

Some definitions for Introduction to Programming (Java)

This document contains some precious definitions about Introduction to Programming, with Java as programming language supported. All of these definitions do not come from my mind, I just wrote a summary of the exhaustive teaching material of prof. [Diego Calvanese](http://www.inf.unibz.it/~calvanese) of the free University of Bolzano. You can read his whole work at this url: <http://www.inf.unibz.it/~calvanese/teaching/06-07-ip/lecture-notes/>

I hope you will find the following interesting and useful. Enjoy it!

A **class** describes a collection of encapsulated instance variables and methods (functions).

Objects are the entities that can be manipulated by programs. Each object belongs to a class (we say it is an instance of the class).

The **instances** of a class are the set of objects that belongs to the class.

A **variable** represents a memory location that can be used to store the reference to an object. The declaration of a variable reserves a memory location for the variable and makes the variable available in the part of the program where the declaration appears.

Initializing a variable means specifying an initial value to assign to it. A variable that is not initialized cannot be used until it is assigned such a value.

In Java, variables cannot contain objects, but only references to objects: the objects are constructed and allocated in memory independently from the declaration of variables

A **local variable** is a variable contained in the body of a method

A **global variable** is a variable declared inside the class, but outside any method, and is qualified as static.

Instance variables are variables defined inside the class, but outside the body of a method. Those variables are preceded by an access modifier (usually private) and are always initialized when the object is created. Instance variables are associated to the single objects and not to the entire class. Instance variables are always visible in all methods of the class

The **scope of a variable** is the region of a program in which the variable is visible. In Java, the scope of a local variable is the body of a method in which it is declared.

The **lifetime of a variable** is the time during which the variable stays in memory and is therefore accessible during program execution

The **value of a variable of type reference to object** is a reference to an object, and not the object itself

The **value of a variable of a primitive type** is a value of the primitive type itself (and not the reference to a value or an object)

Casting consists in the conversion of the type of an expression to another type:

```
Int a = (int) 3.75 // a = 3
```

The static method `parseInt()` of the class `Integer` is used to obtain the number corresponding to the string read as a value of type `int`.

A **method** is a callable function defined within one or more classes.

The **parameters** of a method represent the arguments that the calling block passes to the method.

The **signature of a method** consists of the name of the method and the description of its parameters.

The **header of a method** consists of the signature plus the description of the result

The **definition of a method** contains in the header a list of formal parameters. The call of a method contains the parameters that have to be used as arguments. Such parameters are called actual

parameters.

Methods with the same name are said to be **overloaded**

Static methods are methods that do not require an invocation object. The name of a static method is preceded by the name of the class to which the method belongs (ex: Math.abs(3);).

All the instance methods have an implicit formal parameter denoted by **this**, which is used to access all instance variables and the methods of the invocation object.

```
public class Person {
    // instance variables
    private String name;
    private String residence;

    // methods
    public String getName(){
        return this.name;
    }
    public String getResidence(){
        return this.residence;
    }
    public void setResidence(String residence){
        // instance variable | parameter
        this.residence = residence;
    }
}
```

A **constructor** is a (non static) method of a class that has the same name as the class and does not have an explicit return value. It is automatically called by the virtual machine when an object is created using the new operator

```
Person p; // variable p of type reference to an object of type Person
p = new Person("John Smith", "London"); // creation of a new object Person, assignment its reference to the
variable p
```

When the **return statement** is executed inside a method, it causes the termination of the method and it returns the result of the method to the client module

Public indicates that the method/instance variable is visible outside the class

Private indicates that the method/instance variable is not visible outside the class

```
public class Account {
    public int bal;

    public Account(int x){
        bal = x;
    }
    Account r1, r2;
    r1 = new Account(100);
    r2 = r1;
    r2.bal = 234;
    System.out.print(r1.bal); // PRINTS 234
}
```

Java is equipped with a **selection operator** that allows us to construct conditional expressions.

```
Condition ? DoThisIfTrue : DoThisIfFalse
```

To **compare two strings**, we have to use the equals() method, and not the operator ==

To **compare two objects**, we have to define a suitable method that takes into account the structure of the objects of the class (all its variables and a lot of && conditions)

```
public boolean equalTo(BankAccount ba){
    return this.name.equals(ba.surname) &&
           this.surname.equals(ba.surname) &&
           this.balance == ba.balance;
}
```

In Java, the repetition of statements is obtained with the use of **loop** statements (or iterative statements) or with the use of recursive methods

Example of while loop: counting the occurrences of a character in a string

```
while (position < string.length()){
    if (string.charAt(pos) == character{
        numChars++;
    }
    position++;
}
```

The statement **break** is used to exit a loop.

The statement **continue** is used to jump to the condition of the loop

An **array** object contains a collection of elements of the same type, each of which is indexed by a number

```
int [] a;
a = new int[5] // 5 is the index of an array of 4 elements, starting from 0
a[0] = 23;
a[3] = 25;
a[4] = 333;
System.out.print(a.length); // prints 4
```

Inheritance consists in the possibility of defining a class that is the specialization of an existing class. A subclass inherits all the methods and all the instance variables of the superclass. To derive a class from another, we use the extends keyword. When the subclass has its own instance variables, its constructor must first construct an object of the superclass (using super()). super() must appear as the first executable statement in the body of the constructor of the derived class. The objects of the derived class are compatible with the objects of the base class.

```
class Student extends Person{
    super(name, surname);
}
```

We say that we do **overriding** of a method m() when we define in the subclass a method m() having exactly the same signature as the method m() in the superclass. The method we are defining must have the same header as the original method.

The overriding of methods causes **polymorphism**, which means the presence in a class hierarchy of methods with the same signature that behave differently.

All classes defined in Java are subclasses of the predefined class Object, methods such equals() and toString() are inherited from Object

To write / read a string on a file:

```
import java.io.*

public static void writeOnFile(String[] textToBeWritten) throws IOException{
// opening the file for writing
FileWriter f = new FileWriter("file.txt");
// creation of the object for writing
PrintWriter out = new PrintWriter(f);
for (int i=0; i < text.length; i++){
// write something
out.println(textTobeWritten[i]);
}
// close the file and the Output channel
out.close();
f.close()
}

import java.io.*

public static void readFromFile(String[] textToBeWritten)
throws IOException{
// opening the file for writing
FileWriter f = new FileWriter("file.txt");
// creation of the object for writing
BufferedReader in = new BufferedReader(f);
String line = in.readLine();
while (line != null){
// read the text
line = in.readLine();
}
// close the file and the Output channel
out.close();
f.close()
}
}
```

Java defines a common way of handling all input/output devices. A **stream** is a sequence of data generated by a input device or consumed by an output device.

```
InputStream is = System.in; // keyboard
FileInputStream is = new FileInputStream("file.txt") // a file
URL u = new URL("http://bleah.ext");
InputStream is = u.openStrem(); // URL
```

```
OutputStream os = System.out; // video
FileOutputStream os = new FileOutputStream("bleah.txt"); // a file
```

In Java, the errors that occur at runtime are represented by means of **exceptions**.

Exceptions and errors are represented through Java classes. There are two possibilities for handling exceptions:

- handling the exception where it is generated
- handling the exception in another point of the program

Most common types of exceptions:

- `NullPointerException`: when we invoke a method with a null value, or when we try to access a variable with a null reference
- `ArrayIndexOutOfBoundsException`: when we access an array with an index which is less than zero, or greater than the array length
- `IOException`: generated when an input/output error occurs
- `FileNotFoundException`: when we try to open a non-existent file

All Java methods can use the **throws** clause to handle the exceptions caused by the method itself.

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

Remember that if the method a() throws an exception, and a method b() calls a(), b() must throw the same exception!

We can define new exceptions in this way:

```
public class MyException extends Exception{  
    public MyException (String message){  
        super (message);  
    }  
}
```

To **throw** our exception, we use:

```
throw new MyException("message to be printed");
```

We **catch** an exception if we want to handle them in such a way that the program doesn't terminate

```
try{  
    // block to be checked  
}catch(IOException e){  
    // what to do if a IOException occurs  
}catch(NumberFormatException e){  
    // what to do if a NumberFormatException occurs  
    // you can System.out.print(e.getMessage());  
}finally{  
    // block to be executed afterwards  
}
```

When we want to define and manipulate collections of objects, we need **linked lists**, which are linear linked structures that allow us to store collections of elements organized in a form of linear sequence of nodes. For accessing the whole list, we maintain only a reference to the first node of the list, and access the whole list through its first node, by following the links.

A nonempty list is represented by a reference to its first node

An empty list is represented by null

The basic class for a linked list is a class whose objects represent the information associated to a single element (or node) of the structure:

```
class ListNode{
    ElementType info;
    ListNode next;
}
```

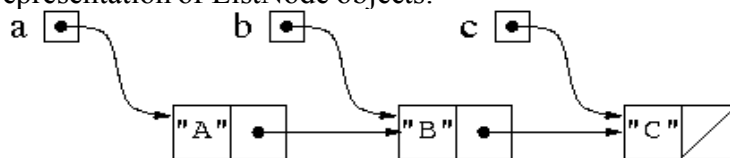
The declaration of the class ListNode should neither be private nor public.

Consider the following example:

```
class ListNode {
    String info;
    ListNode next;
}

public class TestList {
    public static ListNode create3NodesABC() {
        ListNode a = new ListNode();
        ListNode b = new ListNode();
        ListNode c = new ListNode();
        a.info = "A";
        a.next = b;
        b.info = "B";
        b.next = c;
        c.info = "C";
        c.next = null;
        return a;
    }
}
```

The following is the representation of ListNode objects:



If you want to know more about operations on linked lists, I suggest to read the original prof. Calvanese notes about linked lists: <http://www.inf.unibz.it/~calvanese/teaching/06-07-ip/lecture-notes/uni12/uni12-main.html>