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
- Les Chaînes de Caractères

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;
 }
```


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;
    }
  }
```

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.
  - Une seule fois.

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
```

- Java ne permet pas la surcharge des opérateurs, comme C ++.
- 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

- La classe `PointNomme` hérite de `Point`
  - 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`).

Comparaison avec C++

- En C++, pour qu'un appel polymorphe ait lieu, il faut que la méthode ait été déclarée comme `virtual` : par défaut, une méthode redéfinie ne sera pas appelée (ce sera celle de la classe apparente qui le sera).
- En Java, c'est le contraire : par défaut, toute méthode redéfinie sera appelée selon le type dynamique de l'objet.
- Java utilise le mot-clé `final` pour indiquer qu'une méthode ne peut pas être redéfinie. Le compilateur n'inclut pas alors le code de recherche dynamique.
- Ce mot-clé s'applique également à une classe : une classe `final` ne peut pas être dérivée (et toutes ses méthodes sont considérées comme `final`). Lorsqu'une de ses méthodes d'instance est appelée, le compilateur sait qu'il ne peut s'agir d'un appel polymorphe, il n'inclut pas non plus le code de recherche dynamique.

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 : (`PointNomme` hérite de `Point`)

```
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

Transtypage : downcasting

- Soit le code suivant : (`PointNomme` hérite de `Point`)

```
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();
  }
}
```

```
/* 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 **mais inaccessibles pour les utilisateurs de la classe de base**.

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
```

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.

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 monTableau[];`
  - `int[] monTableau;`

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

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 :
  - les chaînes de caractères sont des objets
  - pas de marqueur de fin de chaîne
  - pas de calcul de longueur lors de la création
  - vérification des débordements
  - 2 classes :
    - 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` >= `start`
  - `indexOf(String str)` : première position de `str`
  - `indexOf(String, int start)` : première position de `str` >= `start`
  - `lastIndexOf(char ch)` : dernière position de `ch`
  - `lastIndexOf(char ch, int start )` : dernière position de `ch` <= `start`
  - `lastIndexOf(String str)` : dernière position de `str`
  - `lastIndexOf(String str, int start)` : dernière position de `str` <= `start`

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
- 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)
- 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 chaîne 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.

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)`

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
- 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 : `append`, L'insertion : `insert`
- Suppression de caractères : `delete`
- Affectation d'un caractère particulier : `setCharAt`
- 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