Les génériques

Conception Orientée Objet

Jean-Christophe Routier
Licence mention Informatique
Université Lille – Sciences et Technologies

Université de Lille

Faculté des sciences et technologies
Département Informatique

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 des loueurs sont identiques au type des données près
- dans `User` des méthodes pour chaqaue type de loueur...

partage de code ?

première tentative : 3 classes spécifiques à chaque type

constat :

- les codes des loueurs sont identiques au type des données près
- dans `User` des méthodes pour chaqaue type de loueur...

partage de code ?

ce qui varie c'est le type des données gérées

première tentative : 3 classes spécifiques à chaque type

constat :

- les codes des loueurs sont identiques au type des données près
- dans `User` des méthodes pour chaqaue type de loueur...

partage de code ?

ce qui varie c'est le type des données gérées

nécessité de **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>
```

  tous les types sont possibles comme valeur de `T`

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

```
new Renter<Car>
```

  tous les types sont possibles comme valeur de `T`

- les sous-classes peuvent fixer le type paramètre

```
public class Hotel extends Renter<Room> { ...
```

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

```
new Renter<Car>
```

  tous les types sont possibles comme valeur de `T`

- les sous-classes peuvent fixer le type paramètre

```
public class Hotel extends Renter<Room> { ...
```

  la méthode héritée `add` « devient » alors pour cette classe

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

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

```
new Renter<Car>
```

  tous les types sont possibles comme valeur de `T`

- les sous-classes peuvent fixer le type paramètre

```
public class Hotel extends Renter<Room> { ...
```

  la méthode héritée add « devient » alors pour cette classe

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

- mais elles ne sont pas obligées

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

effacement de type

type erasure on `docs.oracle.com`

Generics were introduced to the Java language ***to provide tighter type checks at compile time*** *and to support generic programming. To implement generics,* ***the Java compiler applies type erasure*** *to:*

- ***Replace all type parameters*** *in generic types with their bounds or* `Object` *if the type parameters are unbounded. The produced bytecode, therefore, contains only ordinary classes, interfaces, and methods.*
- ***Insert type casts*** *if necessary to preserve type safety.*
- ***Generate bridge methods to preserve polymorphism*** *in extended generic types.*

Type erasure ensures that ***no new classes are created*** *for parameterized types; consequently, generics incur no runtime overhead.*

```
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) { ...  }
}
```

devient après « *type erasure* »

```
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) { ...  }
}
```

devient après « *type erasure* »

```
public class Renter {
  protected List available;
  public Renter() {
    this.available = new ArrayList();
  }
  public void add(Object good) {
    this.available.add(good);
  }
  public Object 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());
}
```

devient après « *type erasure* »

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

devient après « *type erasure* »

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

```
public class Hotel extends Renter<Room> {
  // surcharge de Renter<T>.add(T good)
  public void add(Room good) {
    System.out.println("do something here");
    super.add(good);
  }
}
```

devient après « *type erasure* »

```
public class Hotel extends Renter<Room> {
  // surcharge de Renter<T>.add(T good)
  public void add(Room good) {
    System.out.println("do something here");
    super.add(good);
  }
}
```

devient après « *type erasure* »

```
public class Hotel extends Renter {
  // ce n'est pas une surcharge de Renter.add(Object good)
  public void add(Room good) {
    System.out.println("do something here");
    super.add(good);
  }
```

```
public class Hotel extends Renter<Room> {
  // surcharge de Renter<T>.add(T good)
  public void add(Room good) {
    System.out.println("do something here");
    super.add(good);
  }
}
```

devient après « *type erasure* »

```
public class Hotel extends Renter {
  // ce n'est pas une surcharge de Renter.add(Object good)
  public void add(Room good) {
    System.out.println("do something here");
    super.add(good);
  }
  // << bridge method >> ajoutée par le compilateur
  public void add(Object good) {
    this.add((Room) good);
  }
}
```

?

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` » ?

?

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<?>
```

?

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 ?

? 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) { ...
```

?

`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> { ...
```

NB : lors du « *type erasure* » `T` est remplacé par `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`

? 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 transfert 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 transfert 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`