Chapter 2

Object-oriented programming is a powerful technique to exploit huge resources stored in the form of classes many of them freely accessible on the Internet. The writing of a program can be replaced here by the developing applications based on the creation, extension and integration of classes. In this chapter we present how to develop classes and create (instantiate) objects.

Class and Object

A class is a collection of data and methods applied to these data.

Take for example a class Disk:

```
public class Disque{
public double x,y;
public double r;
public double circonference() { return r*2*3.14159;}
public double surface() { return r*R*3.14159;}
}
```

By definition:

```
Disque d;
```

we prepare a reference name but the object itself remains to be dynamically created the *new* operator:

```
d = new Disque();  // creation of one instance
d.x = 2.0;
d.y = 4.0;
d.r = 1.0;
```

Then the circumference method can be applied to the object d:

```
d = new Disque();
double c;
d.r = 1.0;
c = d.circonference();
```

Note that the method has no arguments and is applied to this (*this*) object!

The data in a class can be initialized during the instantiation. To do this, we introduce the constructor method. The constructor's name must be the same as the class.

```
public class Disque{
public double x,y;
public double r;
public Disque(double x, double y, double r)  // constructor
  this.x=x;
  this.y=y;
  this.r=r;
  }
public double circonference() { return r*2*3.14159;}
public double surface() { return r*r*3.14159;}
}
```

Thanks to the constructor we instantiate the object disk and initialize variables:

```
Disque d = new Disque(2.0, 4.5, 2.0);
```

In general, a class can be "equipped" with several constructors to instantiate it in different ways. The methods of construction are managed by *overloading* mechanism.

```
public class Disque{
public double x,y;
public double r;
public Disque(double  r) {     // one argumlent
  this.x=0.0;  this.y=0.0;
  this.r=r;   }
public Disque(double x, double y, double r) {    // three arguments
  this.x=x;  this.y=y;
  this.r=r;  }
public double circonference() { return r*2*3.14159;}
public double surface() { return r*r*3.14159;}
}
```

Variables and class constants

In Java, all variables must be declared in classes. The introduction of the word *static* in the definition of a class allows to attach a value to this class and all objects instantiated from that class. We can speak here of a **class variable**.

```
public class Disque{
public double x,y;
public double r;
static int nombre=0;
public Disque(double x, double y, double r) {    this.x=x;  this.y=y;  this.r=r;
  nombre++;     //variable incremented in each new instance
  }
public double circonference() { return r*2*3.14159;}
public double surface() { return r*R*3.14159;}
}
```

The class variables are initialized when the class is loaded for the first time. The instance variables are initialized when an object is created.A class variable can be altered by these bodies. To create a constant to use in all instances must use the term *final*

```
public class Disque{
public double x,y;
public double r;
static int nombre=0;
public static final double pi=3.14159;
public Disque(double x, double y, double r) {
  this.x=x;  this.y=y;  this.r=r;
  nombre++;     // variable incremented in each new instance
  }
public double circonference() { return r*2*pi;}
public double surface() { return r*r*pi;}
}
```

Note: The variables *static final* replace constants defined by *the #define* in C language

Static methods

In some applications, we need external methods defined independently from classes used in our program.

For example, we perform a calculation to decide (yes or no) if the given point falls inside or outside of a Disc.

```
public class Disque{
public double x,y;
public double r;
static int nombre=0;
public static final double pi=3.14159;
public Disque(double x, double y, double r) {
  this.x=x;  this.y=y;  this.r=r;
  nombre++;     //variable incremented in each  new  instance
  }
public double circonference() { return r*2*pi;}
public double surface() { return r*r*pi;}
public boolean siDedans(double xp, double yp)
  {
  double dx = xp - x;
  double dy = yp - y;
  double distance = Math.sqrt(dx*dx - dy*dy);
  if (distance>r) return false; else reurn true;
}
}
```

Math is not a class defined in our application. The method *sqrt*() is a static; this method is defined in the Math class:

```
public class Math {
public static sqrt(x) { .. }
..}
```

In the next class, we have defined two methods *pluspetit*(), one as a class and one as an instance method.

```
public class Disque {
public double x,y;
public double r;
public static Disque pluspetit(Disque a, Disque b) {            // a &  b are  parameters
  if (a.r<b.r) return a; else return b;
  }
public  Disque pluspetit(Disque a) {                            // a is parameter
  if (a.r<r) return a; else return this;
  }
public double circonference() { return r*2*pi;}
public double surface() { return r*r*pi;}
}
```

The class method can be called by:

```
Disque d = Disque.pluspetit(a,b);          // static method ( classe Disque) with  arguments a & b
```

The instance method may be called by:

```
Disque d = a.pluspetit(b);                 // 'instance method  of object  a with argument b
```

Class organization

Classes are grouped into sets called packages. By default, all classes in a package are visible between them. Classes declared as public are accessible from other packages. Classes without this statement are not accessible from another package. Variables or methods are accessible to other classes in the same package. Variables or methods declared as *private* are accessible only within the class. Variables and methods declared as *public* can be accessed from other packages if their classes are also declared in *public*.

Within methods, Java allows the use of local variables. These variables behave as local variables in C language and are inaccessible from outside the methods.

Heritage

A class can be expanded to incorporate other variables and methods. Java lets you define a new class as an extension or a subclass of the base class. For example, we can add to our class disc *Draw* a new method by creating a new class *DisqueDessin*.

```
import java.awt.*;                                   // awt graphic package
public class DisqueDessin extends Disque {
Color bordure,remplissage;
public void Dessiner (DrawWindow df) {
  df.drawCircle(x,y,r,bordure,remplissage)
  }
}
```

The term *extends* indicates that the class DisqueDessin is a subclass of Disk and it inherits variables and methods of this class. The new subclass can be completed by a constructor:

```
import java.awt.*;
public class DisqueDessin extends Disque {
Color bordure,remplissage;
public  DisqueDessin (double x, double y, double r, Color bordure, Color replissage) {
this.x = x;
this.y = y;
this.r = r;
this.bordure = bordure;
this.remplissage = remplissage;
}
public void Dessiner (DrawWindow df) {
  df.drawCircle(x,y,r,bordure,remplissage)
  }
}
```

This constructor can be simplified through the concept of superclass that is the original class:

```
public  DisqueDessin(double x, double y, double r, Color bordure, Color replissage) {
  super(x,y,r);   // initialisation of super class
  this.bordure = bordure;
  this.remplissage = remplissage;
  }
}
```

Calling the superclass must be placed before the first instruction and/or variable constructor.

Note: In case a class is declared without its constructor, the Java compiler adds a default constructor with the call *super ()*.

Hidden Variables

Take for example our extended class *DisqueDessin* and let us add a new method *Resolution*. The same variable name may be used

```
import java.awt.*;
public class DisqueDessin extends Disque {
Color bordure,remplissage;
float r;  // variable d'instance de resolution
public  DisqueDessin (double x, double y, double r, Color bordure, Color remplissage) {
super(x,y,r);  this.bordure = b; this.remplissage = remplissage;  }
public void Resolution(float resolution) { r = resolution; }
public void Dessiner (DrawWindow df) { df.drawCircle(x,y,r,bordure,remplissage)
}
```

The name of the instance variable *resolution* and the class variable for the radius is the same - r. To get there we use two different references:

```
**this**.r                    // for instance variable
**super**.r                   // for classe variable
```

Overloading and methods of coercion

Overloading (overloading) method allows the use of the same name for different methods if their parameter lists differ depending on the number, types and order of their parameters. When calling an overloaded method, the Java compiler selects the appropriate method by examining the lists of parameters.

```
import java.awt.*;
public class DisqueDessin extends Disque {
Color bordure,remplissage;
float sr;                               // low  resolution
double dr;                              // high  resolution
public  DisqueDessin (double x, double y, double r, Color bordure, Color remplissage) {
  super(x,y,r);  this.bordure = b; this.remplissage = remplissage;
  }
public void Resolution(float resolution) { sr = resolution; }
public void Resolution(double resolution) { dr = resolution; }
}
```

Coercion (overriding) of a method allows to replace a "top" method by a derived method. For example:

```
classe Positive {
int i = 5;
int m() { return i;}
}
classe Negative extends Positive {          // derived method
int i = 10;
int m() { return -i;}
}

public class Coercition {
public static void main(String args[]) {
Negative n = new Negative();
System.out.println(Negative.n);      // prints 10
System.out.println(Negative.m());   // prints -10

Positive p = (Positive) n;             // coercion of l'instance n in « top »    Positive
System.out.println(Positive.p);       // prints 5
System.out.println(Positive.m());    // reference to Negative : prints -10
```

Visibility of methods and variables

Java has several access modifiers to open or restrict access to methods and variables. By default, a method can be available within a package and not accessible outside its package.

We present here the significance of access modifiers:

- *public*
- *protected*
- *private*

TABLE 2. **Visibility of variables and Java methods**

	By default	***public***	***protected***	***private protected***	***private***
accessible to a class in the same package	**yes**	**yes**	**yes**	**no**	**no**
accessible to a class in the same package	**yes**	**yes**	**yes**	**no**	**no**
accessible to a class in another package	**no**	**yes**	**no**	**no**	**no**
accessible to a class in another package	**no**	**yes**	**no**	**no**	**no**
inherited by a subclass in the same package	**yes**	**yes**	**yes**	**yes**	**no**
inherited by a subclass in another package	**no**	**yes**	**yes**	**yes**	**no**

Note that a variable or method marked *private* is only available in its class. The variables and private methods are not visible in the sub-classes and they are not inherited in subclasses. A variable and/or protected method (*protected*) is visible to subclasses and classes in the same package.

Tables (arrays) and Strings

Arrays

Arrays and strings are manipulated by reference. They are created dynamically by the *new* statement and they are automatically deleted when they are no longer used

Note: In some situations, the automatic deletion of an array or a portion of the array can cause malfunction of the program that did not re-allocated the necessary space. There are two ways to create an array, the first using the *new* statement,

```
byte tampon[] = new byte[512];  // initializes with null
int valeurs[] = new int[32];    // initializes with  0
object objets[] = new object[4];  // initializes with null
byte tab2dim[][] = new byte[16][32];    // t bi dimensional array
```

the second creates an array by initializing static array.

```
int tabint[] = {1,2,3,4,5,6,7,8,9,0}:
```

Access to an array element is done by inserting his index between brackets [index]:

```
int valeurs[] = new int[32];    // initialises with 0
for(int i; i<valeurs.length;i++)   valeurs[i]  = i;
```

The length attribute is used to extract the table size. Where the proposed index in the table exceeds the size of the table, exception type ArrayIndexOutOfBoundsException is thrown.

Strings

The Java Strings are not terminated by '\ 0'; they are instances of the class *java.lang.String*. The Strings are treated as constant objects, it is not possible to change the contents of a *String*. A *String* can be initialized statically or dynamically by the *new* statement:

```
String chaine = “Ceci est un test”;
String[] titres ={“resistance”, “capacité”};
String donnee = new String(paquet.getData());      // the size of data int the packet must be known
int taille = donnee.length();
```

The management of String can be done by several methods proposed in the package *java.lang*. For example, the size of a String is given by the *length ()* method.

Classes and abstract interfaces

Java allows to define abstract classes. An abstract class can be considered a generic class (superclass). For example we can design a class Figures in which we are going to place our class disc, then a class Rectangle, Triangle, etc.. All these classes can contain methods to calculate the *circumference* () and *surface* ().

```
public abstract  class Forme {
public abstract double circonference();               // abstract method
public abstract double surface();
}
public class Disque extends Forme{
protected final double pi= 3.14159;
public double x,y, r;
public Disque(double x, double y, double r) {  this.x=x;  this.y=y;  this.r=r;  }
public double circonference() { return r*2*pi;}
public double surface() { return r*r*pi;}
}
public class Rectangle extends Forme{
public double x,y;
public Rectangle(double x, double y) {  this.x=x;  this.y=y;  }
public double circonference() { return 2*(x+y);}
public double surface() { return x*y;}
}
```

Methods declared in abstract class *Shape* are abstract methods. Note that an abstract method has no body framed by () A class that contains an abstract method is automatically an abstract class. An abstract class can not be instantiated. A subclass that does not implement all methods of abstract class is also an abstract class.

An abstract class can be used to instantiate subclasses.

```
Forme[] formes= new Forme[2];                // form for future instances ; not an instanciation !
formes[1] = new Rectangle(5.2,8.4);
formes[2] = new Disque(2.5,5.0,1.0);
double surface_totale = 0.0;
for(int i=0;i<formes.length;i++) surface_totale += formes[i].surface();
```

Interfaces

Java does not allow multiple inheritance - a class inherits from more than one class - but it provides the notion of interface that allows to simulate multiple inheritance. The interfaces are abstract. This means that, like abstract classes, they can not be instantiated. Interfaces only provide definitions of abstract methods. They must be implemented using the same names and parameters in the implementation class. An interface can contain definitions of constants (*static final*).

A class can implement one or more interfaces, while possibly inheriting from another class. This mechanism can simulate the properties of sub-typing of multiple inheritance: all types related to a class are not subtypes of each other. Take for example our classes *Rectangle* and *Disc* and try to expand them by drawing methods: *DessinerDisque() DessinerRectangle().* We can then create an abstract class *DessinerForme*. We also want to keep our classes and methods *circumference*() and *surface*(). But we already have a superclass, *Shape*. To resolve this problem, we can define an interface *DessinerForme*.

```
public interface DessinerForme {
public void Couleur(Color c);
public void Position(double x, double y);
public void Dessiner(DrawWindow dw);
}
```

A class can implement an interface.

```
public class DessinerDisque extends Disque implements DessinerForme {
private Color c;
private double x,y,r;
public DessinerDisque(double x, double y, double r) {super(c,x,y,r);}
public void Couleur(Color c)
  {this.c = c;}
public void Position(double x, double y)
 { this.x = x; this.y = y;}
public void Dessiner(DrawWindow dw)
  { dw.drawCircle(x,y,r,c);}
}
```

Example of utilization:

```
Forme[] formes= new Forme[2];                         // array for future instances
DessinerForme[] dessins = new DessinerForme[2];   // // array for future instances  of implementation
DessinerDisque dd = new DessinerDisque(2.5,5.0,1,green);
DessinerRectangle dr = new DessinerRectangle(5.2,8.4);
formes[0] = dd; dessin[0] = dd;
formes[1] = dr; dessin[2] = dr;
double surface_totale = 0.0;
for(int i=0;i<formes.length;i++) {
surface_totale += formes[i].surface();
DessinerForme[i].Position(i*20.0,i*5.0); DessinerForme[i].Dessiner(fenetre)

}
```

In the same way that a class can have sub-classes, an interface can have sub-interfaces. A sub-interface inherits all the abstract methods of one or more interfaces.

```
public interface Flexible extends Extensible, Rotative, Amovible {}
```

An interface extending multiple interfaces inherits all abstract methods defined in its super-interfaces and may define its own abstract methods. A class that implements such an interface must implement all methods found in all super-interfaces and methods of this interface.

Summary

In this chapter we presented a number of important concepts and mechanisms for object-oriented programming with Java language. We have studied the concept of class and object (instantiation), the concept of visibility of variables and methods, and the inheritance mechanism. Finally we have illustrated the abstract classes and interfaces ..