# Java 5

*(nom de code Tiger)*

*Licence d'informatique 3ème année*



## Introduction

- La version 1.5, contrairement aux versions précédentes, peut être considérée comme une petite révolution pour Java car elle apporte énormément d'améliorations sur le langage.
- Le but principal de ces ajouts est de faciliter le développement d'applications avec Java en simplifiant l'écriture et la lecture de code.
- Un code utilisant les nouvelles fonctionnalités de Java 1.5 ne pourra pas être exécuté dans une version antérieure de la JVM.



## Nouveautés Java 1.5

- Autoboxing / Unboxing
- Static import
- Nouvelle boucle for (for-each)
- Les méthodes à nombres variables d'arguments
- Les énumérations
- Les entrées/sorties formatées
- Les génériques



## Autoboxing / Unboxing

- L'autoboxing permet de transformer automatiquement une variable de type primitif en un objet du type du wrapper correspondant.
- L'unboxing est l'opération inverse.
- Avant la version 1.5,pour ajouter des entiers dans une collection, il était nécessaire d'encapsuler chaque valeur dans un objet de type Integer.
- Depuis la version 1.5, l'encapsulation de la valeur dans un objet n'est plus obligatoire car elle sera réalisée automatiquement par le compilateur.

# Autoboxing / Unboxing

Avant Java 1.5

```
public class TestAutoboxingOld {
  public static void main(String[] args) {
    List liste = new ArrayList();
    Integer valeur = null;
    for(int i = 0; i < 10; i++) {
      valeur = new Integer(i);
      liste.add(valeur);
    }}}
```

Depuis Java 1.5

```
public class TestAutoboxing {
  public static void main(String[] args) {
    List liste = new ArrayList();
    for(int i = 0; i < 10; i++) {
      liste.add(i);
    }}}
```



# Static import

- Jusqu'à la version 1.4, pour utiliser un membre statique d'une classe, il fallait obligatoirement préfixer ce membre par le nom de la classe qui le contenait.

Avant Java 1.5

```
public class TestStaticImportOld {

  public static void main(String[] args) {
    System.out.println(Math.PI);
    System.out.println(Math.sin(0));
  }
}
```



# Static import

- Java 1.5 propose une nouvelle fonctionnalité concernant l'importation de package : l'import statique (static import).
- Ce nouveau concept permet d'appliquer les mêmes règles aux membres statiques qu'aux classes et interfaces pour l'importation classique.
- Elle s'utilise comme une importation classique en ajoutant le mot clé static.

Depuis Java 1.5

```
import static java.lang.Math.*;
public class TestStaticImport {
  public static void main(String[] args) {
    System.out.println(PI);
    System.out.println(sin(0));
  }}
```

L'utilisation de l'importation statique s'applique à tous les membres statiques : constantes et méthodes statiques de l'élément importé.



# Amélioration des boucles pour les collections

- Jusqu'à la version 1.4, l'itération des éléments d'une collection était syntaxiquement un peu lourde avec la déclaration d'un Iterator.

Avant Java 1.5

```
public class TestForOld {
  public static void main(String[] args) {
    List liste = new ArrayList();

    for(int i = 0; i < 10; i++)
      liste.add(i);

    for (Iterator iter = liste.iterator(); iter.hasNext(); )
      System.out.println(iter.next());

  }
}
```

## Amélioration des boucles pour les collections

- La version 1.5 propose une nouvelle syntaxe de l'instruction for qui permet de simplifier l'écriture du code pour réaliser une itération et laisse le soin au compilateur de générer le code nécessaire.

Depuis Java 1.5

```
public class TestFor {
  public static void main(String[] args) {
    List liste = new ArrayList();

    for(int i = 0; i < 10; i++)
      liste.add(i);

    for (Object element : liste)
      System.out.println(element);
  }
}
```



## Amélioration des boucles pour les collections

- Cette nouvelle syntaxe peut aussi être utilisée pour parcourir tous les éléments d'un tableau.

```
public class TestForArray {

  public static void main(String[] args) {
    int[] tableau = {0,1,2,3,4,5,6,7,8,9};

    for (int element : tableau) {
      System.out.println(element);
    }
  }

}
```



## Les méthodes à nombres variables d'arguments

- Cette nouvelle fonctionnalité va permettre de passer un nombre non défini d'arguments d'un même type à une méthode. Ceci va éviter de devoir encapsuler ces données dans une collection.
- Cette nouvelle fonctionnalité implique une nouvelle notation pour préciser la répétition d'un type d'argument : trois points «...»



## Les méthodes à nombres variables d'arguments

```
public class TestVarargs {
  public static void main(String[] args) {
    System.out.println("valeur 1 = " + additionner(1,2,3));
    System.out.println("valeur 2 = " + additionner(2,5,6,8,10));
  }

  public static int additionner(int ... valeurs) {
        int total = 0;
        for (int val : valeurs) {
                total += val;
        }
        return total;
  }
}
```

# Les méthodes à nombres variables d'arguments

- Les arguments sont traités comme un tableau : il est d'ailleurs possible de fournir les valeurs sous la forme d'un tableau.

```
public class TestVarargs2 {
  public static void main(String[] args) {
    int[] valeurs = {1,2,3,4};
    System.out.println("valeur 1 = " + additionner(valeurs));
  }
  public static int additionner(int ... valeurs) {
        int total = 0;
        for (int val : valeurs) {
                total += val;
        }
        return total;
  }
}
```



# Les méthodes à nombres variables d'arguments

- Il n'est cependant pas possible de mixer des éléments unitaires et un tableau dans la liste des éléments fournis en paramètres.

```
public class TestVarargs3 {
  public static void main(String[] args) {
    int[] valeurs = {1,2,3,4};
    System.out.println("valeur 1 = " +
additionner(5,6,7,valeurs));
  }
  public static int additionner(int ... valeurs) {
        int total = 0;
        for (int val : valeurs) {
                total += val;
        }
        return total;
  }
}
```

Cet exemple ne marche pas !!!



# Les énumérations (type enum)

- Souvent lors de l'écriture de code, il est utile de pouvoir définir un ensemble fini de valeurs pour une donnée.
  - Ceci permet de définir les valeurs possibles qui vont caractériser l'état de cette donnée.
- Pour cela, Java 1.5 introduit le type enum qui permet de définir un ensemble de constantes.
  - Cette fonctionnalité existe déjà dans les langages C et Delphi entre autre.



# Les énumérations (type enum)

- Jusqu'à la version 1.4, la façon la plus pratique pour palier au manque du type enum était de créer des constantes dans une classe.

Avant Java 1.5

```
public static final int SEASON_WINTER = 0;
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_FALL   = 3;
```

## Les énumérations (type enum)

- Cette méthode présentes des inconvénients :
  - pas de vérification de type : les saisons étant simplement des valeurs entières, il est possible d'affecter une autre valeur (ne représentant pas une saison) ou de faire une addition de saisons.
  - pas d'espace de nommage : dans notre exemple le nom de chaque constante est préfixé de « SEASON_ » pour éviter toute collision avec une autre énumération de valeurs entières.
  - Les constantes entières s'affichent comme des valeurs entières => pas très informatif.
  - Etc.



## Les énumérations (type enum)

- La version 1.5 propose une fonctionnalité pour déclarer et utiliser un type enumération qui repose sur trois éléments :
  - le mot clé enum
  - un nom pour désigner l'énumération
  - un ensemble de valeurs séparées par des virgules

Depuis Java 1.5

```
enum Season { WINTER, SPRING, SUMMER, FALL }
```



## Les énumérations (type enum)

- A la rencontre du mot clé enum, le compilateur va automatiquement créer une classe possédant les caractéristiques suivantes :
  - un champ static est défini pour chaque élément précisé dans la déclaration enum
  - une méthode values() qui renvoie un tableau avec les différents éléments définis
  - la classe implémente les interfaces Comparable et Serialisable
  - les méthodes toString(), equals(), hashCode() et CompareTo() sont redéfinies



## Les énumérations (type enum)

```
public class Card {
    public enum Rank { DEUCE, THREE, FOUR, FIVE, SIX,
        SEVEN, EIGHT, NINE, TEN, JACK, QUEEN, KING, ACE }
    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }
    private final Rank rank;
    private final Suit suit;
    private Card(Rank rank, Suit suit) {
        this.rank = rank;
        this.suit = suit;
    }

    public Rank rank() { return rank; }
    public Suit suit() { return suit; }
    public String toString() { return rank + " of " + suit; }

    private static final List protoDeck = new ArrayList();

    static {
        for (Suit suit : Suit.values())
            for (Rank rank : Rank.values())
                protoDeck.add(new Card(rank, suit));
    }

    public static ArrayList newDeck() {
        return new ArrayList(protoDeck); }
}
```

# Les énumérations (type enum)

- La classe Card est immuable et une seul instance de chaque carte est créée (ce qui permet de ne pas surcharger equals() et hashCode()).
- La méthode toString() utilise les méthodes toString() de Rank et Suit.
- Le compilateur vous informera si vous intervertissez les deux arguments. Ceci aurait provoqué une erreur d'exécution avec des constantes entières.
- Chaque type énuméré a une méthode statique values qui retourne un tableau contenant toutes les valeurs du type énuméré dans l'ordre des déclarations.
  - Cette méthode est très utile pour être utilisée avec la nouvelle boucle for (for-each)



# Entrées et Sorties formatées

- Il est désormais possible d'utiliser une syntaxe du type `printf` pour la génération de sortie formatée.
- Ceci permet de maîtriser, par exemple, les affichages des numériques (ex. : 2 chiffres après la virgule, etc.).
- La plupart des formatages de la fonction C `printf` est disponible, avec quelques formats supplémentaires pour les classes comme Date et BigInteger.
- Vous pouvez vous référer à la classe java.util.Formatter pour avoir plus d'informations.



# Entrées et Sorties formatées

- Bien que le caractère de nouvelle ligne UNIX '\n' soit accepté, il est recommandé d'utiliser la séquence %n en Java afin d'assurer une portabilité entre les différentes plate-formes.

```
System.out.printf("name count%n");
System.out.printf("%s %5d%n", user, total);
```



# Entrées et Sorties formatées

- La classe Scanner met à notre disposition des fonctions pour la lecture de données à travers la console système, ou tout flux de données.
- L'exemple suivant lit une chaîne de caractères String, puis un nombre entier à partir de l'entrée standard.

```
Scanner s = new Scanner(System.in);
String param = s.next();
int value = s.nextInt();
s.close();
```

- Les méthodes comme next et nextInt de la classe Scanner bloqueront si aucune donnée n'est disponible.
- Si vous avez besoin de gérer des entrées plus complexes, il existe des algorithmes par motif dans la classe java.util.Scanner.

# Les génériques

- Les génériques permettent de faire des abstractions sur les types.
- L'exemple le plus courant de l'utilisation des génériques est celui des containers.

```
List maListeEntiers = new LinkedList();
maListeEntiers.add(new Integer(0));
Integer x = (Integer)maListeEntiers.iterator().next();
```



# Les génériques

- Le transtypage de la 3ème ligne est gênant :
  - Le compilateur garantit que l'itérateur renvoie un Object.
  - Le programmeur doit transtyper pour pouvoir affecter le résultat à une variable de type Integer.
  - Le programmeur doit donc être certain du type d'objet contenu dans le container à cet endroit précis.
  - Si le type de l'objet contenu dans le container ne correspond pas au type dans lequel le programmeur cherche à le transtyper, une runtime error est générée.



# Les génériques

- Les génériques (Java 1.5) permettent de ne plus se trouver confronté à ce problème de transtypage
- Il devient possible de restreindre le type des objets contenus dans un container à un certain type.

```
List<Integer> maListeEntiers = new LinkedList<Integer>();
maListeEntiers.add(new Integer(0));
Integer x = maListeEntiers.iterator().next();
```



# Les génériques

- Cette façon de déclarer la List précise qu'il ne s'agit pas d'une List quelconque mais d'une List d'Integer.
- On dit alors que la List est une interface générique qui prend en paramètre le type des données qu'elle peut contenir (ici Integer).

## Les génériques

- Dans notre exemple le transtypage a donc disparu puisque le compilateur sait que cette List ne peut contenir que des Integer.
- Ce n'est donc plus le programmeur qui sait (cas du transtypage) mais le compilateur.
- Et il est de toute évidence préférable de faire confiance au compilateur plutôt qu'au programmeur.
- Ceci améliore la robustesse et la lisibilité du code.



## Les génériques

- Voici un extrait de la définition des interfaces List et Iterator du package java.util de Java 1.5 :

```
public interface List<E> {
  void add(E x);
  Iterator<E> iterator();
}

public interface Iterator<E> {
  E next();
  boolean hasNext();
}
```



## Les génériques

- Ce qui se trouve entre chevrons sont les paramètres formels de type des interfaces List et Iterator.
- Dans le cas de notre liste d'entiers, le paramètre formel E est remplacé par le paramètre effectif Integer.
- Convention de nommage : il est recommandé d'utiliser un caractère majuscule comme nom pour les paramètres formels de type.



## Les génériques

- Le code suivant est-il valide ?

```
List<String> ls = new ArrayList<String>();
List<Object> lo = ls;
```

- La question qui se pose est : Est-ce qu'une List de String est une List d'Object ?
- Un élément de réponse :

```
lo.add(new Object());
String s = ls.get(0);
```

## Les génériques

- Si Foo est un sous-type (sous-classe) de Bar, et G est une déclaration de type générique, G<Foo> n'est pas un sous-type de G<Bar>.
- C'est probablement la chose la plus difficile que vous ayez à apprendre (et à comprendre) au sujet des génériques car cela va à l'encontre de votre intuition de bons concepteurs objets.



## Les génériques

- Considérons une méthode permettant d'afficher tous les éléments d'une collection.

Avant Java 1.5

```
void printCollection(Collection c) {
  Iterator i = c.iterator();
  for (k = 0; k < c.size(); k++) {
    System.out.println(i.next());
}}
```

Tentative naïve en Java 1.5

```
void printCollection(Collection<Object> c) {
  for (Object e : c)
    System.out.println(e);
}}
```



## Les génériques

- La méthode écrite en Java 1.5 est moins puissante.
- Alors que la première version permet d'afficher les éléments d'une Collection quel que soit leur type, la deuxième ne permet d'afficher que des Object.
- En effet, la méthode prend un argument du type Collection<Object> et nous avons vu que ce n'est pas un super-type de toutes les Collection.



## Les génériques

- Il existe un moyen de définir le super-type de toutes les Collection : Collection<?>
- On peut parler de collection d'inconnus.
- Le ? est appelé « wildcard type » (= joker).
- La méthode pourrait donc s'écrire :

```
void printCollection(Collection<?> c) {
  for (Object e : c)
    System.out.println(e);
}}
```

# Les génériques

- Dans la méthode `printCollection`(), on peut lire les éléments de c et leur donner le type Object. Ceci est toujours *sûr* puisque les collections contiennent forcément des descendants d'Object.
- A l'inverse, ce n'est pas sûr d'ajouter des Object à la Collection :

```
Collection<?> c = new ArrayList<String>();
c.add(new Object());
```

- Ce code génère un erreur à la compilation !!!



# Les génériques

- Considérons l'exemple d'une application de dessin :

```
public abstract class Shape {
  public abstract void draw(Canvas c);
}

public class Circle extends Shape {
  private int x, y, radius;
  public void draw(Canvas c) { ... }
}

public class Rectangle extends Shape {
  private int x, y, width, height;
  public void draw(Canvas c) { ... }
}
```



# Les génériques

- Ces classes peuvent se dessiner sur une toile :

```
public class Canvas {
  public void draw(Shape s) { s.draw(this);
}
```

- Chaque dessin peut contenir plusieurs formes. En supposant que les formes d'un dessin sont contenues dans une List, on souhaite avoir une méthode dans Canvas qui dessine toutes les formes.

```
public void drawAll(List<Shape> shapes) {
  for (Shape s: shapes) { s.draw(this);
}
```



# Les génériques

- Avec ce que nous avons vu auparavant, nous savons que `drawAll()` ne peut être invoquée qu'avec des List<Shape> (exactement) et non, par exemple, avec des List<Circle>.
- Heureusement, il y a une possibilité pour spécifier que la méthode peut accepter une liste de n'importe quel type de Shape.

```
public void drawAll(List<? extends Shape> shapes) {...}
```

## Les génériques

- Maintenant `drawAll()` accepte des List de n'importe quelle sous-class de Shape, comme par exemple List<Circle>.
- List<? extends Shape> est un exemple de *bounded wildcard*.
- Shape est le *upper bound* de la wildcard.



## Les génériques

- Les bounded wildcard sont donc bien pratiques pour parcourir une liste contenant des objets dérivant tous de la même classe.
- Cependant, les bounded wildcard connaissent leur limite car il n'est pas possible d'y ajouter des objets dérivant d'une même classe.

```
public void addRectangle(List<? extends Shape> shapes)
{
  shapes.add(new Rectangle());
}
```

- Ce code génère un erreur à la compilation !!!



## Les méthodes génériques

- Tentons d'écrire une méthode qui à partir d'un tableaux d'objets et d'une collection place les éléments du tableau dans la collection :

```
static void fromArrayToCollection(Object[] a, Collection<?> c)
{
  for (Object o : a)
    c.add(o);
}
```

- Ce code génère un erreur à la compilation !!!



## Les méthodes génériques

- Les méthodes génériques apportent une réponse à ce problème.
- Les déclarations de méthodes peuvent être génériques : paramétrées par un ou plusieurs paramètres de type.

```
static <T> void fromArrayToCollection(T[] a, Collection<T> c)
{
  for (T o : a)
    c.add(o);
}
```

- Cette méthode peut être invoquée avec n'importe quel type de Collection dont le type des éléments est un super-type du type des éléments du tableau.

# Les méthodes génériques

```
Object[] oa = new Object[100];
Collection<Object> co = new ArrayList<Object>();
fromArrayToCollection(oa, co);// T inferred to be Object
String[] sa = new String[100];
Collection<String> cs = new ArrayList<String>();
fromArrayToCollection(sa, cs);// T inferred to be String
fromArrayToCollection(sa, co);// T inferred to be Object
Integer[] ia = new Integer[100];
Float[] fa = new Float[100];
Number[] na = new Number[100];
Collection<Number> cn = new ArrayList<Number>();
fromArrayToCollection(ia, cn);// T inferred to be Number
fromArrayToCollection(fa, cn);// T inferred to be Number
fromArrayToCollection(na, cn);// T inferred to be Number
fromArrayToCollection(na, co);// T inferred to be Object
fromArrayToCollection(na, cs);// compile-time error
```



# Les méthodes génériques

- A l'invocation d'un méthode générique, il n'est pas nécessaire de lui passer le paramètre effectif de type.
- Le compilateur peut infèrer le paramètre de type sur la base des types des paramètres effectifs de la méthode.
- En général, le compilateur choisit le type de plus spécifique (i.e le plus bas dans la hiérarchie) qui rend l'appel correct.



# Les méthodes génériques

- Comment choisir entre méthodes génériques et wildcard ?
- Examinons quelques méthodes de la librairie Collection

```
interface Collection<E> {
  public boolean containsAll(Collection<?> c);
  public boolean addAll(Collection<? extends E> c);
}
```

- En utilisant les méthodes génériques cela aurait donné :

```
interface Collection<E> {
  public <T> boolean containsAll(Collection<T> c);
  public <T extends E> boolean addAll(Collection<T>
c);
}
```



# Les méthodes génériques

- Les méthodes `containsAll` et `addAll` n'utilisent qu'une fois le paramètre de type T.
- Ni le type de retour, ni le type des arguments de la méthodes ne dépendent du paramètre de type.
- Dans ce cas on choisit donc la solution wildcard.
- On utilise les méthodes génériques quand il y a dépendance entre les types des arguments de la méthode et/ou avec son type de retour.

## Les méthodes génériques

- Il est possible d'utiliser, à la fois, les méthodes génériques et les wildcard.

```
class Collections {
  public static <T> void copy(List<T> dest, List<? extends T> src)
    {...}
}
```

- Tout objet copié depuis la liste source, doit pouvoir être affecté à un élément du type T de la liste destination.
- Par conséquent, le type d'élément de la liste source peut être n'importe quel sous-type de T.



## Les méthodes génériques

- Nous aurions pu écrire cette méthode sans utiliser de wildcard :

```
class Collections {
  public static <T, S extends T> void copy(List<T> dest, List<S> src)
  {...}
}
```

- Mais nous constatons que S n'est réellement utilisé qu'une seule fois pour le type des éléments de la liste source.
- C'est typiquement un cas où il faut préférer une wildcard.



## Les méthodes génériques

- Supposons que nous souhaitions mémoriser toutes les requêtes à la méthode drawAll dans notre application de dessins :

```
static List<List<? extends Shape>> history = new ArrayList<List<? extends Shape>>();
public void drawAll(List<? extends Shape> shapes) {
  history.addLast(shapes);
  for (Shape s: shapes)
    s.draw(this);
}
```



## Une classe générique est partagée par toutes ses invocations

```
List <String> l1 = new ArrayList<String>();
List<Integer> l2 = new ArrayList<Integer>();
System.out.println(l1.getClass() == l2.getClass());
```

- Vous pensez que cette portion de code affiche FAUX ? C'est FAUX !
- Toutes les instances d'une classe générique ont la même *run-time class*.
- Les variables et méthodes statiques d'une classe générique sont logiquement partagées par toutes les instances. Il n'est donc pas possible de faire référence à un paramètre de type d'une déclaration de type dans une méthode statique ou un constructeur, ou dans une déclaration ou initialisation d'une variable statique.

# instanceof sur les génériques

- Demander à une instance si c'est une instance d'une invocation particulière d'un type générique n'a pas de sens et est incorrecte :

```
Collection cs = new ArrayList<String>();
if (cs instanceof Collection<String>) { ...} // illegale
```



# Transtypage

```
Collection<String> cstr = (Collection<String>) cs; // unchecked warning
```

```
<T> T badCast(T t, Object o) {return (T) o; // unchecked warning
```

- De la même façon, il n'est pas possible d'utiliser les types génériques dans les cast.