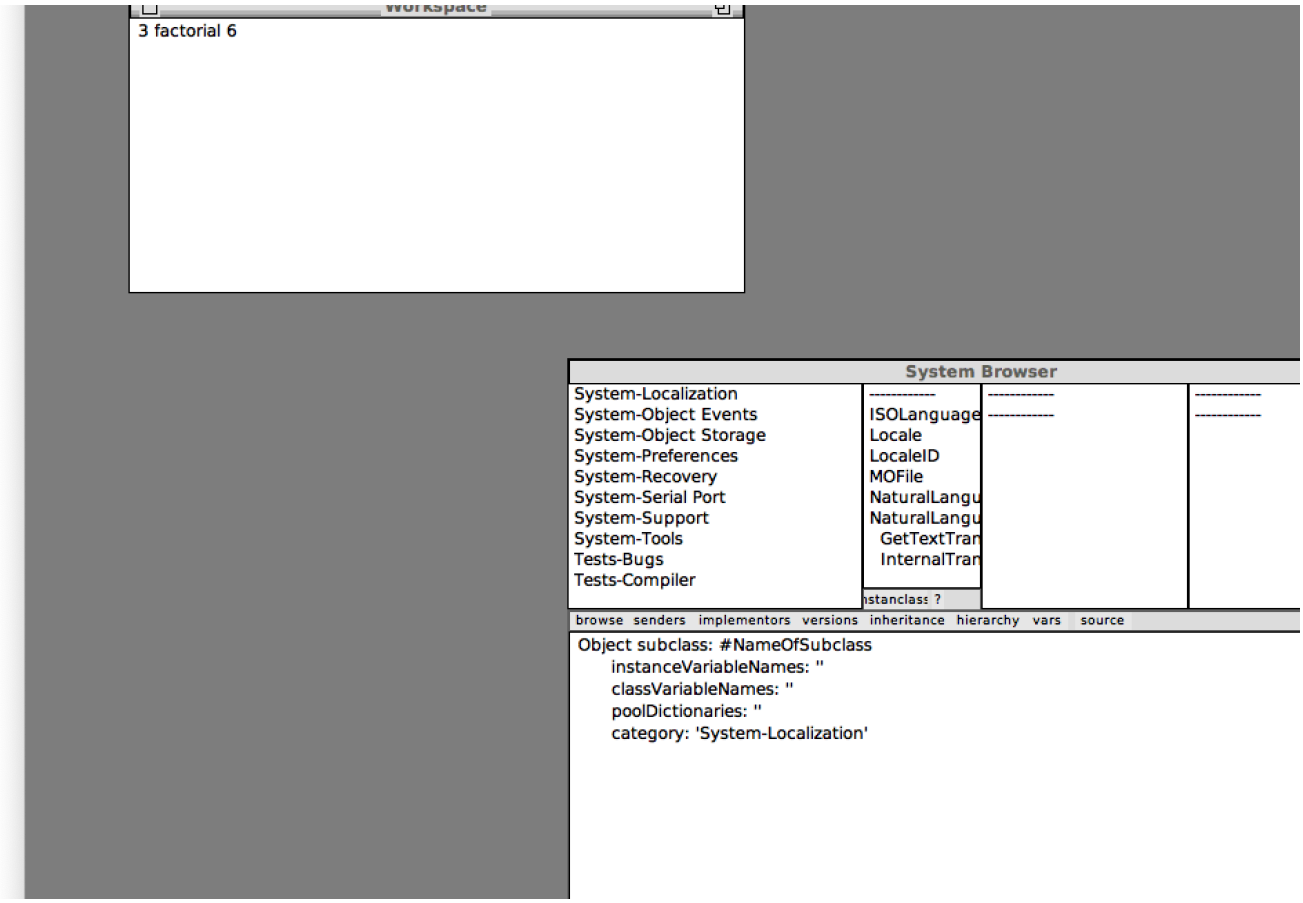
LA PROGRAMMATION OBJET

OOP IS PROGRAMMING BY SIMULATION

Object-oriented programming is most easily described as programming by simulation. The programming metaphor is based on personifying the physical or conceptual objects from some real-world domain into objects in the program domain; e.g., objects are clients in a business, foods in a produce store, or parts in a factory. We try to reincarnate objects from the problem domain into our computer models, giving the objects in our program the same characteristics and capabilities as their real-world counterparts. This process is often referred to as anthropomorphic programming or programming by personification.

The power of simulation as a programming metaphor can be seen from the success of the window-based user interfaces now common in personal workstations. The Apple Macintosh, for example, uses a desktop metaphor in which icons representing such.

Inside Smalltalk (Vol. 1)
WILF R. LALONDE
JOHN R. PUGH



Interface de programmation smalltalk

LES CLASSES ET LES OBJETS

Contrairement à la programmation procédurale on ne va pas identifier les fonctions mais les **objets** qui ferons l'objet de manipulations (calculs). Une fois identifiés les objets il faudra décrire leur caractéristiques et l'interrelation entre eux. Les objets sont des **instances** de la classe.

Une classe a 3 sortes de membres

fields (les données)
methods (les méthodes , fonctions)
constructors (un constructeur ou plusieurs constructeur qui spécifie comment les objets doivent être créés)

Avantage de l'utilisation des classes :

1) Encapsulation

Les Objets encapsulent les données et leur état ainsi que les opérations qui peuvent être exécutées sur l'état de ces données.

La seule façon d'accéder à l'état d'un objet est de lui envoyer un message qui déclenche l'exécution de l'une de ses méthodes.

2) Héritage

Chaque instance d'une classe d'objet hérite des caractéristiques (attributs et méthodes) de sa classe mais aussi d'une éventuelle super-classe.

Pour qu'une sous-classe hérite des champs et des méthodes d'une autre classe on utilise le mot clé : **extends**.

On dé finit une classe générique et on spécifie les particularités dans des sous-classes qui héritent de cette classe générique. Une sous-classe hérite de toutes les variables et méthodes qui sont soit public soit protected dans la super-classe.

```
public class Chat extends Animal {....}
```

Le constructeur d'une sous classes peut appeler le constructeur de sa super classe grâce à la méthode `super()`.

Cet appel doit obligatoirement être la première instruction du constructeur.

De la même façon on peut toujours appeler la méthode d'une super classe (qui aurait était surchargée dans une sous classe) en préfixant le nom de la méthode par super.

3) Polymorphisme

Surcharge des méthodes , le polymorphisme :

On nomme polymorphisme le fait de pouvoir appeler du même nom des méthodes différentes. A l'intérieur d'une même classe, il est possible de créer des méthodes ayant le même nom mais ayant des signatures différentes .

```
void vieillir(){ age++; }
void vieillir(int nb){ age+=nb;}
```

Quelques exemples à tester

Version1

```
class Hello1 {
        public static void main(String[] args) {
                    System.out.println("Hello World by String variable");
        }
}
```

```
(String[] args)
```

permet de rentrer des argument à l'appel de la fonction (args: le nom peut être différent)

Version2

```
class Hello2 {
        public static void main(String[] args) {
                String greetings = "Hello World by String variable";
                System.out.println(greetings);
        }
}
```

Ici je crée un objet String appelé greetings c'est une variable locale dans la méthode main. La méthode main est une méthode de l'objet principal de Java qui est Object.

Version3

```
class Hello3 {

    private static String greetings = "Hello World ";

    public static void main(String[] args) {
                System.out.println(greetings);
        }
}
```

Cette classe a 2 membres de classe (class members)

la variable String appelée greetings et une méthode name
cette variable est appelée un champ (field)

les fields et les methods sont généralement définies ainsi

modifier static *type name value*

modifier ; private, protected , public (ou absent)

static peut être absent

type : plusieurs milliers puisqu'en plus des primitives primaires ça peut être un objet existant dans Java ou un objet créé (ici String dans notre exemple)

En général les classes ont des champs, méthodes et constructeurs (fields, methods, constructor)

Version 4

```
class Hello4 {
        private String greetings = "Hello World with constructor ";

        public Hello4() {
             System.out.println(greetings);
        }

        public static void main(String[] args) {
             new Hello4();
        }
}
```

static permet de ne pas avoir à instancier un objet avec l'opérateur new , on l'exécute directement (exemple : main)

Une méthode static ne peut utiliser directement un champ ou une méthode non-static , dans les exemples précédents main() utilise directement le champ greetings.

Les constructeurs ne sont pas des méthodes mais agissent comme des méthodes static.

LA CLASSE ABSTRAITE

On peut désirer fournir une implémentation partielle d'une classe ou interdire son instantiation.

Le mécanisme disponible pour permettre ceci est de déclarer cette classe comme abstraite.

Le mot clé `abstract` permet définir une classe ou une méthode abstraite.

Ce comportement n'est utile que si la classe abstraite est une super classe.

Toute classe ayant une méthode abstraite devient automatiquement abstraite.

Il est interdit d'instancier un objet à partir d'une classe abstraire, plus aucun appel a new n'est possible.

LES INTERFACES

Le rôle d'une interface est de déclarer des comportements génériques qui seront partagés par plusieurs classes - sans créer de liens d'héritage entre elles.

C'est une réponse à l'impossibilité de l'héritage multiple en Java.

Une classe peut implementer autant d'interfaces qu'elle le désire

Une interface est de fait une classe abstraite car elle n'implémentent aucune des méthodes déclarées.

Les méthodes sont donc implicitement publiques et abstraites.

Une interface n'a pas de champs - uniquement des méthodes.

Exemple d'interface

```
interface Inflammable {
   void enflammer();
}

class Bois implements Inflammable {
 public void enflammer() {
   System.out.println("Je brule et fait des braises");
 }
}

class Dancefloor  implements Inflammable {
 public void enflammer() {
   System.out.println("♪ ♫ Youhouhou ♬ ♫ ");
 }
}

public class Main {
   static public void main(String[] args) {
      Inflammable[] tab = {new Bois(), new Dancefloor()};
      for(Inflammable i : tab)
       i.enflammer();
   }
}
```

Interêt : Si je suis le programmeur uniquement du programme principal quand je crée l'objet je ne dois pas créer spécifiquement chaque sous classe. Je crée uniquement l'objet Inflammable. interface permet ainsi le polymorphisme.

Exercice : ajouter une classe d'un objet qui sera inflammable.

```
interface Inflammable {
   void enflammer();
}
```

```
class Bois implements Inflammable {
 public void enflammer() {
   System.out.println("Je brule et fait des braises");
 }
}
class Dancefloor  implements Inflammable {
 public void enflammer() {
   System.out.println(« Je danse !!!");
 }
}

class Fusée  implements Inflammable {
 public void enflammer() {
   System.out.println(« explosion au décollage");
 }
}

public class Main {
   static public void main(String[] args) {
      Inflammable[] tab = {new Bois(), new Dancefloor(),new Fusée(),};

      for(Inflammable i : tab)
       i.enflammer();
   }
}
```

Thread

Java permet d'effectuer de la **programmation concurrentielle** , plusieurs process peuvent être exécutés en parallèle grâce à la classe Thread

Tester:

```
public  class Bateau  extends Thread {

   // surcharge de la méthode run() de la classe Thread
   public  void run() {

     int n =  0 ;
     while (n++ <  100) {
      System.out.println("Je vogue !") ;
       try {

         Thread.sleep(10) ;
       }  catch (InterruptedException e) {

          // gestion de l'erreur
       }
     }
   }
}
```

```
// dans une méthode main()
 // instanciation d'un objet de type Thread
Bateau b =  new Bateau() ;

 // lancement de ce thread par appel à sa méthode start()
b.start() ;

 // cette méthode rend immédiatement la main...
System.out.println("Thread lancé") ;

 // ... mais la méthode main() ne quitte pas immédiatement
```

La première façon de lancer un thread est la plus simple, mais elle n'est en général pas utilisable, du fait qu'une classe Java ne peut pas étendre deux classes à la fois.

Il existe donc une deuxième façon de faire, qui commence par implémenter l'interface Runnable. Cette interface n'expose qu'une unique méthode : run().

```
public class AutreBateau implements Runnable {

   // implémentation de la méthode run() de l'interface Runnable
   public void run() {

     int n = 0 ;
     while (n++ < 100) {
      System.out.println("Je vogue aussi !") ;
       try {

         Thread.sleep(10) ;
       } catch (InterruptedException e) {

          // gestion de l'erreur
       }
     }
   }
}


// dans une méthode main()
 // instanciantion d'un objet de type Runnable
AutreBateau autreBateau = new AutreBateau() ;

 // construction d'un Thread en passant cette instance de Runnable en paramètre
Thread thread = new Thread(autreBateau) ;

 // lancement de ce thread par appel à sa méthode start()
thread.start() ;

 // cette méthode rend immédiatement la main...
System.out.println("Thread lancé") ;

 // ... mais la méthode main() ne quitte immédiatement
```

TYPES PRIMITIFS

```
boolean      true ou false
char         caractère 16 bits Unicode
byte         entier 8 bits signés (-128 à 127)
short        entier 16 bits signé (-32768 à 32767)
int          entier 32 bits signés ( -2 147 483 648 à 2 147 483 648 )
long         entier 64 bits signés ( -9 223 372 036 854 775 808 à
-9 223 372 036 854 775 808)
float        nombre à virgule flottante 32-bits
double       nombre à virgule flottante 64-bits
```

OPERATEURS

Arithmétiques : +, −, ∗, /, %, + =, − =, ∗ =, / =, ++, −−,

Booléens : == , ! =, =<,=>, ||, &&, ? :.

Les structures de contrôle : if, for , while, switch

LA CLASSE STRING

Créer un String : String mot = "abc";

char data[] = {'a', 'b', 'c'};

String motNouveau = new String(data);

Tester l'égalité : mot.equals("bcd");

Attention au piège avec ==.

Récupérer un String à partir d'un int, d'un double

String nom = String.valueOf(num)

Obtenir des informations : length(), charAt()

LES CLASSES WRAPPER

Les types primitifs peuvent être encapsulés dans des classes :

Integer, Byte, Long, Double, Float, Character, Void.

Exemple :

int num=Integer.parseInt(mot);
double taille=Double.parserDouble(mot2);

LES TABLEAUX

Le type tableau : []

Déclarer un tableau : int []tableauInt;

Déclaration et allocation mémoire :

```
int []tableau=new int [10];
Animal []tableau = new Animal[MAX];
```

Accès aux cases du tableau : int num=tableau[2];

Taille du tableau : int taille =tableau.length

Saisie sur l'entrée standard.

```
class Saisie {
	public static void main(String[] args) {
		BufferedReader buff = new BufferedReader(new InputStreamReader(System.in));
		String chaine=buff.readLine();
		System.out.println("Tu as saisi:" +chaine);

		}
}
```

```
import java.io.* ;
class Saisie {
	public static void main(String[] args) {
	try {
		BufferedReader buff = new BufferedReader(new InputStreamReader(System.in));
		String chaine=buff.readLine();
		System.out.println("Tu as saisi:" +chaine);

	}
	catch(IOException e) {
		System.out.println(" impossible de travailler" +e);
		}
	}
```

`Saisir un entier au clavier`

```
int num = Integer.parseInt(chaine);
```

L'utilisation de static

Définition de variable de classe et non d'instance.

L'accès à cette variable se fait par le nom de la classe.

Exemple1 :

System.out

Exemple 2

```
class Animal {
        boolean vivant;
        private int age;
        private int matricule;
        static int nombre=0;

        public Animal() {  age =0;
                           vivant=true;
                           nombre++;
                           matricule=nombre; }
}
```

Une méthode peut également être qualifiée de **static**.

Exemple : main

Conséquence : toutes les méthodes appelées par une méthode static doivent aussi être static.

Une méthode static ne peut jamais adresser une variable d'instance.

L'utilisation de final

L'attribut final permet de spécifier qu'une variable ne pourra pas subir de modification - c.à.d une constante.

La valeur initiale de la variable devra être donnée lors de la déclaration.

Une méthode peut être qualifiée de final, dans ce cas elle ne pourra pas être redéfinie dans une sous-classe.

Une classe peut être qualifiée de final, dans ce cas elle ne pourra pas être héritée.

Permet de sécuriser une application.

Difference en equals et ==

Soit une classe Liste :

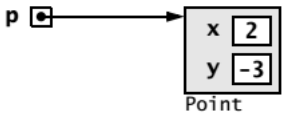
```
Class Liste {
        int [] liste = new int[20] ;
        int entree, sortie ;

        public void nouveau_client (int n)
        {
                if (entree < liste.length ) {
                        liste[entree++] = n ;
                } else {
                        System.out.println('File d'attente pleine');
                }

        }
}
```

```
Liste x ;                   // déclaration de la référence d'un objet de la classe Liste non créé
Liste y = new Liste() ;     // y référence l'objet créé
Liste z = y ;               // z = y ;
Liste w = y.clone() ;       /* w et y référencent 2 objets différents avec les mêmes valeurs
                               (clone() est une méthode prédéfinie de la class Object */
if (w.equals(y))            retourne vrai
if ( z == y )               retourne vrai
if ( w == y )               retourne faux
```

Exemple Point

```
public class Point {

// fields privés
private int x, y ;

// constructeur (même nom que la classe , à lappel de new Point(3,4) les filds de l'bjet
instancié seront x=3 et y=4
public Point(int X, int Y) {
        this.x=X;
        this.y=Y;
}

public boolean equals(Point p) {
        return( x == p.x && y == p.y );
}

public int getX() {
        return x;
}

public int getY() {
        return y;
}

@Override
public String toString(){
    return new String(x+"/"+y);
}

}
```

```
public class TestPoint {
        public static void main(String[] argv) {
                Point p = new Point(2,-3);
                System.out.println("p:" + p );
        }
}
```



Tester le programme sans la méthode surchargée **toString**

Explication pour *System.out.println("p:" + p);*

Le système veux afficher un string puisqu'il propose une concaténation or p est un objet Point et nom un objet String , java va donc chercher la méthode toString de la classe Point (surcharge de toString de java)

Java 5 -> GENERICS

- Stronger type checks at compile time.
  A Java compiler applies strong type checking to generic code and issues errors if the code violates type safety. Fixing compile-time errors is easier than fixing runtime errors, which can be difficult to find.

- Elimination of casts.
  The following code snippet without generics requires casting:

```
List list = new ArrayList();
list.add("hello");
String s = (String) list.get(0);
```

When re-written to use generics, the code does not require casting:

```
List<String> list = new ArrayList<String>();
list.add("hello");
String s = list.get(0);   // no cast
```

Dans java il existe des packages que l'on importe au besoin.
java.lang est chargé par défaut
pour utiliser par exemple la classe random() il faut importer le package java.util dans lequel il y a la classe Random()

Remarques sur les constructeurs:

Les constructeurs ont le même nom que la classe
Il n'ont pas de type de retour
Ils sont invoqués par l'opérateur **new**
Il est possible d'avoir plusieurs constructeurs s'ils ont une signature différente
Ils peuvent invoquer d'autres constructeurs avec les mots clés **this** et **super**
(à voir plus tard)

Objets et Références

```
public class TestPoint {
	public static void main(String[] argv) {
		Point p=null, q:null;
		Point p = new Point(2,-3);
		q = p;
		p = new Point(7,4);
		q = p;
		p = null;
		q = null;
	}
}
```

garbage collector automatique

Un des objectifs de la programmation objet est la réutilisation grâce à la COMPOSITION et l'HERITAGE.

La composition consiste à utilisé d'autres classes pour créer une autre classe