Problématique de la généricité

- les versions de Java antérieures à 1.5 permettaient de créer des classes de structures contenant n'importe quels types d'objet :
  - les collections (classes implémentant l'interface ***`Collection`***)
  - des classes créées par le programmeur et travaillant sur des instances d'**`Object`**
- Problèmes :
  - manipuler les objets référencés par ces structures oblige à faire du transtypage vers le bas (*downcasting*). Ex : **`Truc`** étant une classe

```
Vector v = new Vector();
...
Truc t = (Truc) v.get(12);
```

  - risque d'erreur de cast, repérables uniquement à l'exécution (**`ClassCastException`**)
  - impossibilité d'obliger à ce qu'une collection ne contienne qu'un seul type d'objet (comme avec les tableaux)

Principe de la généricité en Java (1/3)

- La solution consiste à paramétrer les classes qui opéraient avant sur des **`Object`** par un type (classe) qui sera défini lors de l'utilisation de la classe. On utilise une variable de type.
- *Exemple :*

```
public class Pair <T> {

   private T one,two;

   public Pair(T one, T two){this.one = one;this.two = two;}

   public T getFirst(){return this.one;}

   public T getSecond(){return this.two;}

   public void setFirst(T one){this.one = one;}

   public void setSecond(T two){this.two = two;}
}
```

Principe de la généricité en Java (2/3)

- *Exemple d'une classe à deux variables de type :*

```
public class PairBis <X,Y> {

   private X first;
   private Y second;

   public PairBis(X a, Y b) {this.first = a; this.second =
   b;}

   public X getFirst() {return this.first;}

   public Y getSecond() {return this.second;}

   public void setFirst(X a) {this.first = a;}

   public void setSecond(Y b) {this.second = b;}

}
```


Principe de la généricité en Java (3/3)

- Une classe générique **ne peut être utilisée telle quelle** mais doit être instanciée.

- Instancier une classe générique consiste à donner une valeur à la (ou les) variable(s) de type

```
public static void main(String arg[]){

   Pair <String> p = new Pair<String>("bonjour", "le monde");
   p.setFirst(new String("Hello"));
   p.setSecond(new String("World"));

   Pair <Pair> pp = new Pair<Pair>... // interdit

   Pair<Pair<String>> pp = new Pair<Pair<String>>(new Pair("salut","la
   terre"),new Pair("ici","la Lune"));

   HashMap<Integer,String> m = new HashMap<Integer,String>();

}
```

Mécanisme de la généricité

- A la compilation, lors de l'instanciation d'une classe générique, les paramètres de type sont remplacés par le type spécifié
- Une seule classe compilée est produite, quelque soit le nombre d'instanciations
- Les variables de type sont oubliées (ne sont plus connues après compilation)
- Le compilateur doit parfois ajouter des cast pour respecter les contraintes de type
- En pratique, le mécanisme est assez complexe (et devient encore plus compliqué avec les types contraints, voir plus loin)

Utilisation des variables de type

- Les types paramètres peuvent être utilisés pour **déclarer** des variables (y compris des tableaux)

```
public class GenClass <T>{

   private T tab[];
   ...
```

- Les types paramètres *ne peuvent être utilisés pour **créer** des objets* (ni des tableaux)
- On ne peut pas utiliser un paramètre de type avec **`instanceof`** (car à l'exécution, le type n'est plus connu)
- On peut utiliser un paramètre de type pour instancier une classe générique dans la classe paramètrée

```
public class Pair <T>{

   l = new ArrayList<T>();
   ...
```

Généricité et abstraction

- Les interfaces et classes abstraites peuvent être rendues génériques par paramétrage de type

```
public interface MyInterface <T>{

   ...

}
```

- On peut instancier une classe (ou interface) générique par une classe abstraite ou une interface.
  - Les objets utilisés dans la classe instanciée seront par contre forcément des instances de classes concrètes

- Il est impossible d'implémenter deux interfaces qui sont des instanciations de la même interface générique

```
public class MyClass implements MyInterface<String>, MyInterface<Integer>{

   ...

}
```


Variable de type et méthode de classe

- Il est impossible d'utiliser une variable de type dans une méthode **`static`**

```
public class Pair <T> {

   ...

   public static void myMethod(){
        T var; // erreur à la compilation
        ...
   }

}
```

Héritage et généricité (1/2)

- On peut étendre une classe générique par une autre classe générique, à condition que les variables de type soient les mêmes (à une susbtitution près)

```
public class Pair <T> {

   ...

}

public class Triplet <T> extends Pair <T> {

   T three;

   public Triplet (T one, T two, T three){
       super(one,two);
       this.three = three;
   }

   public void setThird(T three){this.three = three;}

   public T getThird(){return this.three;}

}
```


Héritage et généricité (2/2)

- Les règles de sous-typage des classes génériques sont :
  - Si **`C <T> extends D <T>`** alors **`C <T>`** est une sous-classe de **`D <T>`** (au sens où toute instance de l'une est instance de l'autre)
  - La relation d'héritage reste valide entre instanciations de **`C`** et **`D`** par un même type. Par exemple, **`C <String>`** est sous-classe de **`D <String>`.**
  - Cette relation d'héritage n'est pas valide entre instanciations de **`C`** et **`D`** par des types différents, *même si ces types sont liés par héritage!*
  - *Exemple : **`Integer`** est sous-classe de **`Object`** mais le code suivant n'est pas correctement typé*

```
Pair<Integer> p = new Pair<Integer>(3,4);
Pair<Object> q = p;
q.setFirst("toto");
```

*On interdit donc que **`Pair<Integer>`** soit sous-classe de **`Pair<Object>`***

  - Une instanciation d'une classe générique n'est pas sous-classe de la classe générique : **`C <String>`** n'est pas sous-classe de **`C <T>`**
  - *Exemple : le code suivant n'a aucun sens*

```
Pair<T> p = new Pair<Integer>(3,4);
```

Méthodes génériques

- Une méthode peut être paramétrée par un type, qu'elle soit dans une classe générique ou non
- L'appel de la méthode nécessite de l'instancier par un type, sauf si le compilateur peut réaliser une *inférence de type*

```
public class MyClass{

    public static <T> void permute(T[] tab, int i, int j){
        T temp = tab[i];
        tab[i] = tab[j];
        tab[j] = temp;
    }
}

...

String[] tabs = new String[]{"toto","titi","tutu"};
Float[] tabf = new Float[]{3.14f,27.12f};

MyClass.<String>permute(tabs,0,2);
MyClass.permute(tabf,0,1);
```


Types contraints

- Problème :
  - on veut pouvoir manipuler les types paramètres (appeler des méthodes sur des instances de ces types, ...)
  - mais on ne connait rien de ces types, à part qu'ils héritent de **`Object`**
- Solution :
  - on précise qu'un type paramètre hérite d'une classe ou d'une (ou plusieurs) interface(s) qui possède(nt) les méthodes que l'on veut utiliser sur les instances du type

```
public class MyClass <Type1 extends OtherClass & Interface1 & Interface2, Type2> {

    ...
```

- *Exemple :*

```
public class MyList <T extends Comparator> {

    ...
}
```

Instanciation avec joker (1/3)

- Problème : on voudrait qu'une méthode écrite pour un type générique T puisse fonctionner avec toute instance des sous-classes de T

- *Exemple :*

```
public static void printNumber(ArrayList<Number> l){
    for(Iterator i = l.iterator();i.hasNext();){
            System.out.println(l.next());
    }
}
```

```
ArrayList<Integer> l = new ArrayList<Integer>();
...
MyClass.printElements(l); // Erreur à la compilation
// ArrayList<Integer> n'est pas sous-type de ArrayList<Number>
```


Instanciation avec joker (2/3)

- Solution : on peut spécifier qu'un paramètre de type est toute sous-classe ou sur-classe d'une classe donnée
  - **<?>** désigne un type inconnu
  - **<? extends C>** désigne un type inconnu qui est soit **C** soit un sous-type de **C**
  - **<? super C>** désigne un type inconnu qui est soit **C** soit un sur-type de **C**
  - **C** peut être une classe, une interface ou un paramètre de type

- *Exemples :*

```
public static void printNumber(ArrayList<? extends Number> l){
    for(Iterator i = l.iterator();i.hasNext();){
        System.out.println(l.next());
    }
}
```

```
public static <T> void fillElement(ArrayList<? super T> l, T e){
    for(int i = 0;i<l.size();i++){
        l.set(i,e);
    }
}
```

Instanciation avec joker (3/3)

- De tels types avec joker ne peuvent être utilisés qu'à l'instanciation d'une classe générique!
- De tels types avec joker ne peuvent pas être utilisés pour créer des objets ou des tableaux (sauf si le type est non contraint)

```
new ArrayList<? extends Integer>(); // interdit
new List<? extends Integer>[10]; // interdit
new List<?>[10]; // autorisé
```

- Le compilateur impose des règles complémentaires pour éviter des anomalies
- *Exemple : supposons que B soit sous-type de A*

```
ArrayList<? extends A>; l = new ArrayList<B>();
l.add(new A()); // interdit
```


Conclusion

- La généricité apporte souplesse et robustesse dans le code
- La généricité apporte une complexité importante
  - Utiliser les classes génériques existantes n'est pas compliqué
  - Créer ses propres classes génériques est plus difficile
- Le compilateur se charge normalement de détecter la plupart des problèmes (en particulier ceux, nombreux, qui n'ont pas été abordés dans ce cours)
- La généricité est plus simple quand elle est prévue au départ (ADA, C++...)

Autres nouveautés de Java 1.5 (1/2)

- Nombre variable d'arguments :
  - Un seul argument variable par méthode, toujours en fin de liste d'arguments
  - On peut substituer un tableau à l'argument

```
public void myMethod(String t, Object... arg){
    for(int i=0;i<arg.length;i++){
        ...
    }
}

...

myMethod("truc","un","deux","trois",4);
```

- Autoboxing : faciliter le passage entre types primitifs et instances

```
Integer i = 12; // autoboxing

int j = i; // autounboxing
```

- Boucle **`for`** étendue : peut s'utiliser sur toute classe implémentant **`Iterable`**

```
Collection c = new LinkedList();
...
for(Object o : c){
    ...
}
```


Autres nouveautés de Java 1.5 (2/2)

- Enumérations : un nouveau type (chaque enumération produit une classe dans la JVM)
  - **`values()`** renvoie un tableau des valeurs, **`valueOf(String)`** renvoie la valeur

```
enum MyEnum {LUNDI,MARDI,MERCREDI,JEUDI,VENDREDI,SAMEDI,DIMANCHE};

...

MyEnum e = MyEnum.MARDI;

for(int i = 0;i<MyEnum.values().length;i++)
    System.out.println(MyEnum.values()[i]);

for(MyEnum f : MyEnum.values())
    System.out.println(f);
```

- Import static :

```
import static java.lang.Math;
...
double d = PI; // pas besoin de préfixer PI par Math
```

- Amélioration de la javadoc