). Token Taxonomy: The Need for Open-Source Standards around Digital Assets. https://www.blockchainresearchinstitute.org/project/token-taxonomy-the-need-for-open-source-standards-around-digital-assets.

[Tendermint] https://tendermint.com.

### About the author

Fausto Spoto holds a PhD in Computer Science from the University of Pisa, Italy. Currently, he works as an Associate Professor at the University of Verona, Italy, where he teaches courses on programming languages and software development. His research interests are related to the development of high quality software and automatic code verification. His interest for blockchain stems from the fact that the latter is a natural context where software must be correct and reliable.

### About Hotmoka

Hotmoka is a research project of the University of Verona, Italy. Its implementation and documentation is open-source and non-proprietary. You are welcome to check out the code and modify it. The home page of the project is at https://www.hotmoka.io. The project sources and this same tutorial are maintained at https://github.com/Hotmoka. For questions about the project, you can write to `info@hotmoka.io`.