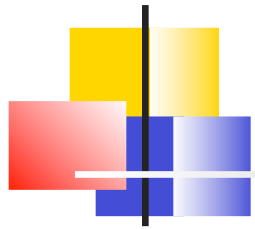


Java

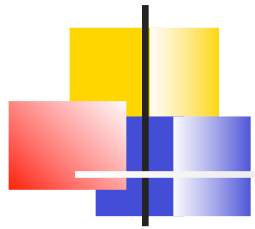
- Java et les Objets
- Les Tableaux
- Les Chaînes de Caractères
- Les classes « Enveloppe »
- Tour d'horizon des structures de contrôle
- API : Exemple



Java

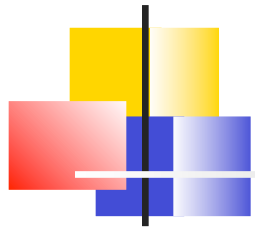
- Java et les Objets

- Classes, Méthodes, Constructeurs, Finaliseurs
- Surcharge
- Héritage, Redéfinition
- Polymorphisme et liaison différée, final
- Transtypage (up & down)
- Héritage et constructeur
- Contrôle d'accès
- Classes abstraites



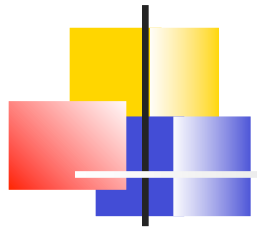
Java est orienté-objet

- Java reprend les meilleurs concepts et caractéristiques des langages objet, tels que **Eiffel**, **Smalltalk**, **Objective C**, et **C++**.
 - Java étend le modèle objet de C++ et en supprime les difficultés majeures.
 - A part les types de données primitifs, tout est objet en JAVA,
 - Les types primitifs pourront être encapsulés à l'intérieur d'objets.
 - + Autoboxing avec Java 1.5



La technologie objet : Rappel

- Pour être orienté-objet, un langage doit respecter au minimum 4 caractéristiques :
 - **L'encapsulation** : implémente le masquage d'informations et la modularité.
 - **Le polymorphisme** : le même message envoyé vers différents objets a un résultat dépendant de la nature de l'objet recevant le message.
 - **L'héritage** : on peut définir de nouvelles classes basées sur des classes existantes, pour obtenir du code réutilisable et de l'organisation.
 - **Liaisons dynamiques** : envoyer des messages à des objets sans connaître au moment du codage leur type spécifique.



Les bases des objets informatiques

- Un objet est défini par :
 - Son état par ses variables d'instance (VI).
 - Les VI sont privées à l'objet.
 - Son comportement par ses méthodes.
 - Les méthodes manipulent les VI pour créer de nouveaux états; Les méthodes d'un objet peuvent ainsi créer de nouveaux objets.
- Une Classe
 - Une classe est une construction logicielle qui définit les VI et les méthodes d'un objet.
 - On obtient des objets concrets en instanciant une classe définie au préalable.



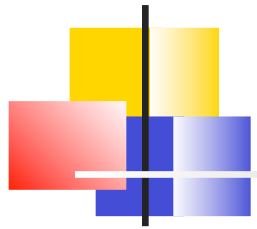
Déclaration de classe

- **Syntaxe**

```
[modificateur_de_classe] class ClassName
    [extends superClass]
    [implements interface [, ...]]
    {
        // Constructeurs
        // Méthodes et attributs
    }
```

- **Exemple de la classe Cercle :**

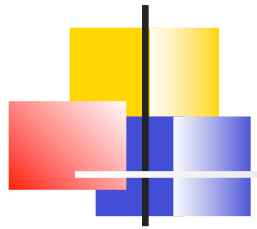
```
public class Cercle
{
    // Membres de la classe
}
```



Les membres de la classe

- Les attributs et les méthodes constituent les membres de la classe
- Syntaxe de déclaration :

```
class ClassName
{
    [modificateur(s)] type attribut_ou_methode;
}
```

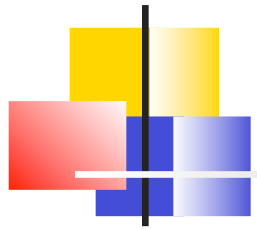


Les membres de la classe

```
public class Cercle
{
    // Déclaration et définition des attributs
    public double x,y; // Coordonnées du centre
    private double r;  // Rayon du cercle

    // Déclaration et définition d'un constructeur
    public Cercle(double rayon)
    { ... }

    // Déclaration et définition d'un méthode
    public double surface()
    { ... }
}
```

Les méthodes

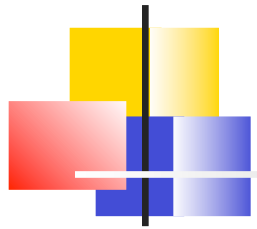
- La partie traitement des objets est contenue dans les méthodes

- Syntaxe de déclaration :

```
[modificateurs] typeRetour nomMethode (TypeArg arg,...)
{
    ...
}
```

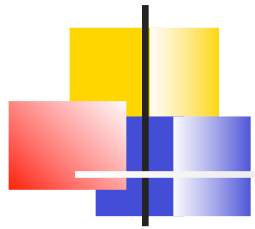
- Exemple dans Cercle

```
// Déclaration et définition d'une méthode
public double surface()
{
    return 3.14*r*r;
}
```



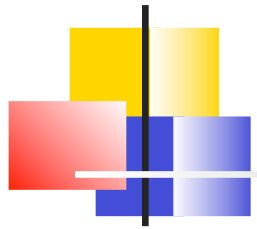
Instancier un objet en Java

- C'est utiliser le « moule Classe » pour obtenir un « exemplaire »
- Exemple :
 - Après avoir déclaré la classe Point,
 - `Point myPoint; // On peut créer un objet Point`
 - `myPoint = new Point();`
 - on accède aux variables de cet objet en se référant au nom de variable, associé au nom de l'objet:
 - `myPoint.x = 10.0;`
 - `myPoint.y = 25.7;`



Les constructeurs

- Le **constructeur** est la méthode appelé à la création (instanciation) de l'objet
 - portent le même nom que la classe, n'ont pas de type de retour.
 - Java fournit un constructeur par défaut qui attribue une valeur par défaut aux VI.
 - Si le programmeur définit un/des constructeurs :
 - Le constructeur par défaut disparaît.
 - Les constructeurs peuvent s'appeler entre eux.
 - En utilisant : `this(..)`
 - Obligatoirement au début du code du constructeur.
- 11 ■ Une seule fois.



Le désignateur `this`

- Le désignateur `this` permet de faire référence à l'objet courant (celui que l'on est en train de définir) ou de désigner ses attributs ou ses méthodes :

```
public class Cercle
{
    public double r;
    public Cercle(double r)
    {
        this.r = r;
    }
    public Cercle plusGrand(Cercle c)
    {
        if(c.r > r) return c; else return this;
    }
}
```

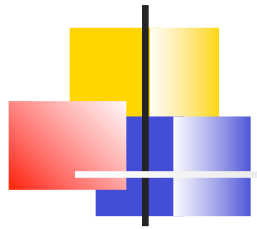
Flower.java

// Calling constructors with "this"

```
public class Flower {
    private int petalCount = 0;
    private String s = new String("null");
    Flower(int petals) {
        petalCount = petals;
        System.out.println(
            "Constructor w/ int arg only,
            petalCount= "+petalCount);
    }
    Flower(String ss) {
        System.out.println(
            "Constructor w/ String arg only, s=" +
            ss);
        s = ss;
    }

    Flower(String s, int petals) {
        this(petals);
        //!    this(s); // Can't call two!
        this.s = s; // Another use of "this"
        System.out.println("String & int args");
    }
}
```

```
Flower() {
    this("hi", 47);
    System.out.println(
        "default constructor (no args)");
}
void print() {
    //!    this(11); // Not inside non-
    constructor!
    System.out.println(
        "petalCount = " + petalCount + " s = " +
        s);
}
public static void main(String[] args) {
    Flower x = new Flower();
    x.print();
}
} ///:~
```



Les finaliseurs

- On peut déclarer un **finaliseur**, optionnel,
 - permettant des actions quand le **garbage collector** est sur le point de libérer un objet.

```
protected void finalize() {  
    File.Close();  
}
```
 - La méthode `finalize` ferme un fichier d'entrée-sortie qui était utilisé par l'objet.



Surcharge des méthodes

- En Java, plusieurs méthodes peuvent porter le même nom pourvu qu'elles puissent être distinguées à l'appel. On dit que la méthode est **surchargée** (ne pas confondre avec redéfinie).
- Lors de leur appel, Java distingue les méthodes surchargées en utilisant la liste de leur paramètres (qui doit donc être discriminante). **Le type du résultat n'est pas pris en compte.**
- Exemples de définitions :

```
void afficher(int i)                // afficher un entier
void afficher(long i)               // afficher un entier long
void afficher(String str)           // afficher une chaîne
void afficher(String str, int largeur); // sur une certaine largeur
void afficher(String str, int largeur, char mode); // mode = 'c', 'g', 'd'
void afficher(int nbFois, String str); // nbFois str
```



Surcharge des méthodes

- Exemples d'appels :

```
afficher(10);           // afficher(int)
afficher(10L);          // afficher(long)
afficher("Bonjour", 20, 'c'); // afficher(String, int, char)
afficher("Bonjour");    // afficher(String)
afficher(true)          // Erreur à la compilation
```

- La surcharge permet de simuler les paramètres par défaut.



Héritage

- Java ne dispose que de l'héritage simple : une classe ne peut dériver que d'une seule classe parente.

```
class Point extends Object {  
    protected double x; /* variable d'instance */  
    protected double y; /* variable d'instance */  
    Point() {             /* constructeur */  
        x = 0.0 ; y = 0.0;  
    }  
} // Fin de classe Point
```

```
class ThreePoint extends Point {  
    protected double z;  
    ThreePoint() {x = 0.0 ; y = 0.0 ; z = 0.0;}  
    ThreePoint(double x, double y, double z) {  
        this.x = x;this.y = y;this.z = z;  
    }  
} // Fin de classe ThreePoint
```



Redéfinition des méthodes

- Si une classe dérivée définit une méthode portant le même nom, attendant les mêmes paramètres et produisant le même type de résultat qu'une méthode de sa super-classe, on dit qu'elle **redéfinit** cette méthode.
- Une redéfinition est différente d'une surcharge : la surcharge consiste à définir plusieurs méthodes de même nom dans la même classe, mais avec des paramètres différents.
- Les méthodes de classe et les méthodes privées ne peuvent évidemment pas être redéfinies dans une classe dérivée.
- Une méthode définie avec le modificateur `final` ne pourra pas être redéfinie dans une classe dérivée.



Polymorphisme et liaison différée

- Un objet a un **type apparent** (ou statique) : celui qu'il a au moment de la compilation (donc celui de sa référence).
- Il a un **type réel** (ou dynamique) : celui qu'il a au moment de son utilisation.
- Quand une méthode est appliquée à un objet, c'est celle associée au type dynamique qui est choisie.
- Le compilateur ne connaît que le type statique, pour chaque méthode, il ajoute donc un code effectuant une recherche dynamique de méthode (**dynamic method lookup**) afin de réaliser une liaison différée (**late binding**) au moment de l'exécution.
- La recherche d'une méthode redéfinie est donc moins rapide qu'un appel direct...



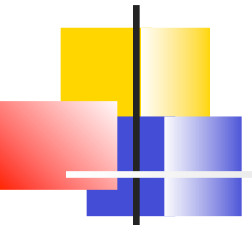
Résolution d'un appel polymorphe

■ Rappel

■ Soit le code :

```
Point p = new PointNomme("A", 1, 2);  
System.out.println(p);           // appel de toString() sur p  
System.out.println(p.getNom()); // Erreur : un Point n'a pas de getNom()
```

- à la compilation, il doit exister une méthode `toString()` et `getNom()` dans la classe `Point`, ou l'une de ses ancêtres (sinon : erreur de compilation).
- Lorsque l'interpréteur exécute le code, il recherche la méthode `toString()` à appeler : c'est la dernière partant du type apparent (`Point`) et allant vers le type dynamique (`PointNomme`).



Classes et méthodes finales

- Raisons de choisir une classe ou une méthode final :
 - **Efficacité** : Le traitement de la liaison dynamique est plus lourd de celui de la liaison statique, les méthodes « virtuelles » s'exécutent donc plus lentement. Le compilateur est incapable de produire une version inline d'une méthode qui n'est pas finale puisqu'elle pourrait être redéfinie. Par contre, il peut le faire pour une méthode `final`.
 - **Sécurité** : La liaison dynamique n'offre pas de contrôle sur ce qui se passera à l'exécution, à l'appel de la méthode. Une méthode `final` garantit que c'est elle qui sera appelée.



Transtypage : upcasting

- Soit le code suivant :

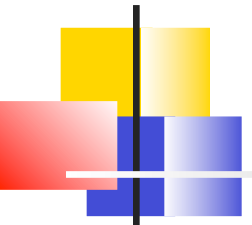
```
Point unPoint;  
PointNomme unPointNomme = new PointNomme("Point X", 10, 20);  
unPoint = unPointNomme;  
System.out.println("p : " + p); // toString() de PointNomme
```

- Le compilateur ne connaît que les types apparents, qui ici ne sont pas identiques : on affecte un `PointNomme` à un `Point`.
 - Cette affectation est autorisée car le compilateur sait qu'un objet de type `PointNomme` est un objet de type `Point` avec des caractéristiques supplémentaires. (Merci l'héritage)
- Le mécanisme permettant de convertir un objet d'une classe dérivée (`PointNomme`) en un objet d'une classe parente (`Point` ou `Object`) s'appelle **upcasting** (on « remonte » dans l'arbre d'héritage).



Transtypage : upcasting

- L'**upcasting** est toujours sûr car on va d'un type spécialisé vers un type plus général (on est sûr que l'objet récepteur contiendra les méthodes et attributs de l'objet de départ).
 - Pour cette raison, le compilateur réalise implicitement l'upcasting lorsque cela est nécessaire.
 - Cela signifie, notamment, que l'on pourra toujours affecter un objet d'une classe quelconque à une instance de la classe `Object`.
- A l'exécution, c'est le type effectif qui sera considéré (ici, `PointNomme`).



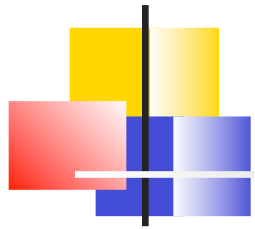
Transtypage : downcasting

- Soit le code suivant :

```
Point unPoint = new PointNomme("Point X", 10, 20);  
PointNomme unPointNomme;  
unPointNomme = unPoint;
```

- Ici, le compilateur interdira cette affectation car les deux types apparents sont différents (et un `Point` n'est pas nécessairement un `PointNomme`). Or, l'exécution, elle, aurait un sens car ici, le point est un point nommé...
- Pour résoudre ce problème, on utilise le **downcasting** : on indique explicitement au compilateur que l'on transtype un objet d'une classe parente en un objet d'une classe dérivée :

```
unPointNomme = (PointNomme)unPoint;
```

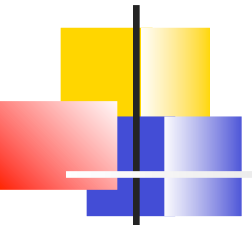
Transtypage : downcasting

- Le downcasting n'est pas sûr : l'objet récepteur fournit éventuellement plus de méthodes que l'objet affecté...

Considérons l'exemple suivant :

```
Point unPoint = new Point(10, 20);  
PointNomme unPointNomme;  
unPointNomme = (PointNomme)unPoint;  
System.out.println(unPointNomme.getNom());
```

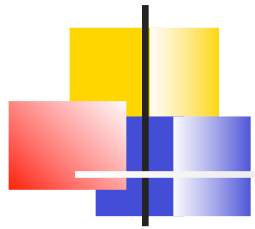
- Un appel à `unPointNomme.getNom()` n'aurait aucun sens... Le compilateur ne se plaindra pas de l'affectation, ni d'un appel à `unPointNomme.getNom()` (à cause du downcasting explicite) mais l'exécution échouera.



Transtypage : downcasting

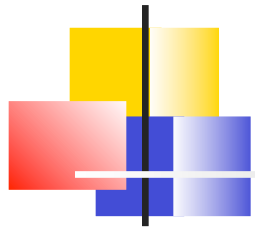
- Afin d'éviter ce type de problème lors de l'exécution, il est fortement conseillé de vérifier le type dynamique de l'objet avant d'appeler une méthode définie dans une classe dérivée :

```
Point unPoint = new Point(10, 20);  
PointNomme unPointNomme;  
if (unPoint instanceof PointNomme) { // donc pas exécuté ici...  
    unPointNomme = (PointNomme)unPoint;  
    System.out.println(unPointNomme.getNom());  
}
```



Transtypage : downcasting

- En pratique, il est préférable **d'éviter le downcasting**. Java ne disposant de la généricité que depuis la version 1.5, pour les versions antérieures le downcasting reste nécessaire pour la simuler...
- Cas typique : les types conteneurs de la bibliothèque Java (`List`, par exemple) manipulent uniquement des instances de la classe `Object` :
 - On peut donc toujours affecter une instance d'une classe quelconque comme élément d'un conteneur (**upcasting**).
 - Le transtypage explicite (**downcasting**) est nécessaire lorsque l'on consulte un élément car les méthodes d'accès de ces classes renvoient toujours un `Object`.



Héritage et constructeur

- Les constructeurs des classes mères sont appelés implicitement
 - Constructeurs sans arguments
- Les constructeurs à argument doivent être appelés explicitement en début de code du constructeur de la classe dérivée.



Cartoon.java

// Constructor calls during inheritance

```
class Art {
    Art() {
        System.out.println("Art constructor");
    }
}

class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing constructor");
    }
}

public class Cartoon extends Drawing {
    Cartoon() {
        System.out.println("Cartoon constructor");
    }

    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
}
```



Cartoon.java

// Constructor calls during inheritance

```
class Art {
    Art() {
        System.out.println("Art constructor");
    }
}

class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing constructor");
    }
}

public class Cartoon extends Drawing {
    Cartoon() {
        System.out.println("Cartoon constructor");
    }

    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
}
```

/* The output for this program shows the automatic calls:



Cartoon.java

// Constructor calls during inheritance

```
class Art {
    Art() {
        System.out.println("Art constructor");
    }
}

class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing constructor");
    }
}

public class Cartoon extends Drawing {
    Cartoon() {
        System.out.println("Cartoon constructor");
    }

    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
}
```

/* The output for this
program shows the
automatic calls:

Art constructor



Cartoon.java

// Constructor calls during inheritance

```
class Art {
    Art() {
        System.out.println("Art constructor");
    }
}

class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing constructor");
    }
}

public class Cartoon extends Drawing {
    Cartoon() {
        System.out.println("Cartoon constructor");
    }

    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
}
```

/* The output for this program shows the automatic calls:

Art constructor

Drawing constructor



Cartoon.java

// Constructor calls during inheritance

```
class Art {
    Art() {
        System.out.println("Art constructor");
    }
}

class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing constructor");
    }
}

public class Cartoon extends Drawing {
    Cartoon() {
        System.out.println("Cartoon constructor");
    }

    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
}
```

/* The output for this
program shows the
automatic calls:

Art constructor
Drawing constructor
Cartoon constructor



Cartoon.java

// Constructor calls during inheritance

```
class Art {
    Art() {
        System.out.println("Art constructor");
    }
}

class Drawing extends Art {
    Drawing() {
        System.out.println("Drawing constructor");
    }
}

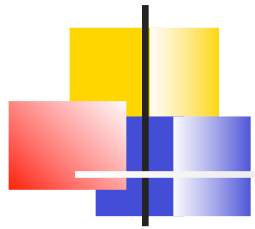
public class Cartoon extends Drawing {
    Cartoon() {
        System.out.println("Cartoon constructor");
    }

    public static void main(String[] args) {
        Cartoon x = new Cartoon();
    }
}
```

/* The output for this
program shows the
automatic calls:

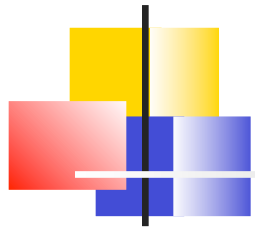
Art constructor
Drawing constructor
Cartoon constructor

*/



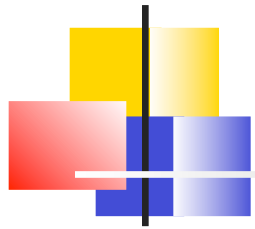
Les contrôles d'accès

- Quand on déclare une nouvelle classe en Java, on peut indiquer les niveaux d'accès permis à ses VI et à ses méthodes :
 - aucun attribut : accessibles par les classes qui font partie du même package, inaccessibles par les autres.
 - **public** : accessibles par toutes les classes
 - **protected** : accessibles par toutes les classes dérivées, et les classes du même package, inaccessibles par les autres
 - **private** : inaccessibles par toutes les classes
- L'attribut **protected** permet de rendre accessibles certains membres pour la conception d'une classe dérivée



Les contrôles accès

- Les VI et les méthodes `protected` sont seulement accessibles depuis les sous-classes de cette classe.
- Les méthodes et les VI `private` sont seulement accessibles de l'intérieur de la classe où elles sont déclarées
 - elles ne sont pas accessibles aux sous-classes de cette classe.



Les variables et méthodes de classe

■ Les Variables de classes

- Une variable de classe est «**locale**» à la classe elle-même.
- C'est une variable partagée par tout objet instance de la classe qui la déclare
- Pour déclarer des variables de classe et des méthodes de classe, il faut les déclarer en `static`.

■ Les Méthodes de classe

- Ce sont des méthodes communes à une classe entière.
- Ne peuvent opérer que sur les V de Classe.
- Ne peuvent pas invoquer des méthodes d'instance.
- Il faut les déclarer `static` (idem VC).



Demostatic

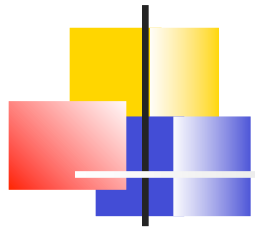
```
public class Demostatic {  
    static public void main (String [] arg) {  
        Partageuse p1=new Partageuse () ;  
        Partageuse p2=new Partageuse () ;  
        p1.modifie();  
        System.out.println("p1 : " + p1.partage+ " " + p1.nonPartage) ;  
        System.out.println("p2 : " + p2.partage+ " " + p2.nonPartage) ;  
    } ;  
}  
  
class Partageuse {  
    static int partage = 2;  
    int nonPartage = 2 ;  
    void modifie () {  
        partage=3 ;  
        nonPartage=3 ;  
    } ;  
}  
  
// p1 : 3 3  
// p2 : 3 2
```



Demostatic

```
public class Demostatic {  
    static public void main (String [] arg) {  
        Partageuse p1=new Partageuse () ;  
        Partageuse p2=new Partageuse () ;  
        p1.modifie();  
        System.out.println("p1 : " + p1.partage+ " " + p1.nonPartage) ;  
        System.out.println("p2 : " + p2.partage+ " " + p2.nonPartage) ;  
    } ;  
}  
  
class Partageuse {  
    static int partage = 2;  
    int nonPartage = 2 ;  
    void modifie () {  
        partage=3 ;  
        nonPartage=3 ;  
    } ;  
}  
  
// p1 : 3 3  
// p2 : 3 2
```

→ La classe Partageuse définit
une variable de classe **partage**



Les classes abstraites

- Une classe **abstraite** est une classe dans laquelle on peut définir des méthodes qui ne sont pas implémentées par la classe.
 - la classe abstraite définit un état générique et un comportement

générique

```
abstract class A {  
    ...  
    abstract void P();  
    ...  
    void Q() {...}  
}
```

```
class B extends A {  
    ...  
    void P() {...}  
    ...  
    void Q() {...}  
}
```

- Il est impossible d'instancier une classe abstraite.



Initialiseurs

- Java permet de définir des blocs d'initialisations
 - Pour les variables de classes
 - Une classe peut avoir un nombre quelconque d'initialiseur statique.
 - Le corps de chaque bloc est incorporé dans la méthode d'initialisation de la classe avec les initialisations statiques des champs
 - Pour les variables d'instances
 - Une classe peut avoir un nombre quelconque d'initialiseur d'instance.
 - Le corps de chaque bloc est inséré au début de chaque constructeur de la classe.
- En pratique c'est assez rarement utilisé.
 - Utiles parce qu'ils placent le code d'initialisation à côté du champ plutôt que de le séparer dans un constructeur.
 - Leur utilité principale est dans l'élaboration des classes internes anonymes dont nous parlerons un peu plus tard.



Initialiseur statique

```
// ExplicitStatic.java Initialisateur Static.  
class Cup { Cup(int marker) {System.out.println("Cup(" + marker + ")");}  
    void f(int marker) {System.out.println("f(" + marker + ")");}  
}  
class Cups {  
    static Cup c1;  
    static Cup c2;  
    static {    c1 = new Cup(1); c2 = new Cup(2); }  
    Cups() { System.out.println("Cups()"); }  
}
```

```
public class ExplicitStatic {  
    public static void main(String args[]) {  
        System.out.println("Inside main()");  
        Cups.c1.f(99);  
        // (1)  
    }  
    static Cups x = new Cups();  
    // (2)  
    static Cups y = new Cups();  
    // (2)  
}
```

Résultat

```
Cup(1)  
Cup(2)  
Cups()  
Cups()  
Inside main()  
f(99)
```



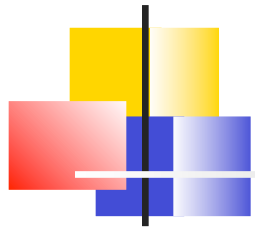
Initialiseur d'instance

```
class Mug {  
    Mug(int marker) { System.out.println("Mug(" + marker + ")");}  
    void f(int marker) {System.out.println("f(" + marker + ")"); }  
}
```

```
public class Mugs {  
    Mug c1; Mug c2;  
    { c1 = new Mug(1); c2 = new Mug(2);  
      System.out.println("init c1,c2"); }  
    Mugs() { System.out.println("Mugs()"); }  
    public static void main(String args[]) {  
        System.out.println("main()"); Mugs x = new Mugs();  
    }  
}
```

Résultat

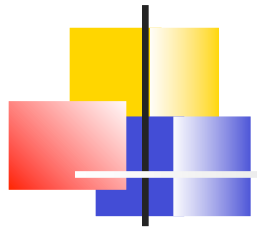
```
main()  
Mug(1)  
Mug(2)  
}  
init c1,c2  
Mugs()
```



Java

- Java et les Objets
- **Les Tableaux**
- Les Chaînes de Caractères
- Les classes « Enveloppe »
- Tour d'horizon des structures de contrôle
- API : Exemple

« Ecrire une fois, exécuter partout »



Les tableaux

- Un tableau est une collection de variables du même type, organisées séquentiellement et accessible au moyen d'un indice entier
- Les tableaux sont des objets
- Les tableaux prennent la forme d'une structure munie d'une borne inférieure et supérieure
 - pas uniquement représenté par une suite d'emplacements mémoire référencés par un pointeur comme en C



Déclaration

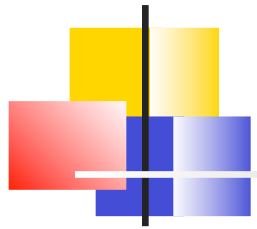
- Tableaux unidimensionnels :

- deux syntaxes :

- `type nomDuTableau[];`
 - `type[] nomDuTableau;`

- exemples :

- `int monBeauTableau[];`
 - `int[] monBeauTableau;`



Déclaration

- Tableaux multidimensionnels :

- deux syntaxes :

- `type nomDuTableau[]+;`
 - `type[]+ nomDuTableau;`

- exemples :

- `char c[][];`
 - `Color rgbCube[][][];`
 - `Color[][][] rgbCube`



Création

- Les tableaux ne sont pas contraints par des bornes au moment de leur déclaration
 - on déclare uniquement une référence
 - les dimensions seront fixées lors de leur création
- La création d'un tableau fait appel à l'opérateur `new`, car un tableau est un objet
- Les dimensions du tableau sont fixées à la création
 - `int[] monTableau = new int[100];`
 - `char c[][] = new char[10][12];`



Création et initialisation

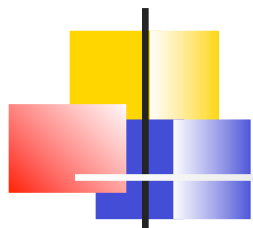
- Seule la première dimension du tableau doit impérativement être fixée à la création :
 - `char c[][] = new char[10][];`
- Les autres dimensions peuvent être fixées ultérieurement
 - `c[0] = new char[24];`
- Initialisation des objets contenus :

```
MaClasse objets[] = new MaClasse[MAX];
for(int i=0; i<objets.length; i++)
    objets[i] = new MaClasse(...);
```



Utilisation

- Les indices des tableaux commencent à 0
- L'attribut `length` (entier) donne la dimension du tableau fixée à l'initialisation
 - indice maximum du tableau = `tableau.length - 1`
- Lors de l'accès à l'un des éléments à l'aide d'un indice, l'appartenance de cet indice à l'intervalle spécifié par les bornes est vérifié
 - lève une `ArrayIndexOutOfBoundsException` si hors limites
 - évite des erreurs de programmation comme en C



Java

- Java et les Objets
- Les Tableaux
- **Les Chaînes de Caractères**
- Les classes « Enveloppe »
- Tour d'horizon des structures de contrôle
- API : Exemple

« Ecrire une fois, exécuter partout »



Les chaînes de caractères

- En C :

- calcul des longueurs de chaînes
- gestion de la fin de chaîne (code *ASCII* 0)
- pas de vérification de débordements

- En Java :

- 2 classes fournies par le JDK : `String` **et** `StringBuffer`
- pas de marqueur de fin de chaîne
- pas de calcul de longueur lors de la création
- vérification des débordements



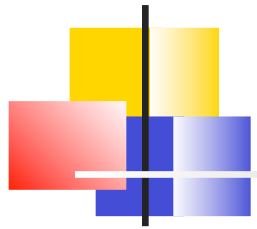
Introduction

- En Java, les chaînes de caractères sont des objets.
- Deux classes sont fournies pour les représentées :
 - la classe `String` offre beaucoup de fonctionnalités mais ne permet pas de modifier une chaîne.
 - la classe `StringBuffer` autorise la modification de la chaîne



Opération de base sur les chaînes

- Des chaînes peuvent être créées implicitement :
 - en utilisant une chaîne quotée : `"bonjour"`
 - en utilisant les opérateur `+` et `+=` sur deux objets `String` pour en créer un nouveau
- Il est également possible de construire explicitement des objet `String` en utilisant l'opérateur `new` :
 - `public String()` : construit un nouvel objet `String` qui a pour valeur `""`
 - `public String(String value)` : construit un nouvel objet `String` qui est une copie de la valeur de l'objet `String` donné
 - ...



Opération de base sur les chaînes

- Deux opérations élémentaires peuvent être réalisées aussi bien sur les `String` que sur les `StringBuffer` :
 - la méthode `length` retourne le nombre de caractère dans la chaîne
 - la méthode `charAt` retourne le caractère situé à la position donnée



Opération de base sur les chaînes

- Il existe aussi des méthodes simples pour trouver la première ou la dernière occurrence d'un caractère particulier ou d'une sous-chaîne dans une chaîne :
 - `indexOf(char ch)` : première position de `ch`
 - `indexOf(char ch, int start)` : première position de `ch` \geq `start`
 - `indexOf(String str)` : première position de `str`
 - `indexOf(String, int start)` : première position de `str` \geq `start`
 - `lastIndexOf(char ch)` : dernière position de `ch`
 - `lastIndexOf(char ch, int start)` : dernière position de `ch` \leq `start`
 - `lastIndexOf(String str)` : dernière position de `str`
 - `lastIndexOf(String str, int start)` : dernière position de



Comparaison de chaînes

- La classe `String` supporte plusieurs méthodes permettant de comparer des chaînes et des parties de chaînes.
- Ces méthodes travaillent selon la valeur des caractères Unicode, un tri placera "acz" avant "aça" car 'c' et 'ç' sont différents (respectivement `\u0063` et `\u00e7`)



Comparaison de chaînes

- La méthode `boolean equals(Object)` retourne `true` si le contenu de l'objet `String` passé en argument est le même que le contenu de l'objet sur lequel s'applique la méthode.
- Une variante `equalsIgnoreCase` réalise le même test sans tenir compte de la casse (majuscule/minuscule)



Comparaison de chaînes

- La méthode `int compareTo(Object)` retourne :
 - un entier inférieur à 0 si la chaîne sur laquelle elle est invoquée est plus petite que celle passée en argument
 - un entier égal à 0 si les deux chaînes sont identiques
 - un entier supérieur à 0 si la chaîne sur laquelle elle est invoquée est plus grande que celle passée en argument
- L'ordre utilisé est celui des caractères Unicode.



Comparaison de chaînes

- Il est également possible de réaliser des tests de comparaison sur des région de chaînes.

```
boolean regionMatches(int start, String other,  
                      int ostart, int len)
```

- Retourne `true` si la région donnée de cet objet `String` correspond à la région donnée de la chaîne `other`.
- La comparaison démarre dans cette chaîne à la position `start`, et dans l'autre chaîne à la position `ostart`.
- Seuls les `len` premiers caractères sont comparés



Comparaison de chaînes

- Il est possible de réaliser des comparaison simples de région de chaînes en testant uniquement le début ou la fin de chaîne.
 - `boolean startsWith(String prefix)` retourne true si cet objet `String` commence par le préfixe donné
 - `boolean endsWith(String suffix)` retourne true si cet objet `String` termine par le suffixe donné



Création de chaînes

- `String substring(int beginIndex)` **retourne la sous-chaîne commençant à l'indice** `beginIndex`
- `String substring(int begIndex, int endIndex)` **retourne la sous-chaîne commençant à l'indice** `begIndex` **et terminant à l'index** `endIndex`
- `String replace(char oldChar, char newChar)` **retourne un nouvel objet** `String` **où toute les occurrences de** `oldChar` **ont été remplacées par le caractère** `newChar`



Création de chaînes

- `String toLowerCase()` retourne un nouvel objet `String` où chaque caractère a été converti en son équivalent en minuscule s'il en a un.
- `String toUpperCase()` retourne un nouvel objet `String` où chaque caractère a été converti en son équivalent en majuscule s'il en a un.
- `String trim()` retourne un nouvel objet `String` où on a supprimé tous les caractères d'espacement du début et de la fin.



Conversion de chaînes

- Il est souvent nécessaire de convertir une chaîne de caractère en et depuis quelque chose d'autre, comme des entiers ou des booléens.
- Conversion d'un type primitif vers une `String`, méthodes `static` appartenant à la classe `String` :
 - `String valueOf(boolean)`
 - `String valueOf(int)`
 - `String valueOf(long)`
 - `String valueOf(float)`
 - `String valueOf(double)`

D'autres fonctions de conversion sont également disponibles dans la classe `primitive`, notamment la méthode `toString()` qui permet de convertir en `String` les valeurs de tous les types primitifs.



Conversion de chaînes

- Conversion d'une `String` vers un type primitif :
 - méthodes statiques :
 - `Integer` : `Integer.parseInt(String)` ;
 - `Long` : `Long.parseLong(String)` ;
 - Création d'un objet temporaire :
 - `Boolean` : `new Boolean(String).booleanValue()` ;
 - `Float` : `new Float(String).floatValue()` ;
 - `Double` : `new Double(String).doubleValue()` ;
- Les conversions en `byte` et `short` sont réalisées en utilisant la classe `Integer`



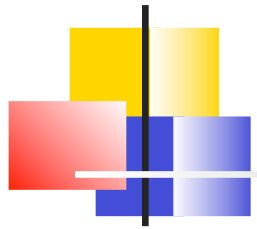
Chaînes et tableaux de `char`

- Le contenu des objets `String` est immuable. Il est parfois utile d'obtenir le contenu d'une chaîne sous forme de tableau de caractères pour le modifier :
 - récupérer le contenu d'un objet `String` sous forme de tableau de caractères
 - traiter (éventuellement modifier) le tableau de caractères
 - recréer un nouvel objet `String` à partir du tableau de caractères



Chaînes et tableaux de `char`

- La méthode `char[] toCharArray()` de la classe `String` retourne un tableau de `char` contenant le contenu de l'objet `String`.
- Le constructeur `String(char[])` permet de construire un objet `String` à partir d'un tableau de `char`.



La classe `StringBuffer`

- La classe `StringBuffer` offre la possibilité de modifier son contenu (i.e la chaîne de caractères)
- Ceci évite de nombreuses créations d'objets `String`
- `StringBuffer` et `String` sont deux classes indépendantes qui toutes deux étendent `Object`
- Constructeurs de la classe `StringBuffer` :
 - `StringBuffer()`
 - `StringBuffer(String)`



Modifier le buffer

- Il y a plusieurs façons de modifier le contenu du buffer d'un `StringBuffer`.
- La concaténation :
 - `StringBuffer append(String str)` : ajoute les caractères de la chaîne `str` à la fin du contenu de cet objet `StringBuffer`. Si `str` est `null`, ajoute les caractères « null »
 - `StringBuffer append(char[] str)`
 - `StringBuffer append(char[] str, int offset, int len)`
 - `StringBuffer append(boolean b)`
 - `StringBuffer append(int i)`
 - `StringBuffer append(char c)`
 - `StringBuffer append(float f)`



Modifier le buffer

- L'insertion :

- `StringBuffer insert(int offset, String str)` : insère la chaîne `str` dans le contenu de l'objet `StringBuffer` à la position `offset`.
- `StringBuffer insert(int offset, char [] str)`
- `StringBuffer insert(int offset, int i)`
- `StringBuffer insert(int offset, float f)`
- `StringBuffer insert(int offset, boolean b)`
- `StringBuffer insert(int offset, char c)`



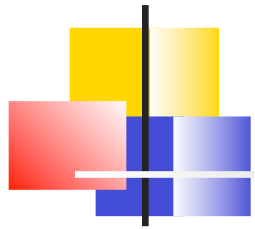
Modifier le buffer

- **Suppression de caractères :**
 - `Stringbuffer delete(int debut, int fin)` : supprime les caractères du buffer compris entre les indices debut et fin
 - `StringBuffer deleteChar(int i)` : supprime le caractère du buffer situé à l'indice i
- **Affectation d'un caractère particulier :**
 - `void setCharAt(int index, char c)` : remplace le caractère situé à l'index i dans le contenu de l'objet `StringBuffer` par le caractère c



Extraction de données

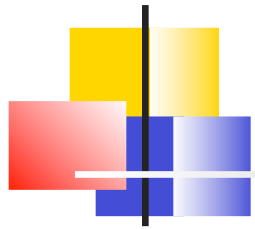
- Pour récupérer un objet `String` à partir d'un objet `StringBuffer` il faut invoquer la méthode `toString()` ou l'une des deux méthodes suivantes :
 - `String substring(int start)`
 - `String substring(int start, int end)`
- `char charAt(int i)`
retourne le caractère situé à l'index `i`



Java

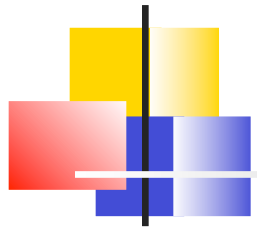
- Java et les Objets
- Les Tableaux
- Les Chaînes de Caractères
- **Les classes « Enveloppe »**
- Tour d'horizon des structures de contrôle
- API : Exemple

« Ecrire une fois, exécuter partout »



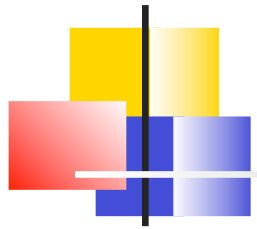
Introduction

- Il existe des types primitifs qui ne sont pas des objets : `char`, `int`, `float`, `boolean`, *etc.*
- A chacun de ces types correspond une classe : `Character`, `Integer`, `Float`, `Boolean`, *etc.*
- Chaque classe permet d'instancier un objet représentant une valeur d'un type primitif.
- Ces classes offrent, à travers leurs méthodes, un ensemble d'opérations usuelles.
- La valeur des objets créé à partir de ces classes est immuable (d'où l'intérêt d'utiliser les types primitifs).



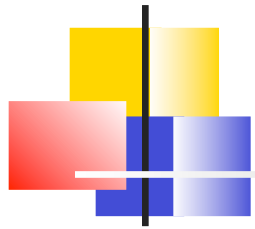
La classe Character

- **Constructeur :**
 - `Character(char value)`
- **Méthodes :**
 - `char charValue()`
- **Méthodes statiques :**
 - `boolean isLowerCase(char ch)`
 - `boolean isUpperCase(char ch)`
 - `boolean isDigit(char ch)`
 - `boolean isLetter(char ch)`
 - `char toLowerCase(char ch)`
 - `char toUpperCase(char ch)`



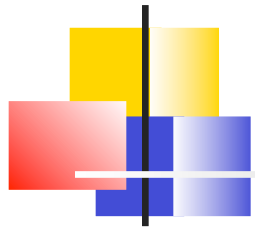
La classe Boolean

- La classe `Boolean` représente un type booléen sous forme de classe.
- Par convention, une chaîne de caractère « `true` » représente la valeur `true` quelque soit la casse. Toute autre chaîne a pour valeur `false`.
- La classe `Boolean` propose deux constructeurs :
 - `Boolean(boolean b)`
 - `Boolean(String str)`
- Peu de méthodes :
 - `boolean booleanValue()`
 - `static Boolean valueOf(String str)`



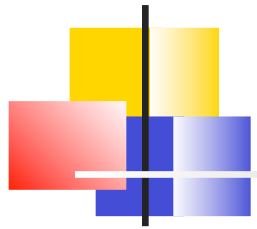
Les classes numériques

- Les classes `Byte`, `Short`, `Integer`, `Long`, `Float` et `Double` dérivent toutes d'une même classe mère : la classe `Number`.
- La classe `Number` dispose de méthodes (redéfinies au niveau de chaque sous-classe) permettant de convertir l'objet `Number` en un type numérique :
 - `byte byteValue()`
 - `double doubleValue()`
 - `float floatValue()`
 - `int intValue()`
 - `longValue()`
 - 71 ■ `shortValue()`



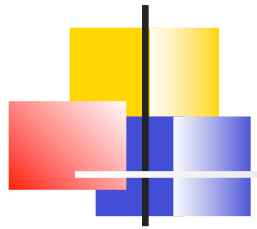
Les classes numériques

- Chaque classe représentant un type numérique dispose des méthodes suivantes :
 - une méthode statique `toString(type)` retournant un objet `String` représentant la valeur de type primitif **type** passé en argument
 - une méthode statique `valueOf(String)` retournant un objet du type numérique décrit par la classe et ayant la valeur décrite par la chaîne passée en argument.



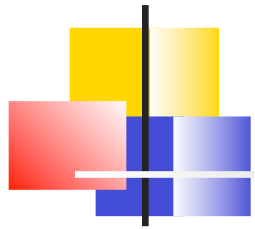
La classe Integer

- `public static int parseInt(String str)`
renvoie la valeur de l'entier décrit par `str`
- `public static int parseInt(String str, int radix)`
renvoie la valeur de l'entier décrit par `str` en base `radix`
- `public static Integer valueOf(String s, int radix)`
renvoie un objet `Integer` contenant la valeur de l'entier décrit par `str` en base `radix`
- `public static String toString(int i, int radix)`
renvoie un objet `String` représentant `i` dans la base `radix`



La classe Long

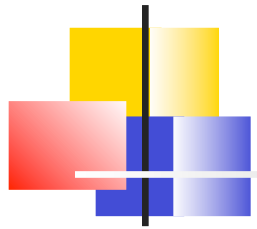
- `public static long parseLong(String str)`
renvoie la valeur du long décrit par `str`
- `public static long parseLong(String str, int radix)`
renvoie la valeur du long décrit par `str` en base `radix`
- `public static Long valueOf(String s, int radix)`
renvoie un objet `Long` contenant la valeur du long décrit par `str` en base `radix`
- `public static String toString(long l, int radix)`
renvoie un objet `String` représentant `l` dans la base `radix`



Java

- Java et les Objets
- Les Tableaux
- Les Chaînes de Caractères
- Les classes « Enveloppe »
- Tour d'horizon des structures de contrôle
- API : Exemple

« Ecrire une fois, exécuter partout »



Les structures de contrôle

- Les instructions (<chaîne de caractères>;)
 - sont exécutées dans l'ordre d'apparition
- Essentiellement les mêmes qu'en C :
 - branchements conditionnels
 - `if ... else`
 - `switch`
 - exécutions itératives :
 - `while`
 - `do ... while`
 - `for`
 - sortie, reprise de boucle : `break` **et** `continue`
 - retour de méthode : `return`



if ... else

- **Syntaxe**

```
if (expression_booléenne)
{
    instructions
}
else
{
    instructions
}
```

- **le bloc else est optionnel**



switch

■ Syntaxe

```
switch (expression)
{
    case expr1:
        instructions;
        [break;]
    ...
    case exprN:
        instructions;
        [break;]
    default:
        instructions;
```

Les différents cas
sont des points
d'entrée

- **expression doit retourner une valeur du type :**
`int, boolean` **ou** `char`



for

■ Syntaxe

```
for(initialisation; test; incrémentation)
{
    instructions
}
```

- `initialisation` est réalisée la première fois
- `test` est la condition d'entrée dans la boucle (testée à chaque rebouclage)
- `incrémentation` est réalisée à chaque rebouclage, avant le `test` et sera donc réalisée même quand le `test` est faux



while **et** do ... while

- **Syntaxe while**

```
while (expression_booléenne)
{
    instructions;
}
```

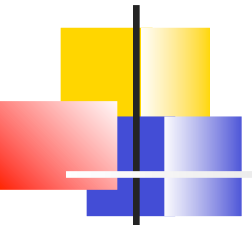
- **Syntaxe do**

```
do {
    instructions;
} while (expression_booléenne)
```



break

- L'instruction `break` permet de sortir de n'importe quel bloc
- Elle est généralement utilisée pour sortir d'une boucle
 - Cette instruction peut-être étiquetée :
`break nomEtiquette;`
 - ou non :
`break;`



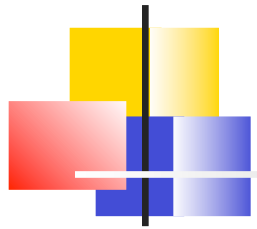
continue

- L'instruction `continue` provoque un rebouclage immédiat
- Elle est généralement utilisée à l'intérieur d'un `for`,
`while` ou `do ... while`
 - Cette instruction peut-être étiquetée :
`continue nomEtiquette;`
 - ou non :
`continue;`



return

- L'instruction `return` termine l'exécution d'une méthode et revient à l'appelant
- Si la méthode a un type de retour, le `return` doit être suivi d'une expression renvoyant une valeur du type correspondant :
 - `return x;` ou `return 3*x;`
- Si la méthode ne retourne aucune valeur, le `return` est utilisé seul :
 - `return;`



Java

- Java et les Objets
- Les Tableaux
- Les Chaînes de Caractères
- Les classes « Enveloppe »
- Tour d'horizon des structures de contrôle
- **API : Exemple**

« Ecrire une fois, exécuter partout »



Des Librairies Standard

- Première approche
 - `java.lang`
 - `java.util`
- Une seule chose à savoir,
 - Mais où donc se trouve la doc en ligne ???
 - Par exemple
 - ici c'est l'index des API



java.lang

Provides classes that are fundamental to the design of the Java programming language.

See:

[Description](#)

Interface Summary

<u><i>Cloneable</i></u>	A class implements the <code>Cloneable</code> interface to indicate to the <u><code>Object.clone()</code></u> method that it is legal for that method to make a field-for-field copy of instances of that class.
<u><i>Comparable</i></u>	This interface imposes a total ordering on the objects of each class that implements it.
<u><i>Runnable</i></u>	The <code>Runnable</code> interface should be implemented by any class whose instances are intended to be executed by a thread.

Class Summary

<u>Boolean</u>	The Boolean class wraps a value of the primitive type <code>boolean</code> in an object.
<u>Byte</u>	The Byte class is the standard wrapper for byte values.
<u>Character</u>	The Character class wraps a value of the primitive type <code>char</code> in an object.
<u>Character.Subset</u>	Instances of this class represent particular subsets of the Unicode character set.
<u>Character.UnicodeBlock</u>	A family of character subsets representing the character blocks defined by the Unicode 2.0 specification.
<u>Class</u>	Instances of the class <code>Class</code> represent classes and interfaces in a running Java application.
<u>ClassLoader</u>	The class <code>ClassLoader</code> is an abstract class.
<u>Compiler</u>	The <code>Compiler</code> class is provided to support Java-to-native-code compilers and related services.
<u>Double</u>	The Double class wraps a value of the primitive type <code>double</code> in an object.
<u>Float</u>	The Float class wraps a value of primitive type <code>float</code> in an object.
<u>InheritableThreadLocal</u>	This class extends <code>ThreadLocal</code> to provide inheritance of values from parent <code>Thread</code> to child <code>Thread</code> : when a child thread is created, the child receives initial values for all <code>InheritableThreadLocals</code> for which the parent has values.
<u>Integer</u>	The Integer class wraps a value of the primitive type <code>int</code> in an object.
<u>Long</u>	The Long class wraps a value of the primitive type <code>long</code> in an object.
<u>Math</u>	The class <code>Math</code> contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
<u>Number</u>	The abstract class <code>Number</code> is the superclass of classes <code>Byte</code> , <code>Double</code> , <code>Float</code> , <code>Integer</code> , <code>Long</code> , and <code>Short</code> .
<u>Object</u>	Class <code>Object</code> is the root of the class hierarchy.
<u>Package</u>	<code>Package</code> objects contain version information about the implementation and specification of a Java package.
<u>Process</u>	The <code>Runtime.exec</code> methods create a native process and return an instance of a subclass of <code>Process</code> that can be used to control the process and obtain information about it.
<u>Runtime</u>	Every Java application has a single instance of class <code>Runtime</code> that allows the application to interface with the environment in which the application is running.
<u>RuntimePermission</u>	This class is for runtime permissions.
<u>SecurityManager</u>	The security manager is a class that allows applications to implement a security policy.
<u>Short</u>	The Short class is the standard wrapper for short values.
<u>StrictMath</u>	The class <code>StrictMath</code> contains methods for performing basic numeric operations such as the elementary exponential, logarithm, square root, and trigonometric functions.
<u>String</u>	The <code>String</code> class represents character strings.

java.util

Contains the collections framework, legacy collection classes, event model, date and time facilities, internationalization, and miscellaneous utility classes (a string tokenizer, a random-number generator, and a bit array).

See:

[Description](#)

Interface Summary

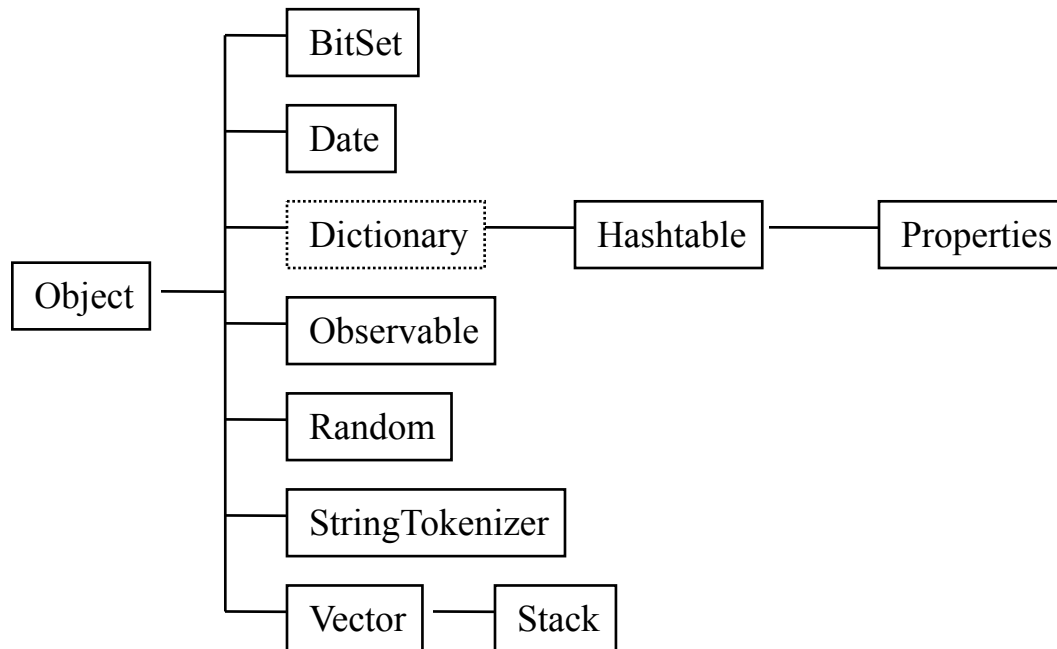
<u>Collection</u>	The root interface in the <i>collection hierarchy</i> .
<u>Comparator</u>	A comparison function, which imposes a <i>total ordering</i> on some collection of objects.
<u>Enumeration</u>	An object that implements the Enumeration interface generates a series of elements, one at a time.
<u>EventListener</u>	A tagging interface that all event listener interfaces must extend.
<u>Iterator</u>	An iterator over a collection.
<u>List</u>	An ordered collection (also known as a <i>sequence</i>).
<u>ListIterator</u>	An iterator for lists that allows the programmer to traverse the list in either direction and modify the list during iteration.
<u>Map</u>	An object that maps keys to values.
<u>Map.Entry</u>	A map entry (key-value pair).
<u>Observer</u>	A class can implement the Observer interface when it wants to be informed of changes in observable objects.
<u>Set</u>	A collection that contains no duplicate elements.
<u>SortedMap</u>	A map that further guarantees that it will be in ascending key order, sorted according to the <i>natural ordering</i> of its keys (see the Comparable interface), or by a comparator provided at sorted map creation time.
<u>SortedSet</u>	A set that further guarantees that its iterator will traverse the set in ascending element order, sorted according to the <i>natural ordering</i> of its elements (see Comparable), or by a Comparator provided at sorted set creation time.

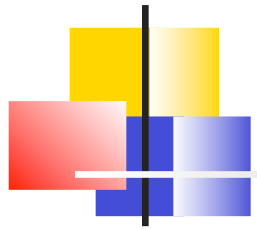
Class Summary

<u>AbstractCollection</u>	This class provides a skeletal implementation of the <code>Collection</code> interface, to minimize the effort required to implement this interface.
<u>AbstractList</u>	This class provides a skeletal implementation of the <code>List</code> interface to minimize the effort required to implement this interface backed by a "random access" data store (such as an array).
<u>AbstractMap</u>	This class provides a skeletal implementation of the <code>Map</code> interface, to minimize the effort required to implement this interface.
<u>AbstractSequentialList</u>	This class provides a skeletal implementation of the <code>List</code> interface to minimize the effort required to implement this interface backed by a "sequential access" data store (such as a linked list).
<u>AbstractSet</u>	This class provides a skeletal implementation of the <code>Set</code> interface to minimize the effort required to implement this interface.
<u>ArrayList</u>	Resizable-array implementation of the <code>List</code> interface.
<u>Arrays</u>	This class contains various methods for manipulating arrays (such as sorting and searching).
<u>BitSet</u>	This class implements a vector of bits that grows as needed.
<u>Calendar</u>	<code>Calendar</code> is an abstract base class for converting between a <code>Date</code> object and a set of integer fields such as YEAR, MONTH, DAY, HOUR, and so on.
<u>Collections</u>	This class consists exclusively of static methods that operate on or return collections.
<u>Date</u>	The class <code>Date</code> represents a specific instant in time, with millisecond precision.
<u>Dictionary</u>	The <code>Dictionary</code> class is the abstract parent of any class, such as <code>Hashtable</code> , which maps keys to values.
<u>EventObject</u>	The root class from which all event state objects shall be derived.
<u>GregorianCalendar</u>	<code>GregorianCalendar</code> is a concrete subclass of <u>Calendar</u> and provides the standard calendar used by most of the world.
<u>HashMap</u>	Hash table based implementation of the <code>Map</code> interface.
<u>HashSet</u>	This class implements the <code>Set</code> interface, backed by a hash table (actually a <code>HashMap</code> instance).
<u>Hashtable</u>	This class implements a hashtable, which maps keys to values.
<u>LinkedList</u>	Linked list implementation of the <code>List</code> interface.
<u>ListResourceBundle</u>	<code>ListResourceBundle</code> is a abstract subclass of <code>ResourceBundle</code> that manages resources for a locale in a convenient and easy to use list.



java.util.*





java.util.Hashtable

- Cette classe gère une collection d'objets au travers d'une table de hachage dont les clés sont des `String` et les valeurs associées des `Object`.

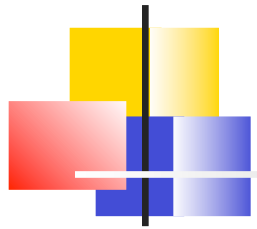
```
Hashtable ht = new Hashtable();

ht.put("noel", new Date("25 Dec 1997"));

ht.put("un vecteur", new Vector());

Vector v = (Vector)ht.get("un vecteur");

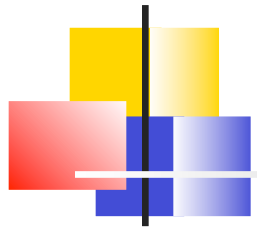
for(Enumeration e = ht.keys(); e.hasMoreElements();) {
    String key = (String)e.nextElement;
    ...
}
```



java.util.Properties

- Cette classe gère une collection d'objets au travers d'une table de hachage dont les clés et les valeurs sont des String.

```
Properties p = new Properties();  
p.put("&eacute;", "'e");  
p.put("&egrave", "`e");  
p.put("&ecirc", "^e");  
  
String s = p.getProperty("&eacute;");  
for(Enumeration e = p.keys(); e.hasMoreElements();) {  
    String key = (String)e.nextElement;  
    ...  
}
```



java.util.StringTokenizer

- Cette classe permet de découper une String selon des séparateurs.

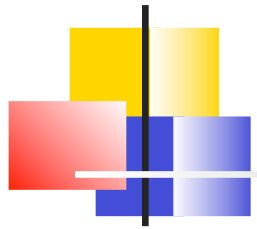
```
String str = "avion, bateau ; train  ";
```

```
StringTokenizer st = new StringTokenizer(str, ";, ");
```

```
System.out.println(st.nextToken()); // --> avion
```

```
System.out.println(st.nextToken()); // --> bateau
```

```
System.out.println(st.nextToken()); // --> train
```



java.util.Vector

- Cette classe gère une collection d'objet dans un tableau dynamique.

```
Vector v = new Vector();
```

```
v.addElement("une chaine");
```

```
v.addElement(new date());
```

```
v.addElement(new String[]);
```

```
v.addElement(new Vector());
```

```
v.setElementAt("abcde", 2);
```

```
System.out.println(v.elementAt(2)); // --> abcde
```