Techniques of Java Programming: Java Basics

Manuel Oriol

April 18, 2006

1 Introduction

Java is the Object-Oriented language of choice for most developers. It is a type-safe, Object-Oriented, multiplatform programming lanuguage Its main advertisement has been for years: "compile once, run everywhere". This is true for programs that are meant to run on multiple platforms. Nevertheless, multiple versions of the Java Virtual Machine (JVM) and of the Standard Development Kit (SDK) do not guarantee that anymore. This is the price to pay for wide acceptance.

In this course, we intend to give an overview of the language to computer scientists that are already proficient in at least one object-oriented language. Thus we will go through the language and its general mechanisms at a fast pace in the first few lessons and then we will present more advanced features.

Readers not only get a basic understanding, that allows them to build Java programs, but also get a deeper understanding of the language internal mechanisms. These mechanisms will be illustrated through experiments that flirt with research. This tutorial explains the language developed in Java 2 version 5.0 as it is the current release. If we consider any other specific version at any point, it will be clearly stated in the text.

2 Java Development

2.1 Big Picture

Developing in Java implies understanding a set of basic principles. Figure 1 shows the main steps and tools that are commonly used during Java development and compilation.

The source code of Java programs is composed of files that have the `.java` extension. Java being class-based, each file contains at least one class that has the same name as the file (without the extension).

The Application Programming Interface (API) documentation can be extracted automatically from the Java source code by using the `javadoc` command. The generated documentation consists of a website in HTML.

The compilation of Java source code, using the `javac` command, generates one file per class of the source code. Each of these files begins by the name of the file where the class was defined and has the extension `.class`. The content of such *class files* is byte code: an intermediate representation that resembles assembly code and is understood by any Java execution platform.

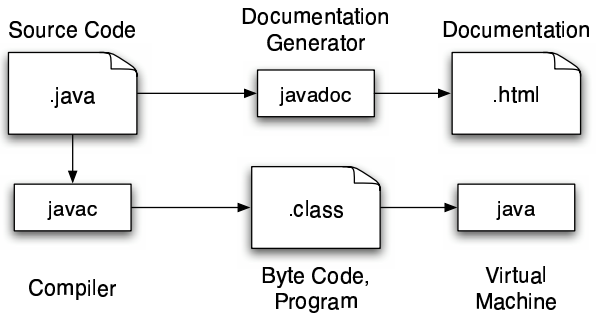

Figure 1: The Java infrastructure.

Such execution platforms are called Java Virtual Machines (JVM) and are most often called `java`.

2.2 Java Environment Variables

As with any environment that contains programs, these programs must be included in the `PATH` variable. In Java, the structure of the Java installation contains a `bin` directory that contains all the programs.

Another environment variable of importance in Java is the `CLASSPATH` variable that stores places from which classes can be retrieved. Schematically, if one wants to use a class that is located in a directory, the path to that directory must be in the `CLASSPATH` to compile or to execute the program. Command-line options can also participate to the actual value of the `CLASSPATH` when needed to have a finer-grained control on the program as described in the next subsections.

2.3 Java Compiler

Table 1: Java Compiler main options

javac	
`-g`	Generate all debugging information (to be used with a debugger)
`-nowarn`	No warnings
`-verbose`	Output messages while compiling to show the effected operations
`-classpath` *path*, `-cp` *path*	Adds the *path* to the `CLASSPATH` value for this execution
`-d` *path*	Generated classes will be put to *path*
`-version`	Gives the current version
`-help`	Print help
`-X` e.g. `-Xstdout` *file*	Print non-standard options e.g. redirects standard out to a *file*.

The Java compiler takes a set of classes as arguments and generates the byte code. As shown in Table 1 the classpath can be extended by using the option `-cp`, warnings can be ignored... and other options can be set.

Each class has its own class file. In the case a source code file `C.java` containing more than one class, the main class (that has the same name as the source file without extension) is compiled into a file `C.class`. Other named classes are compile into classes that have the name `C$`*NameOfTheClass*`.class`. Anonymous classes are compiled into classes that have the name `C$`*number*`.class` where *number* is a unique identifier for anonymous classes within the file.

Example of Java compilation:

```
javac *.java
```

2.4 Java Virtual Machine

Table 2: Java Virtual Machine main options

java	
`-classpath` *path*, `-cp` *path*	Adds the *path* to the `CLASSPATH` value for this execution
`-verbose` [:class—gc—jni]	Output messages on classes/garbage collection/java native interface to show the effected operations
`-version`	Gives the current version and exits
`-help`, `-?`	Print help
`-X` e.g. `-Xms`*size* e.g. `-Xss`*size*	Print non-standard options e.g. Set the initial heap size. e.g. Set the thread stack size.

The Java Virtual Machines (JVM) are the execution environments for Java programs (see Table 2). A JVM is intended to act as an interpreter for the byte code. Thus the byte code is itself portable from one platform to another. Java virtual machines have to be ported and installed on the respective platforms prior to program execution.

By default, the argument class name will be loaded and the `main` method within this class will be called as a starting point. The JVM takes care of memory allocation and deallocation. In particular, the JVM manages automatic garbage collection of instances and code when not used any more .

While interpretation is slow in general, a JVM has lots of facilities to speed up execution. The most important one is Just-In-Time (JIT) compiling which compiles the byte-code to native code when it is loaded, thus giving near native performance on the use of a class once it has been executed at least once. Of course this initial translation yields a performance penalty at first execution of given method.

If `MyMain` is the class containing the bootstrap code, the program can be launched as follows:

```
java MyMain
```

2.5 Java Dynamic Loading

As each class has its own file, class loading can be delayed until the class is actually needed. The default behavior for the Java class loading mechanism is on-demand class loading. A class is actually loaded at the moment its first instance is created. This is often called *lazy class loading*.

This mechanism has the clear advantage of reducing startup time, especially in the case of just-in-time compilation but slows down the execution while a complete set of classes has not been loaded in memory. Please note that this constitutes only a mere introduction to JVM class loading. We will study it in more details later on.

2.6 Java Documentation

Table 3: Java documentation generation tool, main options

javadoc	
`-classpath` *path*, `-cp` *path*	Adds the *path* to the `CLASSPATH` value for this execution
`-verbose`	Output messages on what javadoc is doing
`-quiet`	Do not output messages on classes/garbage
`-version`	Gives the current version and exits
`-help`	Print help
`-d` *path*	Generated documentation will be put to *path*

Java Development Kits (JDK) include the `javadoc` utility (see Table 3 for details on the options) that automatically generates the documentation in HTML format.

In Java, comments can be inserted in the code in two ways: using `//` to comment the following characters in the line or use `/*` to begin a comment and `*/` to finish it.

Each comment written using the `/**` and ending by `*/` just immediately before an element that can be commented ensures that the comment will be put into the documentation. Asterisks at the beginning of a line are ignored.

It is also possible to include tags that lead to the creation of specific categories and links automatically. Including HTML tags is also allowed and will be included as is in generated pages.

As an example, the `javadoc` documentation presents the example from Figure 2 that describes the package `tojp`:

```
/**
 * This package is an example for the course Technique of Java Programming at ETHZ
 *
 * @since 0.1
 * @see java.lang
 * @author Manuel Oriol
 */
package ch.ethz.tojp;
```

Tags usable in the Java documentation are of two kinds: tags that can be used within the text of the documentation itself and tags that define categories and need to be used at the end of the descriptions. In Table 4, we show the

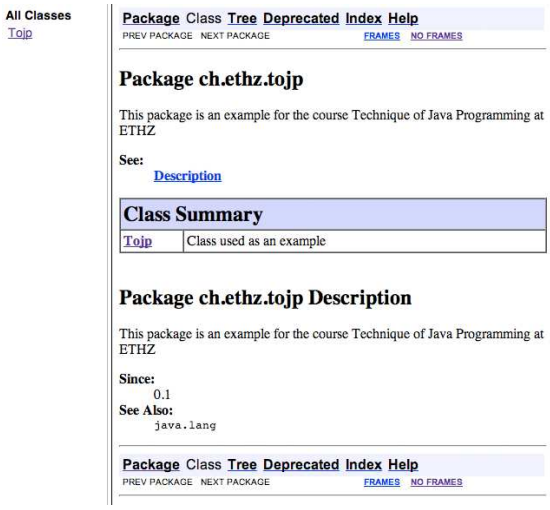

Figure 2: Generated documentation

first category of tags between braces ({ }) and explain all tags with an informal description. A complete documentation for javadoc and tags can be found in the JDK documentation.[1]

Table 4: JavaDoc Tags

Tag	JDK	Use
@author	1.0	Add an author entry
{@code}	1.5	Display text in courier font
{@docRoot}	1.3	Refers to the root of the documentation
@deprecated	1.0	Adds a comment to quit using the class
@exception	1.0	Synonymous for @throws
{@inheritDoc}	1.4	Copies "nearest" documentation
{@link}	1.2	Link to another documentation
{@linkplain}	1.4	As @link except that the link is plain
{@literal}	1.5	Display the text as is
@param	1.0	Adds a parameter description
@return	1.0	Adds a description of returned value
@see	1.0	Adds a "See Also" heading
@serialData	1.2	Description of serialized data
@serialField	1.2	Description of serialized fields
@since	1.1	Indication on since when this was used
@throws	1.2	Gives a description of the thrown exceptions
{@value}	1.4	Displays the actual value of a constant
@version	1.0	adds a version field

An example of the `javadoc` use can be:

```
javadoc *.java -d doc/
```

2.7 Java Archive - JAR

Grouping classes into an archive is a nice way to distribute applications written in Java. The Virtual Machines are all able to load the `*.jar` archives files. It

[1] http://java.sun.com/j2se/1.5.0/docs/tooldocs/solaris/javadoc.html#javadoctags

Table 5: Java Archive main options

jar	
`c`	Create
`t`	List content
`x`	Extract
`u`	Update
`v`	Verbose mode
`f`	First argument is archive file name
`0`	No ZIP compression

has a clear advantage in terms of size as the classes are compressed using the ZIP algorithm. It is also possible to specify a startup class and thus have the jar file being an executable program.

Its use (see Table 5 for details on the options, for a good tutorial, please visit SUN's website[2]) is similar to the `tar` utility:

```
jar cvf classes.jar Foo.class Bar.class
```

2.8 Java Debugger

The basic Java debugger is very similar to gdb and the commands it understands.[3] The command to launch is `jdb` *StartClass* and it launches a second JVM to debug the program that waits for continued execution. Breakpoints can be set by using `stop at` *AClass*`:`*aLineNumber* or `stop in` *aCompleteMethodName*, or `stop in` *AClass*`.<init>` for stopping at the constructors, or `stop in` *AClass*`.<clinit>` to stop at the initialization of the class.

`next` advances to the next line.
`step` advances to the next line within current scope.
`run` starts execution.
`cont` continues execution after a breakpoint has been found.
`print` prints values of the subsequent fields/method calls.
`dump` is more complete with all values of fields dumped.
`threads` lists the number of threads.
`thread` selects the thread with number indicated afterwards.
`where` dumps the stack.
`where all` dumps all the stacks.
`catch` can catch an exception on-the-fly.

2.9 Exercises

1. Compile `HelloWorld.java`.
2. Generate complete documentation for `HelloWorld.java`.
3. Unzip a jar file using unzip.

[2] http://java.sun.com/docs/books/tutorial/deployment/jar/
[3] http://java.sun.com/j2se/1.3/docs/tooldocs/solaris/jdb.html for more details

4. Debug HelloWorld using gdb.

3 Packages

3.1 Naming Convention

By using packages you can set unique names for classes. To assign a file to a package, it is needed to put at the beginning of the file package adherance:

`package` *package.name*`;`

Package names are sequences of characters separated by dots. The hierarchy that is intended goes from the more general to the more particular. In our previous example, we defined the package `ch.ethz.tojp` it means that the convention is to prefix all packages from ETHZ by `ch.ethz`. In the file system this is translated by having a tree-like structure where each folder includes the rest of the package.

Classes from the package ch.ethz.tojp would all be contained in a directory named `tojp`, itself contained in a directory named `ethz`, itself contained in a directory named `ch`. By convention, package names are in lower case, while class names begin with an upper case character, only contain letters and have upper cases at the beginning of each new word in the class name.

3.2 Referencing Classes and Packages

In order for the compiler and the JVM to be able to find the class the directory containing the first directory of the package (`ch` in our case) must be in the CLASSPATH.

Referencing a class in the code must be done using its *fully qualified name*. This name is built by concatenating the package name a dot and the short name of the class. In that case, the class Tojp from ch.ethz.tojp would have the fully qualified name `ch.ethz.tojp.Tojp`. As this is very much time and space consuming, it is also possible to `import` packages and classes at the beginning of a source code file. Then if there is no ambiguity for a given class, the class can be used by only using its short name (`Tojp` in our case).

Importing a package can be done as follows:

```
import ch.ethz.tojp.*;
```

Importing a class can be done as follows:

```
import ch.ethz.tojp.Tojp;
```

3.3 Standard Packages

By default, the package `java.lang` is imported. This package contains most basic classes like `String`, `Runtime`, `System`, `Integer`, `Long`, `Thread`, `Object`...

Other important packages are `java.io` that contains all input/output facilities, `java.awt` that contains a first basic framework for graphical user interfaces (GUI), `java.net` that contains network related classes, `java.util` that contains collections and sets, `java.rmi` that contains facilities for RPC-like mechanisms, `javax.swing` that contains an advanced GUI framework and `org.omg.CORBA` that provides CORBA facilities.

Table 6: Java Primitive Types

Type	**Represents**	**Default**	**Size**	**Min**	Example	**Max**
boolean	*true* or *false*	false	1 bit		`true`	
char	unicode chars	\u000	16 bits	\u0000	'c'	\uFFFF
byte	signed int	0	8 bits	-128	50B	127
short	signed int	0	16 bits	-32768	500S	32767
int	signed int	0	32 bits	-2147483648	250	2147483647
long	signed int	0	64 bits	-2^{63}	4587L	$2^{63}-1$
float	IEEE 754	0.0	32 bits	$\pm 3.4E+38$	34.545F	$\pm 1.4E-45$
double	IEEE 754	0.0	64 bits	$\pm 1.7E+308$	23.54e5	$\pm 4.9E-324$

3.4 Exercises

1. Define your own package name for HelloWorld.
2. Reference HelloWorld in a class outside the package.

4 Types

4.1 Primitive Types

The first kind of types that exists in Java is primitive types, the other one being reference types. Primitive types (see Table 6 for details) are not classes. Contrary to other types, non-local variables of these types are automatically initialized to valid values. It is also possible to use standard arithmetic operators (`+`, `-`, `*`, `/`) with numbers as well as comparison operators (`<`, `>`, `==`, `>=`, `<=`, `!=`). There is an automatic conversion policy for numbers when used in formulae and if there is no loss of precision during the conversion. Contrary to the C language from which it borrows the types names, there is no possibility to assign a number to a char and vice-versa.

Primitive types have equivalent classes: `boolean`'s equivalent is `Boolean`, `char`'s equivalent is `Character`, `int`'s equivalent is `Integer`, `byte`'s equivalent is `Byte`, `short`'s equivalent is `Short`, `float`'s equivalent is `Float`, and `double`'s equivalent is `Double`. Note that in the most recent version of Java, SUN has introduced automatic boxing in the language. This allows to use `int` variables and `Integer` in an interchangeable manner while calculating arithmetic formulae and making assignments.

4.2 Reference Types

Reference types are types for which an instance of a class is the actual value of the variable. The reserved keyword for denoting unassigned reference variables is `null`. It is also the default value assigned to any uninitialized reference type variable. Note that `null` can be returned by any method returning a reference type.

4.3 Classes

Named classes begin with a visibility modifier as described in Figure 7 the keyword `class`, a name, an opening brace ({). It ends with a closing brace (}).

By default, a class is only accessible from within its own package.

The following modifiers can be applied to a classes definitions: `abstract` is for saying that the class cannot be instantiated and must be subclassed to be used, `final` means that the class cannot be subclassed, `public` means that the class is accessible from other packages.

As we already wrote in Subsection 3.1, class names begin by convention with an upper case character, they only contain letters and have upper case characters at each beginning of a new word in the class name. This is often referred to as CamelCase.[4]

As an example, the class HelloWorld could be declared as follows:

```
public class HelloWorld {

}
```

4.4 Interfaces

Interfaces are equivalent to abstract classes because they cannot be used for creating instances. Interfaces do not contain any implementation, only signature of methods (see Subsection 6.2). Interfaces begin with the keyword `interface`, a name, an opening brace ({). It ends with a closing brace (}). By default, interfaces are only accessible within their package. A `public` interface means that the interface is accessible from other packages.

Most interfaces are called *something*`able` because they act as an enabler for a property.

As an example, the toy interface `Wavable` could be declared as follows:

```
public interface Wavable{

}
```

4.5 Inheritance

Java has single inheritance, which means that a class can have one parent class only. Methods are directly inherited in children classes. Their name do not change and they can be used as is if the client code respects the given visibility for the methods. At the top of the inheritance tree, the class `Object` defines the set of common methods.

A class specifies that it inherits from another one by writing `extends` *OtherClass* between its name and the opening brace. As an example:

```
public class HelloWorld extends Object {

}
```

If no parent class is specified, the class inherits from `Object` by default. In a child class, methods and fields from the parent are directly accessible. There is no way to rename methods nor fields. Please see Subsection 6.6 for details on overriding methods.

[4] http://en.wikipedia.org/wiki/CamelCase

4.6 Implementation

A class may *implement an interface*. This means that it gives an actual implementation for each of the method signatures from the interface. Note that instances of a class implementing an interface are also of the type of the interface. Thus they can be treated as variables of that type. This constitutes the main use of interfaces: getting rid of multiple inheritance and still being able to automate treatments over classes from several branches of the inheritance tree.

The class HelloWorld implementing Wavable, it could be declared as follows:

```
public class HelloWorld implements Wavable {
...
}
```

4.7 Nested/Inner Classes

Nested classes are classes that are defined in another class. If a class is defined at the same level (outside) as the main class of the file, then it is implicitly defined as a static class. This means that it can be accessed from outside the class by writing *NameOfMainClass*.*NameOfNestedClass*. If not, it may be defined as a static or non-static class.

Inner classes are classes defined within another main class and are non-`static` classes. They can inherit from a static class. They cannot declare static members other than constant (`final`) ones. There are several ways of defining inner classes. A local class (a class defined for the scope of a method) may be defined as a variable. The scope of the class is then the immediately enclosing block. As an example:

```
// method declaration
public void wave(){
   class Hello2 extends HelloWorld{

   };
   ...
}
```

Inner classes can also be anonymous. Please remember that they are non-`abstract` and `final` in addition of being non-`static`. It is then made on-the-fly when calling a constructor by opening a brace ({) immediately after the call (before the `;`) and beginning to define/redefine methods before closing the brace. This kind of declaration is very useful when willing to redefine a class for only one variable as it happens very often for graphical user interfaces. Example:

```
public void wave(){
   Object o = new Object(){
      public Object clone(){
         return this;
      }
   };
...
}
```

The last possibility for describing inner classes is to include a class definition at the same level as the class's methods. In that case, the scope of the class is the inner class itself. Example:

```
public class HelloWorld extends Object{
   class Hello2 extends HelloWorld{

   };
   // method declaration
   public void wave(){
   ...
   }
}
```

4.8 Generic Classes

With the most recent version of Java genericicity has been introduced. The way to declare generics is to add the class name that is used to parameterize the generic type just after the class name of the generic type. The syntax used to declare the generic parameter is the angle brackets `<` and `>`. If a generic type is parameterized using the type E, it is defined as a regular type within the scope of the generic class. When declaring a variable of the generic type, the generic parameter must be actualized using the angle brackets.

Example:

```
public class List<E>{
   // method declaration
   E m(){
      ...
   }
   // actual type
   Iterator<Integer> i;
}
```

Declaring a generic type that uses more than one parameter is done through using of commas between generic parameters. As an example, this is the declaration of a generic type that uses two types internally, given by the two generic parameters:

```
public class DoubleList<E,G>{
...
}
```

4.9 Arrays

Arrays are declared similarly to C/C++: by adding `[ ]` after the type of which we are considering the array. When actually instanciating the array itself a new type is defined. As an example:

```
// defines an array of int called a
int[] a;
```

```
// defines an array of HelloWorld called hellos
HelloWorld[] hellos;
```

4.10 Exercises

1. Why would one use interfaces?
2. What is the difference between interfaces and abstract classes?
3. Why use inner classes?
4. What is the difference between primitive types and reference types?

5 Variables

5.1 Declaring Variables

Variables are declared using C-style declaration: first the type of the variable and then the name of the variable.

```
int a;
String c;
```

Each declaration ends with a semicolon (`;`) and several variable names sharing the same type can be declared at once by separating variable names with a comma (`,`).

```
int a,b;
String c;
```

5.2 Using Variables

Assigning a variable is made through the use of `=` and assignments at initialization can be done that way.

```
int a=3;
String c="tojp";
// defines an array of int called b and iniotialized textually
int []b={3,4,5,6};
// defines an array of HelloWorld called hellos
HelloWorld [] hellos;
...
a=a+2;
// initialization of hellos
// creates an array of size 10
hellos=new HelloWorld[10];
```

Note that compact forms for assignments of primitive types are also available as in C. As an example, `i++` is equivalent to `i=i+1`, `i--` is equivalent to `i=i-1`, `i-=m` is equivalent to `i=i-m`, and `i+=m` is equivalent to `i=i+m`.

Values stored in an array can be accessed by providing an index into the array: *nameOfArray*`[` *integerValue*`]`. Arrays values are indexed from 0 to n-1 where n is the size of the array. As an example:

```
int a;
int []b={3,4,5,6};
...
a=b[2];
// a is now equal to 5
```

Another way of using variables is to test their value. By default, the usual comparison operators (`==`, `!=`, `<`, `>`, `=<`, `>=`) apply naturally on number types. Equality/inequality can also be tested on variables of reference types, denoting respectively then *reference equality* and *reference inequality*.

5.3 Local Variables

Unlike C or other languages, the declaration of local variables can be made at any point in the code as long as they do not compete with any other declaration of another local variable. Still, it is usually done at the beginning of the method. The scope of a local variable starts after its declaration and is valid through the the whole innermost block (group of instructions between { and }).

5.4 `this`

Within instances methods, it is possible to reference the current object by using `this`. This can be usefull when another declaration shadows a variable or to use the current object as a parameter of another method. As an example:

```
int a;
...
public void add(int a){
   this.a=this.a+a;
   System.out.println(this);
}
```

5.5 Instance Variables

Instance variables are variables declared within a class but outside a method. Each instance has its own values for instance variables (even instances of subclasses). Instance variables can have a visibility qualifier (see Table 7) expressed in first position of their declaration.

Table 7: Visibility Modifiers

Qualifier	`public`	`protected`	*default*	`private`
Not in package/Not in a subclass	yes	no	no	no
In subclass	yes	yes	no	no
Within package	yes	yes	yes	no

Instance variables may be accessed through the use of `this.`*variableName* or more simply *variableName*. As an example, consider that `HelloWorld` has a field giving the message to print:

```
public class HelloWorld {
public String message = "Hello World!";
...
   System.out.println(this.message);
...
   System.out.println(message);
...
}
```

Instance variables can also have the following modifiers: `final`, `transient`, `volatile`. Fields that are `final` can only be assigned once and must be assigned in any constructor. Fields that are `transient` do not participate to the persistent state of the object (they will not be serialized). `volatile` fields may be accessed by threads safely (we will come back to this notion in later lectures). The qualifiers `volatile` and `final` are incompatible.

Possibly `HelloWorld` could declare the field message as `final`:

```
public class HelloWorld {
public final String message = "Hello World!";
...
}
```

5.6 Class Variables

Class variables have the qualifier `static`. Instances of the class all share the same value for the variable. Accessing class variables may be made through the use of *ClassName*`.`*variableName* or more simply within the class by simply using *variableName*.

Visibility as explained in Table 7 is still usable as well as the qualifiers `final`, `transient` and `volatile`.

In the case of `HelloWorld`, it is possible to declare the message as a class variable:

```
public class HelloWorld {
public static String message = "Hello World!";
...
   System.out.println(HelloWorld.message);
...
   System.out.println(message);
...
}
```

5.7 Exercises

1. Why is it incompatible to have a `final` and `volatile` field?
2. Would you use a `static` qualifier for a multithreaded application? Would you use another qualifier?
3. Is a local variable accessible from outside the class?
4. Is a local variable accessible from outside the code in which it is defined?

5. How would you declare a *String* that should be visible outside the package, shared by all instances and may be accessed concurrently?

6 Methods

6.1 Constructors

A constructor is a creation procedure. Constructors have the same visibility modifiers as the ones described in Table 7. Constructors in Java have the same name as the class for which they build instances. Constructors do not have return types (implicitely they return a fresh instance of their class). Constructors are not inherited by child classes. Constructors *must* contain a call to a constructor of the parent class (using `super` instead of the constructor name), if they fail to do that, a call to the default constructor with no argument is added by the compiler. If this still fails, there is a compile-time error. Example:

```
/*
 * Class to wave at the world.
 **/
public class HelloWorld {

   /*
    * The message that will be displayed
    **/
   public String message="Hello World!";

   /*
    * Constructor that changes the message displayed
    **/
   public HelloWorld(String s){
      super();
      message=s;
   }
}
```

Calling a constructor in a client class is done by adding the keyword `new` in front of the class name and then adding the arguments separated by commas between parentheses. Suppose the example for `HelloWorld`:

```
HelloWorld world=new HelloWorld("Hello my dearest World!!!");
```

A constructor for generic classes has to indicate the bindings of the generic types in its signature:

```
/*
 * Class to wave at the world.
 **/
public class HelloWorld <E>{
   /*
    * Constructor that changes the message displayed
    **/
```

```
   public HelloWorld<E>(String s){
      super();
   }
}
```

A call to a constructor of a generic class must also include the generic parameter:

```
HelloWorld <String> world=
      new HelloWorld<String>("Hello my dearest World!!!");
```

6.2 Declaration

A method is declared through writing:

visibilityModifiers otherModifiers ReturnType methodName `(` *arguments* `){`
...implementation...
`}` *visibilityModifiers* are the same ones as the ones shown in Table 7. *otherModifiers* can be:

`abstract`: The method is not implemented, only the signature is given, the {*...implementation...*} is replaced by `;`. The class is declared `abstract`;

`static`: The method is a class method. It can be called by using *ClassName*`.`*methodName* and does not need an instance.

`final`: The method cannot be overridden (through inheritance).

`native`: The method is platform dependent and implemented natively.

`strictfp`: Expressions of type float or double in the body of this method are FP-strict: all results will confirm to the EEE 754 arithmetic on operands.

`synchronized`: All `synchronized` methods of the class execute in mutual exclusion if they have the same scope (instance or class).

ReturnType can be a reference type or a primitive type. It can also be `void` if there is no return type. In the code for the method's body, it is possible to interrupt the computation by using `return` *anExpression*, that will return the value calculated by the expression.

methodName is generally a verbal proposition in which there are only characters and in which words are separated by having the first letter of each word be a capital letter.

arguments are C-style declarations of types and local name bindings within the method (optional in the case of an `abstract` method). Each argument declaration is made of a type and a name, declarations are separated by commas. As in C it is possible to declare methods with a variable number of arguments. Arguments are then automatically "boxed" into an array. Within the method, they can then be used as if they were in an array.

Examples method declaration:

```
public abstract Void wave();

public int getCount(){
   return count;
}

public in countArgs(String... args){

   String s0=args[0];
}
```

Interestingly enough subtyping may become unsafe when using generics. As an example, if one assigns a list of drivers to a list of persons, then the system may allow to add persons that are not drivers and then corrupt the list of drivers without breaking subtyping.

To solve this problem, the specification for the generics follow a policy that is based on the use of wildcards when trying to use their types. Then the super type of all Lists is:

```
public printList(List<?> l){
...
}
```

In this case we make no assumption on the actual type of the generic type of l. It is however useful to use methods from a certain type sometimes. As an example, the method `printList` can be defined as follows:

```
public printList(List<? extends MyClass> l){
...
}
```

The same thing is possible using the keyword `super` instead of `extends` and to combine several conditions using `&`. The excellent tutorial from Gilad Bracha "Generics in the Java Programming Language"[5] explains these constructs in more details.

6.3 Using Methods

Only reference types accept methods invocations. Generally, a method is called on an instance and possibly returns a result. Given a method declared with `static`, the method can be called using the class name instead of an instance name. The call itself is made using parentheses to delimit arguments and one need to put the parentheses even if there is no argument. Example:

```
out2=out.clone();
out2.println("Hello World!");
```

If a method returns a result, calls may be chained and applied with a default left side parentheses. It can also be combined with fields accesses.

```
System.out.println("Hello World!");
```

[5] http://java.sun.com/j2se/1.5/pdf/generics-tutorial.pdf

6.4 Method Conformance

A method m_2 conforms to another method m_1 if:

- they have the same name.
- they have the same number of formal parameters and each parameter has the same type in each method.
- the return type of m_2 is the same or a subtype of the one of m_1.

6.5 Overloading Methods

Overloading methods is giving the same name to several methods and have each of them provide different implementations. As long as methods have different (non-conforming) signatures, it is allowed to overload them. This allows to make the binding at load-time.

6.6 Overriding Methods

It was already stated that children classes inherit methods as they are defined in the parent (e.g. no renaming). These methods can be used freely on the instances of the subclass if the visibility modifyer allows it.

Overriding methods is the ability to use polymorphism by redefining a method with a conforming method that has an equivalent or more permissive visibility. In that case, even if the instance is stored in a variable of parent's type, the redefinition of the method is called in place of the parent's definition. In Java, overriding is implicit. This allows to have a particular treatment for instances of a subclass when used as instances of their parent class.

Still, it can be necessary to access the parent method from the child. Similarly to constructors, it is possible to call a method from the parent by writing `super.`*methodName*.

6.7 Visibility

Visibility modifiers for methods are the same as the ones described in Table 7.

6.8 The `main` Method

In each class it is possible to implement a `main` method that allows to startup the system. The arguments are stored in an array of Strings. The method should have the following definition:

```
public static void main(String []args){
...
}
```

Launching the application can be made by calling `java` on the class containing the definition of the `main` method. Example:

```
java HelloWorld
```

6.9 Exercises

1. Why does the following code not compile?

```
public class HelloWorld {
   public String message="Hello World!";
   public HelloWorld(String s){
      super();
      message=s;
   }
}
public class HelloWorld2 extends HelloWorld{
   public HelloWorld2(String s){
      message=s;
   }
}
```

2. And why does this one not compile either?

```
public class HelloWorld {
   public String message="Hello World!";
   private HelloWorld(String s){
      super();
      message=s;
   }
}
public class HelloWorld2 extends HelloWorld{
   public HelloWorld2(String s){
      super(s);
   }
}
```

3. What will happen here?

```
public class Parent {
   public Parent wave(Parent p){
      System.out.println("in the parent");
      return null;
   }
}
public class Child extends Parent{
   public Child wave(Object o){
      System.out.println("in the child");
      return null;
   }
   public static void main(String []args){
      Parent p=new Child();
      p.wave(p);
   }
}
```

4. And here?

```
public class Parent {
   public Parent wave(Parent p){
      System.out.println("in the parent");
      return null;
   }
}
public class Child extends Parent{
   public Child wave(Parent p){
      System.out.println("in the child");
      return null;
   }
   public static void main(String []args){
      Parent p=new Child();
      p.wave(p);
   }

}
```

5. What is invalid here?

```
public class HelloWorld{
   String s="Hello World";
   public static void main(String []args){
      System.out.println(s);
   }
}
```

7 Expressions

7.1 Instructions

A statement can be:

- An empty statement (e.g. `;`).
- A labeled statement to be used in conjunction with `break` or `continue`.[6]
- An expression statement that is calculated. Note that it covers method invocations and calculations.
- A control structure that is affecting the execution.
- An assert statement that tries to verify a condition if assertions are on.[7]
- A return statement that terminates the methods invocation and possibly returns a value.
- A throw statement that generates a new exception.

[6] http://java.sun.com/docs/books/jls/third_edition/html/statements.html#14.7
[7] http://java.sun.com/docs/books/jls/third_edition/html/statements.html#14.10

- A try statement that catches an exception.

At the end of each non-block statement, a `;` is needed.

7.2 Method invocation

As we already expressed, a method is invoked through a call of the form *variableName*`.`*methodName* or *ClassName*`.`*methodName*. This expression can also be used on an expression returning a reference type variable instead of *variableName*. Note that even if the method returns a result, it can be ignored by not assigning it to a variable.

7.3 instanceof

The operator `instanceof` is a primitive of the language that allows programmers to verify that an instance is an instance of a given class. In particular it also considers that instances of subclasses are instances of the parent classes. The expression *variableName* `instanceof` *ClassName* returns a boolean that is `true` if the variable is an instance of the class, `false` otherwise. This expression can also be used on an expression returning a reference type variable instead of *variableName*.

7.4 Casting

A cast is a way of having an instance conform to the type of a variable by trying to change its declared type. It is made by prefixing the reference to be cast by `(`*TypeName*`)`. This used to be extremely useful before the existence of generic types in Java because any object being put in a container would loose its type indication when retrieved. Example of use:

```
String s;
Object o;
...
s=(String)o;
```

7.5 Blocks

A block is an instruction terminated by a semi-colon (`;`) or a group of instructions packed by { and }. Executing a block means that each instruction is executed one after the other. Blocks are instructions.

7.6 `if`

If statements have two different forms:

`if (`*test* `)` *block*

`if (`*test* `)` $block_1$ `else` $block_2$

The first expression executes *block* only if the evaluation of the *test* returns `true`.

The second expression executes $block_1$ only if the evaluation of the *test* returns `true`, otherwise it executes $block_2$. Example:

```
if (a>0)
   a=a+1;
else
   a=a-1;
```

7.7 switch

Switch statements have the following form:

`switch (`*expression* `){`
  `case` $constant_1$`:` $block_1$
  `case` $constant_2$`:` $block_n$
...
  `default:` $block_{final}$
}

As in C or C++, the execution of a `switch` statement is as follows: if the value of the expression (that must be of type `char`, `byte`, `int`, `short`, `Character`, `Byte`, `Integer` or `Short`) is equal to one of the constants, then all the blocks following this constant will be executed. When there is a *break* instruction, the code continues execution after the end of the `switch` statement. In the case the expression does not evaluate to any constant, the final block is executed.

Example of use:

```
int a;
switch (a){
   case 1:
      System.out.println(1);
      break;
   case 2:
      System.out.println(2);
      break;
   default:
      System.out.println("N/A");
}
```

7.8 while

While statements have the following form:

`while (`*test* `)` *block* `;`

While the *test* is `true`, the block is executed. Example:

```
while(true){
   i=i+1;
}
```

7.9 do...while

Do...while statements have the following form:

`do` *block* `while(` *expression* `);`

Execute the block and do it again until the *expression* is `false`. Example:

```
do{
   i=i+1;
}while(i<10);
```

7.10 `for`

For statements have the following form:

`for(`*initInstruction*`;`*test*`;`*loopInstruction*`)` *block*

The semantics of the statement is that *initInstruction* is executed, then the *test* is verified, if it returns `false` the next statement is executed. While the *test* is verified *block* is executed, then *loopInstruction* is executed and then *test* verified. Example:

```
for(i=0;i<10;i++) System.out.println("I print this 10 times");
```

7.11 Enhanced `for`

In the case we are considering `Iterable` variables, there is a form for a for statement that applies to all the elements.

`for(`*Type Identifier*`:`*expression*`)` *block*

The only condition is that *expression* return an `Iterable`. In that case, it is equivalent[8] to:

```
for (I #i = Expression.iterator(); #i.hasNext(); ) {
   VariableModifiersopt Type Identifier = #i.next();
   block
}
```

Where `#i` is a fresh variable. As an example:

```
int sum(int[] a) {
   int sum = 0;
   for (int i : a)
       sum += i;
   return sum;
}
```

7.12 `break`/`continue` and labels

The `break` instruction allows to break any loop (`while`, `for`, `do...while`) and continue with the execution of the rest of the program. `continue` allows to go back to the *test* part of the loop and skip the rest of the loop. Note that `break` does not act on `if` statements, thus it is possible to exit from a loop by using `break` in an `if` statement:

```
while(true){
   i=i+1;
   if (i>10) break;
}
```

[8] Example from http://java.sun.com/docs/books/jls/third_edition/html/statements.html#14.14.2

A less known way of using `break` and `continue` is to use them with labels. It is possible to use labels as follows:

label: *block*

Actually when used as `break` *label*`;` within a *block* the execution interrupts the block that has the same label and continues execution.

In the case the label is defined on a loop statement, it is also possible to use `continue` *label*`;` to effect a `continue` on this loop rather than on the innermost loop. Example:

```
test:{
      System.out.println("beginning test");
      while (true){
         if (i<5) {
            break test;
         } else i++;
         System.out.println("end test");
      }
      }

test2:while (true) {
         System.out.println("beginning test2");
         if (i<5) {
            break test2;
         } else {
            i++;
            continue test2;
         }
      }
```

7.13 Exercises

1. Look up in the documentation[9] for the documentation on assert.

2. What does the following program do:

   ```
   while (true) {
      System.out.println("beginning test2");
      if (i<5) {
         break ;
      } else {
         i++;
         continue ;
      }
   }
   ```

3. And this one?

   ```
   int i;
   test:for (;true;) {
   ```

[9] http://java.sun.com/docs/books/jls/third_edition/html/statements.html

```
        while(true){
        System.out.println("beginning test2");
           if (i<5) {
              break test;
           } else {
              i++;
              continue ;
           }
        }
     }
```

4. Find a good question for this section and exchange them with your most direct neighbor.

8 Exceptions and Errors

8.1 `Throwable`

Contrary to other languages, exceptions in Java can be used a programming style. Still, it is important to differentiate between real errors and exceptions. `Throwable` is a class that is subclassed by both `Error` and `Exception`.

The `throw` statement allows to raise the throwable and interrupt execution until a corresponding `catch` block is found. Example of use:

```
throw new ActivationException("message");
```

When there is a `Throwable` that is raised execution is blocked and enclosing blocks and methods are terminated until a block containing a matching `catch` clause is found. The syntax of `try`/`catch` statements is the following one:

`try` *block*
`catch (` $ThrowableType_1$ $varName_1$ `)` $block_1$
`catch (` $ThrowableType_2$ $varName_2$`)` $block_2$
...
`finally` $block_f$

It has the basic semantics of trying to execute *block* and if a `Throwable` is called during the execution and that the type of the `Throwable` is a subtype of at least one of the variables of the `catch` clauses, then the corresponding block of the first matching clause is executed. In the case the optional `finally` clause is present, the $block_f$ is also executed. And program continues execution. If no clause matches, then the optional the $block_f$ is executed if present and then the `Throwable` is propagated.

In the case there is no `Throwable` raised, $block_f$ is executed after *block* and the program continues execution normally. An example of use is the following:

```
public int getCount() {
   if (count>1000) throw new TooManyElementsError();
   return count;
}
...
try{
```

```
   a=getCount();
} catch (Error e){
   System.out.println("Too many elements!!!");
}
...
```

8.2 Error

`Error` mostly consist of problems that cannot be resolved like `LinkageError`, `ThreadDeath`, `VirtualMachineError`. It is not needed to declare that a method may throw an `Error` as it generally cannot be foreseen.

8.3 Exceptions

There are plenty of exceptions, we can however cite common ones like: `ClassNotFoundException`, `IOException`, `ArrayIndexOutOfBoundsException`...

A method that may raise an exception needs to declare that it `throws` an exception. Thus programmers cannot be surprised by exceptions that they did not foresee. This indication is added just before the body of the method. As an example:

```
public int getCount() throws TooManyElementsException {
   if (count>1000) throw new TooManyElementsException();
   return count;
}
...
try{
   a=getcount();
} catch (Exception e){
   System.out.println("Too many elements!!!");
}
...
```

8.4 Exercise

1. To treat errors happening on sockets, would you define exceptions or errors?

2. When willing to use a persistence framework if the main storage is not accessible, would you use an exception or an error to report it?

3. Are exceptions gotos? Give some other point of view than the one used in Java.

4. what does the following code do?

   ```
   public int ping() throws PingException {
      System.out.println("Ping")
      throw new PingException();
   }
   ...
   ```

```
public int pong() throws PongException {
   System.out.println("Pong")
   throw new PongException();
}
...
boolean b;
while (true)
   try{
      if (b)
         ping();
      else
         pong();
   } catch (PingException pi){
      b=false;
   }catch (PongException po){
      b=true;
   }
```

9 Standard Classes

9.1 Object

The class `Object` has a limited set of methods. They are all inherited by any class. The main ones are:

clone(): returns a copy of the object (not a deep copy though).

equals(Object obj): returns `true` if the current `Object` is equal to the argument (according to the method) returns `false` otherwise.

finalize(): method called by the garbage collector when the object is removed from the heap.

getClass(): the method returns the actual class of the object.

hashCode(): method that returns a hash code for the current object. Note that for objects that are equals regarding to `equals` should have the same `hashCode`.

toString(): returns a `String` that represents the current object.

The other methods of `Object` are used for concurrent accesses.

9.2 String

The class `String` is a class that is immutable in Java. This means that the value of the instances of this class cannot change once created. Some of the methods of the class `String` are:

charAt(int index): returns the char at the position specified.

concat(String str): returns a `String` that is a concatenation of the current `String` and the argument.

equals(Object anObject): returns `true` if the argument is a `String` with the same value.

length(): returns the size of the current `String`.

matches(String regex): returns `true` if the current `String` matches the regular expression passed as an argument.

Please note also the existence of a whole set of `static` methods called `valueOf` and that returns the String representations of values from a primitive type.

9.3 System

The `System` is a class that intends to provide a library for useful functionalities (`static` methods) related to the interactions with the system. In particular:

exit(int status): quits the program and sends the return code `status` to the system.

getenv(): returns a `Map` of environment variables and values.

nanoTime(): returns a `long` containing the most precise available time.

Moreover, the three fields `err` to output on the error channel, `in` to read users inputs, `out` to output on standar outpout.

9.4 Vector

The class `Vector` is a generic class that allows to store in an array-like structure values of a given type. The methods of the class `Vector`:

add(element): appends the element to the current `Vector`.

clear(): removes all the elements of the current `Vector`.

elementAt(int index): returns the element at position `index`.

remove(Object o): removes `o` from the current `Vector`.

size(): returns an `int` giving the size of the current `Vector`.

9.5 Exercises

1. Inspect the class `Array`.
2. Inspect the interface `Iterable`.
3. Inspet the class `OutputStream`.

10 Basic Examples

10.1 Hello World

The most simple version of Helloworld is:

```
/**
  * Class to startup students
  *
  *@author Manuel Oriol
  */
public class HelloWorld {
   public static void main(String[] args){
      System.out.println("Hello World!");
  }
}
```

10.2 A Small Note taker

```
import java.io.*;
/**
  * Class to simply take notes into a file.
  */
public class NoteTaker{

   /**
     * Reader from the keyboard
     */
   public static BufferedReader standard;

   /**
     * Stream to write in the file
     */
   public static FileOutputStream out;

   /**
     * This program taks the first argument as a file name,
     * it opens it and write what user write into it
     */
   public static void main(String[] args){
      String s;

      standard = new BufferedReader(new InputStreamReader(System.in));

      // checks arguments number
      if (args.length!=1) System.exit(0);

      // open the file name
      try {out = new FileOutputStream(args[0]);}
      catch (FileNotFoundException e){System.exit(0);}
```

```
      // users have to leave by using Control-C
      while(true){
         try {
            // read and write
            s=standard.readLine();
            out.write(s.getBytes());
            out.write("\n".getBytes());
         } catch (IOException e){
            System.out.println("I/O error");
            System.exit(0);
         }

      }

   }

}
```

10.3 A Small Stack

We design here a tiny example of stack to show how generics work in practice. For a nice implementation of `Stack`, please use `java.util.Stack`.

```
/**
  * Small Stack Class to serve as an example for generics.
  *
  *@author Manuel Oriol
  */
public class MySmallStack <E> {

   /**
     * Private class for the Elements of the stack.
     */
   private class Element <E>{
      public Element<E> next;
      public Element<E> previous;
      public E value;
      public Element(E value){
         this.value=value;
      }
      public E getValue(){
         return value;
      }
   };

   /**
     * First element of the stack.
     */
   private Element<E> first;
```

```
   /**
     * Constructor for the stack.
     */
   public MySmallStack(){
      first=null;
   }

   /**
     * Method that adds a value to the stack.
     *
     *@param value the value to add
     */
   public void add(E value){
      Element<E> e;
      e= new Element<E>(value);
      e.next=this.first;
      this.first=e;
   }

   /**
     * Method that removes a value from the stack.
     *
     *@return the removed value
     */
   public E remove(){
      E retValue;
      if (this.first!=null){
         retValue=this.first.getValue();
         this.first=this.first.next;
         if (this.first!=null) this.first.previous=null;
         return retValue;
      }
      return null;

   }
   /**
     * Test for the stack. The expected result is 3 2 1
     */
    public static void main(String[] args){
      MySmallStack<Integer> myStack = new  MySmallStack<Integer>();
      myStack.add(1);
      myStack.add(2);
      myStack.add(3);
      System.out.println(myStack.remove());
      System.out.println(myStack.remove());
      System.out.println(myStack.remove());
   }


}
```