Séquence 1

- **Les classes** : modèle pour construire des objets,
  = attributs (privés) + méthodes (publiques)

- **Les attributs** : nom/type/valeur,
  types primitifs (valeur) et objets (référence)

- **Les méthodes** : fonction (*return type*), procédure (void),
  constructeur (spécial, rôle)

- **minuscules/Majuscules** : règles de nommage standard

- **Les variables** : **a**ttribut/**p**aramètre/**v**ariable locale

- **`this`** : objet courant

Séquence 2

- **Les ordres :** attributs, constructeurs, méthodes, instructions, paramètres

- **Les commentaires :** `//` fin de ligne, `/**` javadoc, multilignes, `@ */`

- **Définition de méthode :** signature + corps,
  fonction (typeRetour, `return`) et procédure (`void`)

- **Les constructeurs :** plusieurs, par défaut, naturel, quelconque

- **Duplication de code :** constructeur appelle un autre : `this();` en 1ère instruction

- **Création d'objets :** déclaration/création, instance/instancier, `new`, paramètres

- **Accesseurs & modificateurs :** rôles, fonction/procédure, paramètre ou pas

- **Expressions et instructions :** valeur ou pas, types, affectation/comparaison,
  terminaison/affichage

- **Tests :** `if`, `if else`, parenthèses, une ou plusieurs instructions

- **Récursivité :** méthode qui s'appelle elle-même, relation de récurrence,
  test d'arrêt, paramètre qui change ; `return;`

Séquence 3

- **Appel de méthode :** objet**.**méthode(paramètres), expression/instruction, `new Classe(paramètres)`

- **`return expression;` `return;`**

- **`null`, `NullPointerException`**

- **`if (condition) return true; else return false;`** dans une fonction booléenne

- **`if` non indépendants, `else`**

- **`x=y;` `x == y` `x.equals(y)`**

- **`System.out.println` `System.out.print`**

- **`private`,** protection autour de la classe et non de chaque objet

Séquence 4

- Tableaux
- Les boucles `for` et `while`
- Limitations du type `int` et du type double
- Type primitif ≠ Type objet (*reference type*)
- Expressions : entières, réelles, booléennes, String, Cercle, ...
- Instruction ≠ expression
- Méthodes statiques = méthodes de classe (et non méthodes d'instance)
- Comment écrire une méthode d'après un énoncé ?

Séquence 5

- **Attributs statiques** = attributs de classe = variables de classe (partagés => en un seul exemplaire).
- **Constantes** => attributs statiques
  `private static final int NOM_EN_MAJ = 1000;`
- **héritage** entre classes, seulement si « est une sorte de »
- **signature** de la classe suivie de `extends ClasseMère`
- **type déclaré / type constaté** (compilation / exécution)
- Pourquoi `Animal vA1 = new Oiseau();`
  et pas `Oiseau vO1 = new Animal();` ?
- **Redéfinition** d'une méthode dans une sous-classe
- **`Object`**, mère de toutes les classes => `extends Object` automatique, pourtant un seul `extends` !?
- **Constructeur naturel** dans une sous-classe :
  combien de paramètres ?

Séquence 6

- **vocabulaire** : Polymorphisme.
- **classe abstraite** => non instanciable
- constructeurs **inutiles ?**
- **méthode abstraite** = signature sans corps => non exécutable
- méthode non abstraite **peut appeler** une méthode abstraite
- **interface** => ni attribut ni constructeur,
  que des méthodes abstraites
- **`implements`** n'est pas incompatible avec
  `extends UneSuperClasse`
- **Exemples** d'interfaces dans le JDK :
- Il est possible de **créer** ses propres interfaces
- **Héritage d'interface** (multiple !)
- Nouveau mot java : `unObjet` **`instanceof`** `UnType`

Séquence 7

- **Framework Collections** = ensemble de classes abstraites ou non, d'interfaces, et d'algorithmes.

- **>**tableaux : - seulement des objets + extensible + propriétés/méthodes

- Primitifs interdits => classes enveloppes, auto-(un)boxing, généricité systématique => Coll<TypeE> *(à la déclaration ET à la création)*

- Interface la plus haute = `Collection`
  `List extends Collection`, `AbstractList implements List`, `ArrayList / LinkedList extends AbstractList`, `Collections`

- `Iterator<TypeE>` permet de parcourir une `Collection<TypeE>`, fournit 3 méthodes : `hasNext`, `next`, `remove`

- Les parcours : simples ou non ?
  - Simple => boucle *for each* (fonctionne aussi sur les tableaux)
  - Non simple *(ou jusqu'à java 1.4.2)* => boucle `while` avec un itérateur

Séquence 8 (1/2)

1. Qu'est-ce qu'une exception ?

2. Qu'est-ce qu'une exception en Java ?

3. Quand/comment est créé cet objet ?

4. Avantages attendus du système des exceptions

5. Exceptions existant dans Java

Hiérarchie, `Error`, `Exception`, `RuntimeException`

6. Que faire des exceptions lancées par le JDK ?

- successivement plusieurs blocs `catch` après un seul `try`
- traitement commun à plusieurs exceptions héritant d'une même classe
- pas `RuntimeException` ==> `try/catch` ou `throws`
- traiter partiellement et relancer l'exception dans `catch`
- quelle que soit la sortie de la méthode, bloc `finally`

Séquence 8 (2/2)

- **7.** Lancer volontairement des exceptions du JDK
- 7.1. `if (mauvais paramètre)`
  `throw new UneClasseException("...");`
- 7.2. Ne pas rattraper une exception dans la même méthode
- **8.** Créer ses propres exceptions
- 8.1. Classe `extends Exception` ou une de ses sous-classes
- 8.2. Redéfinir `getMessage()` (visible aussi dans `toString()`)
- 8.3. Ajouter des attributs (donc constructeur),
  voire des méthodes (accesseurs, ...)
- **9.** Bonus : Les assertions
- 9.1. Programmation par contrat, sécurité :
  pré-condition, post-condition, pas d'invariant
- 9.2. `assert condition : message;`
- 9.3. Pour ne pas vérifier les assertions, supprimer `-ea`

(séq.9) Ligne de commande

- **Lancer un programme java depuis la ligne de commande :**

- **`java MaClasse`**
- **ou bien**
- **`java MaClasse mot1 ... motN`**

(séq.9) Méthode `main`

- **Quelle méthode de MaClasse est alors appelée ?**
- **La méthode qui a une signature précise :**

- L'appel provient de l'extérieur de la classe
- L'appel se fait directement sur la classe
- Elle ne retourne rien ==> procédure
- Son nom doit être imposé
- On veut pouvoir passer des arguments sur la ligne de commande (0 ou 1 ou plusieurs)

- `public static void main( final String[] pArgs )`

(séq.9) Extraction d'un nombre à partir d'une `String`

- entier ==> `Integer` ==> `parseInt`

```
int vN;
try {
    vN = Integer.parseInt( uneString );
}
catch ( final NumberFormatException pE ) {
    vN = uneValeurParDéfaut;
} // t/c
```

- réel ==> `Double` ==> `parseDouble`

(séq.9) Développement sans BlueJ

- **`javac MaClasse.java`**
- ou
- **`javac *.java`**
- et aussi
- **`javadoc ...`**

(séq.9) Générateur aléatoire

- **Générateur de nombres (pseudo-)aléatoires :**
- `Random vMonGen = new Random();`
- `vMonGen.nextDouble()`
  retourne un réel dans l'intervalle [0.0, 1.0]
- `vMonGen.nextInt(N)`
  retourne un entier dans l'intervalle [0, N-1]

- `Random vMonGen = new Random( seed );`
  permet d'obtenir toujours la même suite de nombres.

(séq.9) Les classes anonymes

- Sans nom, définies 'à la volée' :
- Dans `UserInterface`
  `monBouton.addActionListener( ? );`
- **?** doit être un `ActionListener`, donc sa classe doit implémenter l'interface `ActionListener`.
- Dans *zuul-with-images*, on met `this` car `UserInterface implements ActionListener`, ce qui veut dire qu'elle redéfinit la méthode `actionPerformed`.

(séq.9) Les classes anonymes

```
monBouton.addActionListener(
   new ActionListener() {
     public void actionPerformed(
       final ActionEvent pE ) {
         actions à réaliser
     } // actionPerformed(.)
} );
```

- On crée une instance d'une classe anonyme (puisqu'on ne lui donne pas de nom !) qui implémente l'interface `ActionListener` en redéfinissant la méthode `actionPerformed`.
- **Attention !** Lorsqu'on utilise `this` entre les accolades, cela fait référence à l'instance de la classe anonyme.

Traité en projet

- Évènements d'interface graphique (composants graphiques Swing)
- Lecture de fichiers de texte
- `HashMap`, `Set`, `Stack`
- `enum`
- Instruction `switch`

- Paquetages

Non traité

- Autres types primitifs *(8 en tout)*
- Autres types enveloppes *(8 en tout)*
- Autres collections *(beaucoup)*
- Méthodes de la classe String *(beaucoup)*
- Entrées/sorties nio *(fichiers)*
- Sérialisation *(transmission d'objets)*
- Threads *(parallélisme)*
- Java 8 *(fonctions lambda, …)*
- etc …