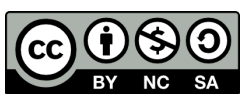
Les génériques

Conception Orientée Objet

Clément Quinton
Licence mention Informatique
Université de Lille



Génériques

```
public class Cup {
  private Object object;

  public void add(Object object) {
    this.object = object;
  }

  public Object get() {
    return object;
  }
}

public class Tea {
  public String toString() {
    return "Tea";
  }
}

public class Coffee {
  public String toString() {
    return "Coffee";
  }
}
```

```
public class Main {

  public static void main(String[] args) {
    Cup cup = new Cup();
    cup.add(new Tea());
  }
}
```

Génériques

```
public class Cup {
  private Object object;

  public void add(Object object) {
    this.object = object;
  }

  public Object get() {
    return object;
  }
}

public class Tea {
  public String toString() {
    return "Tea";
  }
}

public class Coffee {
  public String toString() {
    return "Coffee";
  }
}
```

```
public class Main {

  public static void main(String[] args) {
    Cup cup = new Cup();
    cup.add(new Tea());
    cup.add(new Coffee());
  }
}
```

Génériques

```java
public class Cup {
  private Object object;

  public void add(Object object) {
    this.object = object;
  }

  public Object get() {
    return object;
  }
}

public class Tea {
  public String toString() {
    return "Tea";
  }
}

public class Coffee {
  public String toString() {
    return "Coffee";
  }
}
```

```java
public class Main {

  public static void main(String[] args) {
    Cup cup = new Cup();
    cup.add(new Tea());
    cup.add(new Coffee()); // Compile
  }
}
```

Génériques

```
public class Cup {
  private Object object;

  public void add(Object object) {
    this.object = object;
  }

  public Object get() {
    return object;
  }
}

public class Tea {
  public String toString() {
    return "Tea";
  }
}

public class Coffee {
  public String toString() {
    return "Coffee";
  }
}
```

```
public class Main {

  public static void main(String[] args) {
    Cup cup = new Cup();
    cup.add(new Tea());
    cup.add(new Coffee()); // Compile
    Tea obj = (Tea) cup.get();
  }
}
```

Génériques

```java
public class Cup {
  private Object object;

  public void add(Object object) {
    this.object = object;
  }

  public Object get() {
    return object;
  }
}

public class Tea {
  public String toString() {
    return "Tea";
  }
}

public class Coffee {
  public String toString() {
    return "Coffee";
  }
}
```

```java
public class Main {

  public static void main(String[] args) {
    Cup cup = new Cup();
    cup.add(new Tea());
    cup.add(new Coffee()); // Compile
    Tea obj = (Tea) cup.get(); // Compile
  }
}
```

Génériques

```java
public class Cup {
  private Object object;

  public void add(Object object) {
    this.object = object;
  }

  public Object get() {
    return object;
  }
}

public class Tea {
  public String toString() {
    return "Tea";
  }
}

public class Coffee {
  public String toString() {
    return "Coffee";
  }
}
```

```java
public class Main {

  public static void main(String[] args) {
    Cup cup = new Cup();
    cup.add(new Tea());
    cup.add(new Coffee()); // Compile
    Tea obj = (Tea) cup.get(); // Compile
    // java.lang.ClassCastException
  }
}
```

Génériques

```java
public class Cup {
  private Object object;

  public void add(Object object) {
    this.object = object;
  }

  public Object get() {
    return object;
  }
}

public class Tea {
  public String toString() {
    return "Tea";
  }
}

public class Coffee {
  public String toString() {
    return "Coffee";
  }
}
```

```java
public class Main {

  public static void main(String[] args) {
    Cup cup = new Cup();
    cup.add(new Tea());
    cup.add(new Coffee()); // Compile
    Tea obj = (Tea) cup.get(); // Compile
    // java.lang.ClassCastException
  }
}
```

- “Il suffit de déclarer *object* de type Tea”

Génériques

```
public class Cup {
  private Object object;

  public void add(Object object) {
    this.object = object;
  }

  public Object get() {
    return object;
  }
}

public class Tea {
  public String toString() {
    return "Tea";
  }
}

public class Coffee {
  public String toString() {
    return "Coffee";
  }
}
```

```
public class Main {

  public static void main(String[] args) {
    Cup cup = new Cup();
    cup.add(new Tea());
    cup.add(new Coffee()); // Compile
    Tea obj = (Tea) cup.get(); // Compile
    // java.lang.ClassCastException
  }
}
```

- "Il suffit de déclarer *object* de type Tea"
- *Cup* pas très utile, ne peut contenir que du thé...


Cup<T>

Génériques

```java
public class Cup <T> {
  private T content;

  public void add(T content) {
    this.content = content;
  }

  public T get() {
    return content;
  }
}

public class Tea {
  public String toString() {
    return "Tea";
  }
}

public class Coffee {
  public String toString() {
    return "Coffee";
  }
}
```

```java
public class Main {

  public static void main(String[] args) {
    Cup cup = new Cup();
    cup.add(new Tea());
    cup.add(new Coffee()); // Compile
    Tea obj = (Tea) cup.get(); // Compile
    // java.lang.ClassCastException
  }
}
```

```java
public class Main {

  public static void main(String[] args) {
    Cup<Tea> cup = new Cup<Tea>();
  }
}
```

Génériques

```java
public class Cup <T> {
  private T content;

  public void add(T content) {
    this.content = content;
  }

  public T get() {
    return content;
  }
}

public class Tea {
  public String toString() {
    return "Tea";
  }
}

public class Coffee {
  public String toString() {
    return "Coffee";
  }
}
```

```java
public class Main {

  public static void main(String[] args) {
    Cup cup = new Cup();
    cup.add(new Tea());
    cup.add(new Coffee()); // Compile
    Tea obj = (Tea) cup.get(); // Compile
    // java.lang.ClassCastException
  }
}
```

```java
public class Main {

  public static void main(String[] args) {
    Cup<Tea> cup = new Cup<Tea>();
    cup.add(new Tea());
    System.out.println(cup.get()); // "Tea"
  }
}
```

Génériques

```java
public class Cup <T> {
  private T content;

  public void add(T content) {
    this.content = content;
  }

  public T get() {
    return content;
  }
}

public class Tea {
  public String toString() {
    return "Tea";
  }
}

public class Coffee {
  public String toString() {
    return "Coffee";
  }
}
```

```java
public class Main {

  public static void main(String[] args) {
    Cup cup = new Cup();
    cup.add(new Tea());
    cup.add(new Coffee()); // Compile
    Tea obj = (Tea) cup.get(); // Compile
    // java.lang.ClassCastException
  }
}
```

```java
public class Main {

  public static void main(String[] args) {
    Cup<Tea> cup = new Cup<Tea>();
    cup.add(new Tea());
    System.out.println(cup.get()); // "Tea"
    cup.add(new Coffee());
  }
}
```

Génériques

```java
public class Cup <T> {
  private T content;

  public void add(T content) {
    this.content = content;
  }

  public T get() {
    return content;
  }
}

public class Tea {
  public String toString() {
    return "Tea";
  }
}

public class Coffee {
  public String toString() {
    return "Coffee";
  }
}
```

```java
public class Main {

  public static void main(String[] args) {
    Cup cup = new Cup();
    cup.add(new Tea());
    cup.add(new Coffee()); // Compile
    Tea obj = (Tea) cup.get(); // Compile
    // java.lang.ClassCastException
  }
}
```

```java
public class Main {

  public static void main(String[] args) {
    Cup<Tea> cup = new Cup<Tea>();
    cup.add(new Tea());
    System.out.println(cup.get()); // "Tea"
    cup.add(new Coffee()); // Ne compile pas
  }
}
```

Fil rouge

?

On souhaite définir divers services de location : de voitures, de chambres, de vidéos, ...

- on peut ajouter un nouvel élément à louer au service ;
- un utilisateur peut louer l'un des objets ;
- on peut rendre un objet loué ;
- etc.

Première tentative : 3 classes spécifiques à chaque type

Première tentative : 3 classes spécifiques à chaque type

Constat : les codes sont identiques au type des données près

Partage de code ? Comment ?

Première tentative : 3 classes spécifiques à chaque type

Constat : les codes sont identiques au type des données près

Partage de code ? Comment ?

Ce qui varie c'est le type des données gérées

Première tentative : 3 classes spécifiques à chaque type

Constat : les codes sont identiques au type des données près

Partage de code ? Comment ?

Ce qui varie c'est le type des données gérées

Il faut donc **paramétrer le type des données**

Classes génériques

Classes génériques

=

Des classes paramétrées par un (ou plusieurs) **types paramètres**

Classes génériques

Classes génériques

=

Des classes paramétrées par un (ou plusieurs) **types paramètres**

- Création
  La variable de type, « `T` », apparaît dans la déclaration de classe.

```
public class Renter<T> { ...
```

- Il est possible d'avoir plusieurs types paramètres

```
Map<K,V>
```

- Le type paramètre peut être instancié à la création d'instance

```
new Renter<Car>
new Cup<Tea>
```

Tous les types sont possibles comme valeur de `T`

- Le type paramètre peut être instancié à la création d'instance

```
new Renter<Car>
new Cup<Tea>
```

  Tous les types sont possibles comme valeur de `T`

- Les sous-classes peuvent fixer le type paramètre

```
public class Hotel extends Renter<Room> { ...
public class CupOfTea extends Cup<Tea> { ...
```

- Le type paramètre peut être instancié à la création d'instance

```
new Renter<Car>
new Cup<Tea>
```

  Tous les types sont possibles comme valeur de `T`

- Les sous-classes peuvent fixer le type paramètre

```
public class Hotel extends Renter<Room> { ...
public class CupOfTea extends Cup<Tea> { ...
```

  La méthode `add` dans `Hotel` devient alors

```
public void add(Room good) { ...
```

- Le type paramètre peut être instancié à la création d'instance

```
new Renter<Car>
new Cup<Tea>
```

  Tous les types sont possibles comme valeur de `T`

- Les sous-classes peuvent fixer le type paramètre

```
public class Hotel extends Renter<Room> { ...
public class CupOfTea extends Cup<Tea> { ...
```

  La méthode `add` dans `Hotel` devient alors

```
public void add(Room good) { ...
```

- Mais elles ne sont pas obligées

```
public class SpecificRenter<T> extends Renter<T> { ...
```

Utilisation

```
public class Renter<T> {
  protected List<T> available;
  public Renter() {
    this.available = new ArrayList<T>();
  }
  public void add(T good) {
    this.available.add(good);
  }
  public T rentTo(User u) { ...  }
}
```

Utilisation

```
public class Renter<T> {
  protected List<T> available;
  public Renter() {
    this.available = new ArrayList<T>();
  }
  public void add(T good) {
    this.available.add(good);
  }
  public T rentTo(User u) { ...  }
}
```

```
public static void main(String[] args) {
  Renter<Car> carRenter = new Renter<Car>();
  carRenter.add(new Car());
  Car c = carRenter.rentTo(new User());
}
```

?

Dans `User` : *méthode `storeRented` pour, lors d'une location, mémoriser pour chaque bien loué son loueur*

- Quelle signature ?
- Quel type pour la structure de données de mémorisation ?
- Comment exprimer « n'importe quel `Renter` » ?

« *wildcard* »

Renter<?>

```
protected void storeRented(Renter<?> renter, Object rented) { ...

    private Map<Object, Renter<?>> rentedGoods;
```

? Dans `User` : *une méthode `rentSomething` qui permet d'obtenir un bien loué auprès d'un `Renter`*

Quelle signature ? Quelle valeur de retour ?

Le type de la valeur de retour, dépend du type paramètre du `Renter`

?

Dans `User` : *une méthode `rentSomething` qui permet d'obtenir un bien loué auprès d'un `Renter`*

Quelle signature ? Quelle valeur de retour ?

Le type de la valeur de retour, dépend du type paramètre du `Renter`

Méthode générique

?

Dans `User` : *une méthode `rentSomething` qui permet d'obtenir un bien loué auprès d'un `Renter`*

Quelle signature ? Quelle valeur de retour ?

Le type de la valeur de retour, dépend du type paramètre du `Renter`

Méthode générique

```
public <T> T rentSomething(Renter<T> renter) { ...
```

? Dans `User` : *une méthode `rentSomething` qui permet d'obtenir un bien loué auprès d'un `Renter`*

Quelle signature ? Quelle valeur de retour ?

Le type de la valeur de retour, dépend du type paramètre du `Renter`

Méthode générique

```
public <T> T rentSomething(Renter<T> renter) { ...
```

`<T>` déclare la méthode `rentSomething` générique avec un paramètre, `T`.
Le type de retour est `T`. Sans le `<T>` initial, le symbole `T` serait *undefined*

?

`Renter`, méthode `rentTo` : lors de la location, s'assurer que le `User` satisfait des conditions, propres à la catégorie d'objet louable :

- avoir le permis pour les voitures
- avoir 18 ans pour louer une chambre d'hôtel
- ...

?

`Renter`, méthode `rentTo` : lors de la location, s'assurer que le `User` satisfait des conditions, propres à la catégorie d'objet louable :

- avoir le permis pour les voitures
- avoir 18 ans pour louer une chambre d'hôtel
- ...

Nécessité pour les objets louables de partager un type

?

`Renter`, méthode `rentTo` : lors de la location, s'assurer que le `User` satisfait des conditions, propres à la catégorie d'objet louable :

- avoir le permis pour les voitures
- avoir 18 ans pour louer une chambre d'hôtel
- ...

Nécessité pour les objets louables de partager un type

Ajout de l'interface `Rentable`

Contraindre/borner le type paramètre

Restreindre les valeurs que peut prendre le type paramètre

Contraindre/borner le type paramètre

Restreindre les valeurs que peut prendre le type paramètre

Borne supérieure

```
? extends T
```

```
public class Renter<T extends Rentable> { ...
```

?

Dans `Renter` : *méthode `addList` qui permet d'ajouter une liste de nouveaux bien louables*

?

Dans `Renter` : *méthode `addList` qui permet d'ajouter une liste de nouveaux bien louables*

`Car` hérite de `Vehicle`

mais

`List<Car>` **n'est pas** un sous-type de `List<Vehicle>`

Dans `Renter<Vehicle>` on doit pouvoir ajouter
n'importe quelle liste qui contient des objets de type `Vehicle`.

Plus généralement,

dans `Renter<T>` on doit pouvoir ajouter
n'importe quelle liste qui contient des objets de type `T`.

Dans `Renter<Vehicle>` on doit pouvoir ajouter
n'importe quelle liste qui contient des objets de type `Vehicle`.

Plus généralement,

dans `Renter<T>` on doit pouvoir ajouter
n'importe quelle liste qui contient des objets de type `T`.

= n'importe quelle liste dont le type des éléments est un **sous-type de `T`**

Dans `Renter<Vehicle>` on doit pouvoir ajouter
n'importe quelle liste qui contient des objets de type `Vehicle`.

Plus généralement,

dans `Renter<T>` on doit pouvoir ajouter
n'importe quelle liste qui contient des objets de type `T`.

= n'importe quelle liste dont le type des éléments est un **sous-type de `T`**

sous-type de `T` = `? extends T`

Dans `Renter<Vehicle>` on doit pouvoir ajouter
n'importe quelle liste qui contient des objets de type `Vehicle`.

Plus généralement,

dans `Renter<T>` on doit pouvoir ajouter
n'importe quelle liste qui contient des objets de type `T`.

= n'importe quelle liste dont le type des éléments est un **sous-type de `T`**

sous-type de `T` = **`? extends T`**

```
public void addList(List<? extends T> goods) { ...
```

?

Dans `Renter` : *méthode `transferAllGoodsTo` qui permet de transférer tous les objets louables à un autre objet* `Renter`

Dans `Renter` : *méthode `transferAllGoodsTo` qui permet de transférer tous les objets louables à un autre objet* `Renter`

Un `Renter<T>` doit pouvoir transférer à tout `Renter` capable de louer des objets de type `T`

Qui peut le plus peut le moins

?

Dans `Renter` : *méthode `transferAllGoodsTo` qui permet de transférer tous les objets louables à un autre objet* `Renter`

Un `Renter<T>` doit pouvoir transférer à tout `Renter` capable de louer des objets de type `T`

Qui peut le plus peut le moins

Tout `Renter` capable de louer des objets d'**un super-type de `T`** peut louer des objets de type `T`

Borne inférieure

```
? super T
```

```
public void transferAllGoodsTo(Renter<? super T> other) { ...
```

?

Dans `Renter` : *méthode `addGoodsAndTransferDuplicate` qui ajoute à un `Renter` tous les objets (louables) d'une liste s'ils ne sont pas déjà gérés par ce `Renter`, et transfère les doublons à un autre `Renter` fourni*

?

Dans `Renter` : *méthode `addGoodsAndTransferDuplicate` qui ajoute à un `Renter` tous les objets (louables) d'une liste s'ils ne sont pas déjà gérés par ce `Renter`, et transfère les doublons à un autre `Renter` fourni*

```
public void addGoodsAndTransferDuplicate
        (List<? extends T> goods, Renter<? super T> other) {
```

PECS

PECS

syntaxe	interprétation
`?`	n'importe quel type
`? extends T`	n'importe quel sous-type de T
`? super T`	n'importe quel super-type de T

règle PECS : "Producer `extends`, Consumer `super`"

Exemples :

- dans `java.util.List<E>` :

```
boolean addAll(Collection<? extends E> c)
```

- dans `java.util.Collections` :

```
static <T> void sort(List<T> list, Comparator<? super T> c)
```

- dans `java.util.Collections` :

```
static <T> void copy(List<? super T> d, List<? extends T> s)
```

?

Définir un type de `Renter` qui ne gèrent que des véhicules électriques.

?

Définir un type de `Renter` qui ne gèrent que des véhicules électriques.

La contrainte sur le type est d'être à la fois un sous-type de `Vehicle` et un sous-type de `Electric`

? *Définir un type de `Renter` qui ne gèrent que des véhicules électriques.*

La contrainte sur le type est d'être à la fois un sous-type de `Vehicle` et un sous-type de `Electric`

Bornes multiples

```
T extends Vehicle & Electric
```

```
public class ElectricVehicleRenter<T extends Vehicle & Electric>
        extends Renter<T> { ...
```

`ElectricVehicleRenter` ne permet pas de créer un type qui mélange `ElectricCar` et `ElectricScooter`

cf `ElectricVehicleRenterManager`