

Mise à Niveau Java Eclipse

Gilles Ardourel

Département Informatique
Université de Nantes

Introduction

Besoins du génie logiciel
Programmation Objet
Diagrammes
Java

Introduction

Besoins du Génie Logiciel
Programmation Objet
Diagrammes
Java

- ▶ Augmenter la fiabilité
 - ▶ vérifications a priori
 - ▶ limiter l'impact des erreurs
- ▶ Réduire le *time to Market*
 - ▶ facilité de conception
 - ▶ réutilisation
- ▶ Réduire le coût de maintenance
 - ▶ flexibilité
 - ▶ documentation

- ▶ Augmenter la fiabilité
 - ▶ vérifications a priori
 - ▶ limiter l'impact des erreurs
- ▶ Réduire le *time to Market*
 - ▶ facilité de conception
 - ▶ réutilisation
- ▶ Réduire le coût de maintenance
 - ▶ flexibilité
 - ▶ documentation

- ▶ Augmenter la fiabilité
 - ▶ vérifications a priori
 - ▶ limiter l'impact des erreurs
- ▶ Réduire le *time to Market*
 - ▶ facilité de conception
 - ▶ réutilisation
- ▶ Réduire le coût de maintenance
 - ▶ flexibilité
 - ▶ documentation

- ▶ Augmenter la fiabilité
 - ▶ vérifications a priori
 - ▶ limiter l'impact des erreurs
- ▶ Réduire le *time to Market*
 - ▶ facilité de conception
 - ▶ réutilisation
- ▶ Réduire le coût de maintenance
 - ▶ flexibilité
 - ▶ documentation

- ▶ Augmenter la fiabilité
 - ▶ vérifications a priori
 - ▶ limiter l'impact des erreurs
- ▶ Réduire le *time to Market*
 - ▶ facilité de conception
 - ▶ réutilisation
- ▶ Réduire le coût de maintenance
 - ▶ flexibilité
 - ▶ documentation

- ▶ Augmenter la fiabilité
 - ▶ vérifications a priori
 - ▶ limiter l'impact des erreurs
- ▶ Réduire le *time to Market*
 - ▶ facilité de conception
 - ▶ réutilisation
- ▶ Réduire le coût de maintenance
 - ▶ flexibilité
 - ▶ documentation

Réponses de la programmation

- ▶ Augmenter la fiabilité
 - ▶ vérifications a priori
 - ▶ limiter l'impact des erreurs
- ▶ Réduire le *time to Market*
 - ▶ facilité de conception
 - ▶ réutilisation
- ▶ Réduire le coût de maintenance
 - ▶ flexibilité
 - ▶ documentation

Typage
Modularité, Exceptions

Modèle Objet
Héritage, Modularité

Sous typage
Méthode Objet

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Réponses de la programmation

- ▶ Augmenter la fiabilité
 - ▶ vérifications a priori
 - ▶ limiter l'impact des erreurs
- ▶ Réduire le *time to Market*
 - ▶ facilité de conception
 - ▶ réutilisation
- ▶ Réduire le coût de maintenance
 - ▶ flexibilité
 - ▶ documentation

Typage
Modularité, Exceptions

Modèle Objet
Héritage, Modularité

Sous typage
Méthode Objet

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Réponses de la programmation

- ▶ Augmenter la fiabilité
 - ▶ vérifications a priori
 - ▶ limiter l'impact des erreurs
- ▶ Réduire le *time to Market*
 - ▶ facilité de conception
 - ▶ réutilisation
- ▶ Réduire le coût de maintenance
 - ▶ flexibilité
 - ▶ documentation

Typage
Modularité, Exceptions

Modèle Objet
Héritage, Modularité

Sous typage
Méthode Objet

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Réponses de la programmation

- ▶ Augmenter la fiabilité
 - ▶ vérifications a priori
 - ▶ limiter l'impact des erreurs
- ▶ Réduire le *time to Market*
 - ▶ facilité de conception
 - ▶ réutilisation
- ▶ Réduire le coût de maintenance
 - ▶ flexibilité
 - ▶ documentation

Typage
Modularité, Exceptions

Modèle Objet
Héritage, Modularité

Sous typage
Méthode Objet

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Réponses de la programmation

- ▶ Augmenter la fiabilité
 - ▶ vérifications a priori
 - ▶ limiter l'impact des erreurs
- ▶ Réduire le *time to Market*
 - ▶ facilité de conception
 - ▶ réutilisation
- ▶ Réduire le coût de maintenance
 - ▶ flexibilité
 - ▶ documentation

Typage
Modularité, Exceptions

Modèle Objet
Héritage, Modularité

Sous typage
Méthode Objet

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Réponses de la programmation

- ▶ Augmenter la fiabilité
 - ▶ vérifications a priori
 - ▶ limiter l'impact des erreurs
- ▶ Réduire le *time to Market*
 - ▶ facilité de conception
 - ▶ réutilisation
- ▶ Réduire le coût de maintenance
 - ▶ flexibilité
 - ▶ documentation

Typage
Modularité, Exceptions

Modèle Objet
Héritage, Modularité

Sous typage
Méthode Objet

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Réponses de la programmation

► Augmenter la fiabilité

- vérifications a priori
- limiter l'impact des erreurs

Typage
Modularité, Exceptions

► Réduire le *time to Market*

- facilité de conception
- réutilisation

Modèle Objet
Héritage, Modularité

► Réduire le coût de maintenance

- flexibilité
- documentation

Sous typage
Méthode Objet

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Réponses de l'environnement

- ▶ Augmenter la fiabilité
 - ▶ vérifications a priori
 - ▶ limiter l'impact des erreurs
- ▶ Réduire le *time to Market*
 - ▶ facilité de conception
 - ▶ réutilisation
- ▶ Réduire le coût de maintenance
 - ▶ flexibilité
 - ▶ documentation

Compilation incrémentale
Warnings (encapsulation...)

Génération de code
Assistants

Génération commentaires

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Réponses de l'environnement

- ▶ Augmenter la fiabilité
 - ▶ vérifications a priori
 - ▶ limiter l'impact des erreurs
- ▶ Réduire le *time to Market*
 - ▶ facilité de conception
 - ▶ réutilisation
- ▶ Réduire le coût de maintenance
 - ▶ flexibilitéRépercution MàJ
 - ▶ documentation

Compilation incrémentale
Warnings (encapsulation...)

Génération de code
Assistants

Génération commentaires

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Réponses de l'environnement

- ▶ Augmenter la fiabilité
 - ▶ vérifications a priori
 - ▶ limiter l'impact des erreurs
- ▶ Réduire le *time to Market*
 - ▶ facilité de conception
 - ▶ réutilisation
- ▶ Réduire le coût de maintenance
 - ▶ flexibilité
 - ▶ documentation

Compilation incrémentale
Warnings (encapsulation...)

Génération de code
Assistants

Génération commentaires

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Réponses de l'environnement

► Augmenter la fiabilité

- vérifications a priori
- limiter l'impact des erreurs

Compilation incrémentale
Warnings (encapsulation...)

► Réduire le *time to Market*

- facilité de conception
- réutilisation

Génération de code
Assistants

► Réduire le coût de maintenance

- flexibilité Répercussion MàJ
- documentation

Génération commentaires

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Réponses de l'environnement

► Augmenter la fiabilité

- vérifications a priori
- limiter l'impact des erreurs

Compilation incrémentale
Warnings (encapsulation...)

► Réduire le *time to Market*

- facilité de conception
- réutilisation

Génération de code
Assistants

► Réduire le coût de maintenance

- flexibilité Répercussion MàJ
- documentation

Génération commentaires

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Réponses de l'environnement

► Augmenter la fiabilité

- vérifications a priori
- limiter l'impact des erreurs

Compilation incrémentale
Warnings (encapsulation...)

► Réduire le *time to Market*

- facilité de conception
- réutilisation

Génération de code
Assistants

► Réduire le coût de maintenance

- flexibilitéRépercution MàJ
- documentation

Génération commentaires

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Réponses de l'environnement

► Augmenter la fiabilité

- vérifications a priori
- limiter l'impact des erreurs

Compilation incrémentale
Warnings (encapsulation...)

► Réduire le *time to Market*

- facilité de conception
- réutilisation

Génération de code
Assistants

► Réduire le coût de maintenance

- flexibilitéRépercution MàJ
- documentation

Génération commentaires

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Java, avantages

- ▶ Langage à Objets
- ▶ Encapsulation des structures et comportements
- ▶ Représentation par classes
- ▶ Regroupement en packages
- ▶ Héritage simple et typage multiple
- ▶ Liaison dynamique
- ▶ Exceptions
- ▶ Compilé vers bytecode pour JVM
- ▶ Garbage Collector
- ▶ Généricité paramétrique (enfin !)
- ▶ Bibliothèques nombreuses
- ▶ Environnements de développement

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

- ▶ Langage à Objets
 - ▶ Encapsulation des structures et comportements
 - ▶ Représentation par classes
 - ▶ Regroupement en packages
 - ▶ Héritage simple et typage multiple
 - ▶ Liaison dynamique
 - ▶ Exceptions
 - ▶ Compilé vers bytecode pour JVM
 - ▶ Garbage Collector
 - ▶ Généricité paramétrique (enfin !)
 - ▶ Bibliothèques nombreuses
 - ▶ Environnements de développement

- ▶ Langage à Objets
- ▶ Encapsulation des structures et comportements
 - ▶ Représentation par classes
 - ▶ Regroupement en packages
 - ▶ Héritage simple et typage multiple
 - ▶ Liaison dynamique
 - ▶ Exceptions
 - ▶ Compilé vers bytecode pour JVM
 - ▶ Garbage Collector
 - ▶ Généricité paramétrique (enfin !)
 - ▶ Bibliothèques nombreuses
 - ▶ Environnements de développement

- ▶ Langage à Objets
- ▶ Encapsulation des structures et comportements
- ▶ Représentation par classes
- ▶ Regroupement en packages
- ▶ Héritage simple et typage multiple
- ▶ Liaison dynamique
- ▶ Exceptions
- ▶ Compilé vers bytecode pour JVM
- ▶ Garbage Collector
- ▶ Généricité paramétrique (enfin !)
- ▶ Bibliothèques nombreuses
- ▶ Environnements de développement

- ▶ Langage à Objets
- ▶ Encapsulation des structures et comportements
- ▶ Représentation par classes
- ▶ Regroupement en packages
- ▶ Héritage simple et typage multiple
- ▶ Liaison dynamique
- ▶ Exceptions
- ▶ Compilé vers bytecode pour JVM
- ▶ Garbage Collector
- ▶ Généricité paramétrique (enfin !)
- ▶ Bibliothèques nombreuses
- ▶ Environnements de développement

- ▶ Langage à Objets
- ▶ Encapsulation des structures et comportements
- ▶ Représentation par classes
- ▶ Regroupement en packages
- ▶ Héritage simple et typage multiple
- ▶ Liaison dynamique
- ▶ Exceptions
- ▶ Compilé vers bytecode pour JVM
- ▶ Garbage Collector
- ▶ Généricité paramétrique (enfin !)
- ▶ Bibliothèques nombreuses
- ▶ Environnements de développement

- ▶ Langage à Objets
- ▶ Encapsulation des structures et comportements
- ▶ Représentation par classes
- ▶ Regroupement en packages
- ▶ Héritage simple et typage multiple
- ▶ Liaison dynamique
- ▶ Exceptions
- ▶ Compilé vers bytecode pour JVM
- ▶ Garbage Collector
- ▶ Généricité paramétrique (enfin !)
- ▶ Bibliothèques nombreuses
- ▶ Environnements de développement

- ▶ Langage à Objets
- ▶ Encapsulation des structures et comportements
- ▶ Représentation par classes
- ▶ Regroupement en packages
- ▶ Héritage simple et typage multiple
- ▶ Liaison dynamique
- ▶ Exceptions
- ▶ Compilé vers bytecode pour JVM
- ▶ Garbage Collector
- ▶ Généricité paramétrique (enfin !)
- ▶ Bibliothèques nombreuses
- ▶ Environnements de développement

- ▶ Langage à Objets
- ▶ Encapsulation des structures et comportements
- ▶ Représentation par classes
- ▶ Regroupement en packages
- ▶ Héritage simple et typage multiple
- ▶ Liaison dynamique
- ▶ Exceptions
- ▶ Compilé vers bytecode pour JVM
- ▶ Garbage Collector
- ▶ Généricité paramétrique (enfin !)
- ▶ Bibliothèques nombreuses
- ▶ Environnements de développement

- ▶ Langage à Objets
- ▶ Encapsulation des structures et comportements
- ▶ Représentation par classes
- ▶ Regroupement en packages
- ▶ Héritage simple et typage multiple
- ▶ Liaison dynamique
- ▶ Exceptions
- ▶ Compilé vers bytecode pour JVM
- ▶ Garbage Collector
- ▶ Généricité paramétrique (enfin !)
- ▶ Bibliothèques nombreuses
- ▶ Environnements de développement

- ▶ Langage à Objets
- ▶ Encapsulation des structures et comportements
- ▶ Représentation par classes
- ▶ Regroupement en packages
- ▶ Héritage simple et typage multiple
- ▶ Liaison dynamique
- ▶ Exceptions
- ▶ Compilé vers bytecode pour JVM
- ▶ Garbage Collector
- ▶ Généricité paramétrique (enfin !)
- ▶ Bibliothèques nombreuses
- ▶ Environnements de développement

- ▶ Langage à Objets
- ▶ Encapsulation des structures et comportements
- ▶ Représentation par classes
- ▶ Regroupement en packages
- ▶ Héritage simple et typage multiple
- ▶ Liaison dynamique
- ▶ Exceptions
- ▶ Compilé vers bytecode pour JVM
- ▶ Garbage Collector
- ▶ Généricité paramétrique (enfin !)
- ▶ Bibliothèques nombreuses
- ▶ Environnements de développement

- ▶ Langage à Objets
- ▶ Encapsulation des structures et comportements
- ▶ Représentation par classes
- ▶ Regroupement en packages
- ▶ Héritage simple et typage multiple
- ▶ Liaison dynamique
- ▶ Exceptions
- ▶ Compilé vers bytecode pour JVM
- ▶ Garbage Collector
- ▶ Généricité paramétrique (enfin !)
- ▶ Bibliothèques nombreuses
- ▶ Environnements de développement

proche de C++ sans pointeurs et gestion explicite de l'allocation mémoire.

- ▶ déclaration de variable, d'attribut : `type nom ;`
`Personne toto ;`

- ▶ déclaration de méthode :
`typeretour nommethode(type1
param1, ...)`

- ▶ création d'objet toto instance de `Personne` :
`toto = new Personne() ;`

- ▶ qualification des attributs, methodes par :
`public, private, protected, static, ...`

- ▶ transtypage (cast) :
`(NouveauType)nomVariable`

Une classe Java

```
package gestionclub.core ;  
import java.util.Vector ;  
public class Personne {  
    private String nom ;  
    private Personne[] potes ;  
    public Personne()  
        {super() ;}  
  
    public void ditBonjour()  
        {System.out.println(this.nom) ;}
```

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

packages java

- ▶ java.lang -> classes de base
- ▶ java.util -> Structures, evenements
- ▶ java.math -> Fonction math
- ▶ java.io -> Gestion des flots entrées sorties
- ▶ ...

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Décomposition modulaire centrée sur les objets.
Communication par envoi de message.

- ▶ Proche de la "réalité"
- ▶ Facile à communiquer
- ▶ Résistant aux changements de fonctionnalités
- ▶ Extensible par de nouveaux objets
- ▶ Réutilisable

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Décomposition modulaire centrée sur les objets.
Communication par envoi de message.

- ▶ Proche de la "réalité"
- ▶ Facile à communiquer
- ▶ Résistant aux changements de fonctionnalités
- ▶ Extensible par de nouveaux objets
- ▶ Réutilisable

Décomposition modulaire centrée sur les objets.
Communication par envoi de message.

- ▶ Proche de la "réalité"
- ▶ Facile à communiquer
- ▶ Résistant aux changements de fonctionnalités
- ▶ Extensible par de nouveaux objets
- ▶ Réutilisable

Décomposition modulaire centrée sur les objets.
Communication par envoi de message.

- ▶ Proche de la "réalité"
- ▶ Facile à communiquer
- ▶ Résistant aux changements de fonctionnalités
- ▶ Extensible par de nouveaux objets
- ▶ Réutilisable

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Décomposition modulaire centrée sur les objets.
Communication par envoi de message.

- ▶ Proche de la "réalité"
- ▶ Facile à communiquer
- ▶ Résistant aux changements de fonctionnalités
- ▶ Extensible par de nouveaux objets
- ▶ Réutilisable

Décomposition modulaire centrée sur les objets.
Communication par envoi de message.

- ▶ Proche de la "réalité"
- ▶ Facile à communiquer
- ▶ Résistant aux changements de fonctionnalités
- ▶ Extensible par de nouveaux objets
- ▶ Réutilisable

Un Objet est caractérisé par

- ▶ un état (déterminé par la valeur de ses attributs)
- ▶ un comportement (déclenché par des messages et déterminé par les méthodes y correspondant)

Par exemple : mon véhicule est une vieille voiture verte

- ▶ état : (Couleur : vert) (condition : vieille)
- ▶ comportement : (demarrer : "faire un bruit inquiétant")

Un Objet est caractérisé par

- ▶ un état (déterminé par la valeur de ses attributs)
- ▶ un comportement (déclenché par des messages et déterminé par les méthodes y correspondant)

Par exemple : mon véhicule est une vieille voiture verte

- ▶ état : (Couleur : vert) (condition : vieille)
- ▶ comportement : (demarrer : "faire un bruit inquiétant")

Définit une catégorie d'objets ayant des caractéristiques communes (attributs et comportements)

- ▶ Ma voiture appartient à la classe véhicule
- ▶ C'est une *instance* de véhicule

Elle permet de créer des objets.

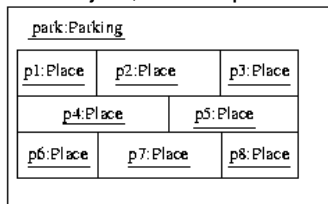
objets, classes et relations

Modélisation basée sur des objets, décrits par des classes.

v1:Véhicule

v2:Véhicule

v3:Véhicule



Introduction

Besoins du génie logiciel

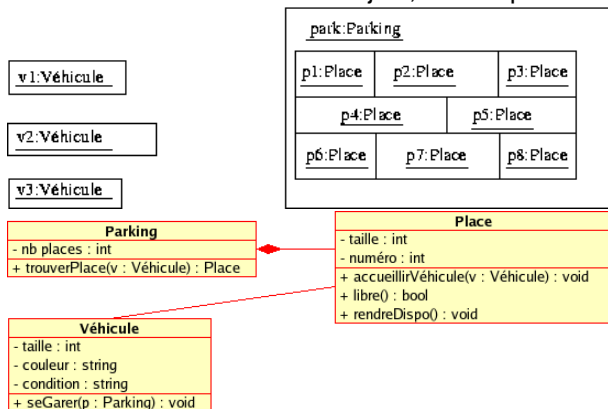
Programmation Objet

Diagrammes

Java

objets, classes et relations

Modélisation basée sur des objets, décrits par des classes.



Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Des messages aux méthodes

On envoie les messages aux objets au travers de références

- ▶ L'objet doit posséder une méthode correspondant
- ▶ Le code de la méthode est interprété dans le contexte de l'objet receveur

Introduction

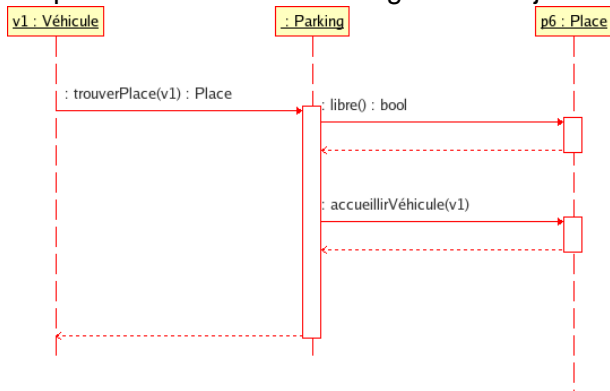
Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Séquence d'envois de message entre objets : un parking



Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Qui commence ?

Pour commencer il faut un programme principal qui

- ▶ crée les objets nécessaires
grâce aux constructeurs de leur classe
- ▶ leur envoie des messages au travers de références
En réaction desquels ils peuvent faire de même

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Qui commence ?

Pour commencer il faut un programme principal qui

- ▶ crée les objets nécessaires

grâce aux constructeurs de leur classe

- ▶ leur envoie des messages au travers de références

En réaction desquels ils peuvent faire de même

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Qui commence ?

Pour commencer il faut un programme principal qui

- ▶ crée les objets nécessaires
grâce aux constructeurs de leur classe
- ▶ leur envoie des messages au travers de références
En réaction desquels ils peuvent faire de même

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Qui commence ?

Pour commencer il faut un programme principal qui

- ▶ crée les objets nécessaires
grâce aux constructeurs de leur classe
- ▶ leur envoie des messages au travers de références

En réaction desquels ils peuvent faire de même

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Qui commence ?

Pour commencer il faut un programme principal qui

- ▶ crée les objets nécessaires
grâce aux constructeurs de leur classe
- ▶ leur envoie des messages au travers de références
En réaction desquels ils peuvent faire de même

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Les objets en mémoire

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

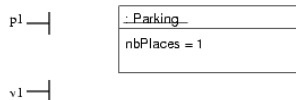
Java

p1 —|

v1 —|

```
public class Principal{  
    public static void main(String[] args){  
        Parking p1;  
        Vehicule v1;  
        p1 = new Parking();  
        v1 = new Vehicule("verte","vieille");  
        v1 = new Vehicule("rouge","neuve");  
        v1.seGarer(p1);  
    }  
}
```

Les objets en mémoire

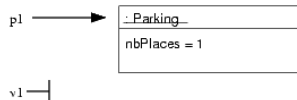


```
public class Principal{  
    public static void main(String[] args){  
        Parking p1;  
        Vehicule v1;  
        p1 = new Parking();  
        v1 = new Vehicule("verte","vieille");  
        v1 = new Vehicule("rouge","neuve");  
        v1.seGarer(p1);  
    }  
}
```

Introduction

Besoins du génie logiciel
Programmation Objet
Diagrammes
Java

Les objets en mémoire



```
public class Principal{  
    public static void main(String[] args){  
        Parking p1;  
        Vehicule v1;  
        p1 = new Parking();  
        v1 = new Vehicule("verte","vieille");  
        v1 = new Vehicule("rouge","neuve");  
        v1.seGarer(p1);  
    }  
}
```

Introduction

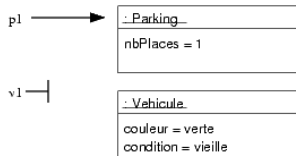
Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Les objets en mémoire

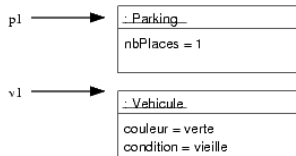


```
public class Principal{  
    public static void main(String[] args){  
        Parking p1;  
        Vehicule v1;  
        p1 = new Parking();  
        v1 = new Vehicule("verte","vieille");  
        v1 = new Vehicule("rouge","neuve");  
        v1.seGarer(p1);  
    }  
}
```

Introduction

Besoins du génie logiciel
Programmation Objet
Diagrammes
Java

Les objets en mémoire



```
public class Principal{
    public static void main(String[] args){
        Parking p1;
        Vehicule v1;
        p1 = new Parking();
        v1 = new Vehicule("verte","vieille");
        v1 = new Vehicule("rouge","neuve");
        v1.seGarer(p1);
    }
}
```

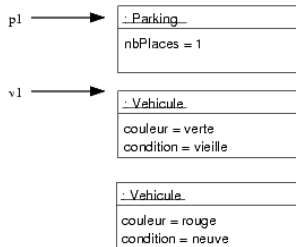
Introduction

Besoins du génie logiciel
Programmation Objet
Diagrammes
Java

Les objets en mémoire

Introduction

Besoins du génie logiciel
Programmation Objet
Diagrammes
Java

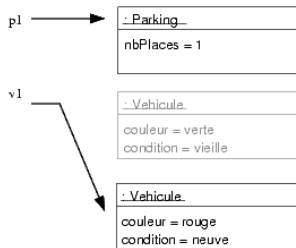


```
public class Principal{  
    public static void main(String[] args){  
        Parking p1;  
        Vehicule v1;  
        p1 = new Parking();  
        v1 = new Vehicule("verte","vieille");  
        v1 = new Vehicule("rouge","neuve");  
        v1.seGarer(p1);  
    }  
}
```

Les objets en mémoire

Introduction

Besoins du génie logiciel
Programmation Objet
Diagrammes
Java

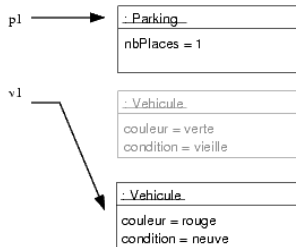


```
public class Principal{
    public static void main(String[] args){
        Parking p1;
        Vehicule v1;
        p1 = new Parking();
        v1 = new Vehicule("verte","vieille");
        v1 = new Vehicule("rouge","neuve");
        v1.seGarer(p1);
    }
}
```

Les objets en mémoire

Introduction

Besoins du génie logiciel
Programmation Objet
Diagrammes
Java



```
public class Principal{
    public static void main(String[] args){
        Parking p1 ;
        Vehicule v1 ;
        p1 = new Parking() ;
        v1 = new Vehicule("verte","vieille") ;
        v1 = new Vehicule("rouge","neuve") ;
        v1.seGarer(p1) ;
    }
}
```



```
public class Vehicule{
    private int taille;
    private String couleur;
    private String condition;
    public Vehicule(String coul, String cond){
        this.couleur =coul;
        this.condition =cond;
    }

    public void seGarer(Parking p){
        Place p2;
        p2 = p.accueillirVehicule(this);
    }
}
```

```
public class Vehicule{
    private int taille;
    private String couleur;
    private String condition;
    public Vehicule(String coul, String cond){
        this.couleur =coul;
        this.condition =cond;
    }

    public void seGarer(Parking p){
        Place p2;
        p2 = p.accueillirVehicule(this);
    }
}
```

```
public class Vehicule{
    private int taille;
    private String couleur;
    private String condition;
    public Vehicule(String coul, String cond){
        this.couleur =coul;
        this.condition =cond;
    }

    public void seGarer(Parking p){
        Place p2;
        p2 = p.accueillirVehicule(this);
    }
}
```

Classe Véhicule

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

```
public class Vehicule{
    private int taille;
    private String couleur;
    private String condition;
    public Vehicule(String coul, String cond){
        this.couleur =coul;
        this.condition =cond;
    }

    public void seGarer(Parking p){
        Place p2;
        p2 = p.accueillirVehicule(this);
    }
}
```

```
public class Vehicule{
    private int taille;
    private String couleur;
    private String condition;
    public Vehicule(String coul, String cond){
        this.couleur =coul;
        this.condition =cond;
    }

    public void seGarer(Parking p){
        Place p2;
        p2 = p.accueillirVehicule(this);
    }
}
```

Classe Véhicule

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

```
public class Vehicule{
    private int taille;
    private String couleur;
    private String condition;
    public Vehicule(String coul, String cond){
        this.couleur =coul;
        this.condition =cond;
    }

    public void seGarer(Parking p){
        Place p2;
        p2 = p.accueillirVehicule(this);
    }
}
```

Classe Véhicule

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

```
public class Vehicule{
    private int taille;
    private String couleur;
    private String condition;
    public Vehicule(String coul, String cond){
        this.couleur =coul;
        this.condition =cond;
    }

    public void seGarer(Parking p){
        Place p2;
        p2 = p.accueillirVehicule(this);
    }
}
```

Objectif : logiciels

- ▶ fiables
- ▶ maintenables
- ▶ réutilisables

Moyen : programmation

- ▶ Objet
- ▶ Modulaire (découplage)
- ▶ Orientée par la dynamique

Introduction

Besoins du génie logiciel

Programmation Objet

Diagrammes

Java

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage
généricité

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Type

Un type détermine un ensemble de messages applicables.

Lorsque

- ▶ une variable
- ▶ un retour de méthode
- ▶ un attribut

est déclaré d'un type T, l'objet référencé doit pouvoir répondre aux messages déclarés pour T.

La rupture du contrat donne lieu à une erreur à la compilation ou à l'exécution.

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Type

Un type détermine un ensemble de messages applicables.

Lorsque

- ▶ une variable
- ▶ un retour de méthode
- ▶ un attribut

est déclaré d'un type T, l'objet référencé doit pouvoir répondre aux messages déclarés pour T.

La rupture du contrat donne lieu à une erreur à la compilation ou à l'exécution.

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Type

Un type détermine un ensemble de messages applicables.

Lorsque

- ▶ une variable
- ▶ un retour de méthode
- ▶ un attribut

est déclaré d'un type T, l'objet référencé doit pouvoir répondre aux messages déclarés pour T.

La rupture du contrat donne lieu à une erreur à la compilation ou à l'exécution.

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Types en Java

- ▶ primitifs
- ▶ classes
- ▶ interfaces

Identifiés par un nom et caractérisés par un ensemble de signatures

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Exemples : typage et protection

Le typage

permet d'éviter les envois de messages que le receveur ne peut comprendre.

```
Dans le code de la classe Véhicule :  
void methode(Place p12, Vehicule v1) {  
    v1.accueillir(v1);  
    p12.accueillir(p2);  
    p12.taille=1;  
    p12.accueillir(v2);  
}
```

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Exemples : typage et protection

Le typage

permet d'éviter les envois de messages que le receveur ne peut comprendre.

```
Dans le code de la classe Véhicule :  
void methode(Place p12, Vehicule v1) {  
    v1.accueillir(v1);  
    p12.accueillir(p2);  
    p12.taille=1;  
    p12.accueillir(v2);  
}
```

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Exemples : typage et protection

Le typage

permet d'éviter les envois de messages que le receveur ne peut comprendre.

```
Dans le code de la classe Véhicule :  
void methode(Place p12, Vehicule v1) {  
    v1.accueillir(v1);  
    p12.accueillir(p2);  
    p12.taille=1;  
    p12.accueillir(v2);  
}
```

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Exemples : typage et protection

La protection

permet d'éviter les envois de messages que l'émetteur n'a pas le droit d'envoyer au receveur.

Dans le code de la classe Véhicule :

```
void methode(Place p12, Vehicule v1) {  
    v1.accueillir(v1);  
    p12.accueillir(p2);  
    p12.taille=1;  
    p12.accueillir(v2);  
}
```

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Exemples : typage et protection

La protection

permet d'éviter les envois de messages que l'émetteur n'a pas le droit d'envoyer au receveur.

Dans le code de la classe Véhicule :

```
void methode(Place p12, Vehicule v1) {  
    v1.accueillir(v1);  
    p12.accueillir(p2);  
    p12.taille=1;  
    p12.accueillir(v2);  
}
```

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Principales raisons de restreindre l'accès à des propriétés

- ▶ Respect de l'intégrité
- ▶ Respect des spécifications
- ▶ Respect de règles de génie logiciel

Principales raisons de restreindre l'accès à des propriétés

- ▶ Respect de l'intégrité
→ on ne peut changer la taille d'un Véhicule
- ▶ Respect des spécifications
- ▶ Respect de règles de génie logiciel

Principales raisons de restreindre l'accès à des propriétés

- ▶ Respect de l'intégrité
→ on ne peut changer la taille d'un Véhicule
- ▶ Respect des spécifications
→ parking est le seul à pouvoir "rendre disponible" une place
- ▶ Respect de règles de génie logiciel

Principales raisons de restreindre l'accès à des propriétés

- ▶ Respect de l'intégrité
→ on ne peut changer la taille d'un Véhicule
- ▶ Respect des spécifications
→ parking est le seul à pouvoir "rendre disponible" une place
- ▶ Respect de règles de génie logiciel
→ masquage de l'implémentation

Principales raisons de restreindre l'accès à des propriétés

- ▶ Respect de l'intégrité
→ on ne peut changer la taille d'un Véhicule
- ▶ Respect des spécifications
→ parking est le seul à pouvoir “rendre disponible” une place
- ▶ Respect de règles de génie logiciel
→ masquage de l'implémentation
→ réduction de couplage

Sous Type

Si T est un sous-type de U alors l'ensemble de messages accepté par le type T inclus les messages acceptés par U .

Substituabilité

Si T est un sous-type de U et t un objet de type T , t doit pouvoir être substitué à un objet de type U .

Avantages : Une code (méthode ou constructeur) utilisant un type U fonctionnera si on lui passe des sous types de U . Extensibilité.

Sous Type

Si T est un sous-type de U alors l'ensemble de messages accepté par le type T inclus les messages acceptés par U .

Substituabilité

Si T est un sous-type de U et t un objet de type T , t doit pouvoir être substitué à un objet de type U .

Avantages : Une code (méthode ou constructeur) utilisant un type U fonctionnera si on lui passe des sous types de U . Extensibilité.

Sous Type

Si T est un sous-type de U alors l'ensemble de messages accepté par le type T inclus les messages acceptés par U .

Substituabilité

Si T est un sous-type de U et t un objet de type T , t doit pouvoir être substitué à un objet de type U .

Avantages : Une code (méthode ou constructeur) utilisant un type U fonctionnera si on lui passe des sous types de U . Extensibilité.

Les clients classiques sont

- ▶ le déclarant
- ▶ les héritiers
- ▶ les utilisateurs
- ▶ les instanciants

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Les clients classiques sont

- ▶ le déclarant
- ▶ les héritiers
- ▶ les utilisateurs
- ▶ les instanciants

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Les clients classiques sont

- ▶ le déclarant
- ▶ les héritiers
- ▶ les utilisateurs
- ▶ les instanciants

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Les clients classiques sont

- ▶ le déclarant
- ▶ les héritiers
- ▶ les utilisateurs
- ▶ les instanciants

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Les clients classiques sont

- ▶ le déclarant
- ▶ les héritiers
- ▶ les utilisateurs
- ▶ les instanciants

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Objectif

- ▶ factoriser du code
- ▶ éviter les répétitions.

Permettre une modification incrémentale.

- ▶ Ajout d'attributs
- ▶ Ajout de méthodes

Les nouvelles méthodes peuvent utiliser les anciennes.

En java, une sous classe est un sous type. Le sous typage est automatiquement respecté grâce à l'héritage.

Objectif

- ▶ factoriser du code
- ▶ éviter les répétitions.

Permettre une modification incrémentale.

- ▶ Ajout d'attributs
- ▶ Ajout de méthodes

Les nouvelles méthodes peuvent utiliser les anciennes.

En java, une sous classe est un sous type. Le sous typage est automatiquement respecté grâce à l'héritage.

```
public class Etudiant extends Personne
```

- ▶ Etudiant **hérite de** Personne
- ▶ Etudiant **est une sous classe de** Personne
- ▶ Personne **est une superclasse de** Etudiant

Les objets de la classe Etudiant savent répondre aux messages auxquels les objets de la classe Personne savent répondre.

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Des messages aux méthodes (bis)

On envoie les messages aux objets au travers de références

- ▶ L'objet doit posséder une méthode correspondant
- ▶ Le code de la méthode est interprété dans le contexte de l'objet receveur

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Des messages aux méthodes (bis)

On envoie les messages aux objets au travers de références

- ▶ L'objet doit posséder une méthode correspondant
Lookup : **il cherche dans sa classe puis dans une super-classe**
- ▶ Le code de la méthode est interprété dans le contexte de l'objet receveur

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Des messages aux méthodes (bis)

On envoie les messages aux objets au travers de références

- ▶ L'objet doit posséder une méthode correspondant
Lookup : il cherche dans sa classe puis dans une super-classe
- ▶ Le code de la méthode est interprété dans le contexte de l'objet receveur
quelle que soit la classe qui l'a défini

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition

On peut redéfinir une méthode afin d'adapter le comportement.

Exemple dans la classe `Personne`

```
public String toString(){
    String s ;
    s= "nom " +this.nom ;
    s=s+" prenom "+this.prenom ;
    s=super.toString() ;
    s=s+" formation "+this.formation ;
    return s ;
}
```

On peut faire appel à l'ancienne méthode avec `super.anciennemethode.`
et à un super constructeur avec `super.`

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition

On peut redéfinir une méthode afin d'adapter le comportement.

Exemple dans la classe `Personne`

```
public String toString(){
    String s ;
    s= "nom " +this.nom ;
    s=s+" prenom "+this.prenom ;
    s=super.toString() ;
    s=s+" formation "+this.formation ;
    return s ;
}
```

On peut faire appel à l'ancienne méthode avec `super.anciennemethode.`
et à un super constructeur avec `super.`

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition

On peut redéfinir une méthode afin d'adapter le comportement.

Exemple dans la classe **Etudiant**

```
public String toString(){  
    String s ;  
    s= "nom " +this.nom ;  
    s=s+" prenom "+this.prenom ;  
    s=super.toString() ;  
    s=s+" formation "+this.formation ;  
    return s ;  
}
```

On peut faire appel à l'ancienne méthode avec `super.anciennemethode.`
et à un super constructeur avec `super.`

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition

On peut redéfinir une méthode afin d'adapter le comportement.

Exemple dans la classe **Etudiant**

```
public String toString(){
    String s ;
    s= "nom " +this.nom ;
    s=s+" prenom "+this.prenom ;
    s=super.toString() ;
    s=s+" formation "+this.formation ;
    return s ;
}
```

On peut faire appel à l'ancienne méthode avec `super.anciennemethode.`
et à un super constructeur avec `super.`

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition

On peut redéfinir une méthode afin d'adapter le comportement.

Exemple dans la classe **Etudiant**

```
public String toString(){
    String s ;
    s= "nom " +this.nom ;
    s=s+" prenom "+this.prenom ;
    s=super.toString() ;
    s=s+" formation "+this.formation ;
    return s ;
}
```

On peut faire appel à l'ancienne méthode avec `super.anciennemethode.`
et à un super constructeur avec `super.`

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition

On peut redéfinir une méthode afin d'adapter le comportement.

Exemple dans la classe **Etudiant**

```
public String toString(){
    String s ;
    s= "nom " +this.nom ;
    s=s+" prenom "+this.prenom ;
    s=super.toString() ;
    s=s+" formation "+this.formation ;
    return s ;
}
```

On peut faire appel à l'ancienne méthode avec `super.anciennemethode.`
et à un super constructeur avec `super.`

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition

On peut redéfinir une méthode afin d'adapter le comportement.

Exemple dans la classe **Etudiant**

```
public String toString(){  
    String s ;  
    s= "nom " +this.nom ;  
    s=s+" prenom "+this.prenom ;  
    s=super.toString() ;  
    s=s+" formation "+this.formation ;  
    return s ;  
}
```

On peut faire appel à l'ancienne méthode avec `super.anciennemethode.`

et à un super constructeur avec `super.`

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition

On peut redéfinir une méthode afin d'adapter le comportement.

Exemple dans la classe **Etudiant**

```
public String toString(){
    String s ;
    s= "nom " +this.nom ;
    s=s+" prenom "+this.prenom ;
    s=super.toString() ;
    s=s+" formation "+this.formation ;
    return s ;
}
```

On peut faire appel à l'ancienne méthode avec `super.anciennemethode.`
et à un super constructeur avec `super.`

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Type Dynamique

Un objet est typé.

- ▶ Statiquement,
en fonction du type de sa référence.
- ▶ Dynamiquement,
en fonction de la classe dont le constructeur a créé l'objet.
- ▶ Le typage statique permet au compilateur
de vérifier que les messages envoyés sont corrects et autorisés.
- ▶ Le typage dynamique permet la liaison dynamique :
l'appel de la méthode la plus spécifique à l'objet.

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Un objet est typé.

- ▶ Statiquement,
en fonction du type de sa référence.
- ▶ Dynamiquement,
en fonction de la classe dont le constructeur a créé l'objet.
- ▶ Le typage statique permet au compilateur
de vérifier que les messages envoyés sont corrects et autorisés.
- ▶ Le typage dynamique permet la liaison dynamique :
l'appel de la méthode la plus spécifique à l'objet.

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
paul = new Etudiant();  
ungars = new Etudiant();
```

```
System.out.println(paul.toString());  
System.out.println(ungars.toString());
```

```
paul.setNumEtud("AA001");  
ungars.setNumEtud("AA001");
```

type statique Etudiant
type statique Personne

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
paul = new Etudiant();  
ungars = new Etudiant();
```

```
System.out.println(paul.toString());  
System.out.println(ungars.toString());
```

```
paul.setNumEtud("AA001");  
ungars.setNumEtud("AA001");
```

type statique Etudiant
type statique Personne

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
paul = new Etudiant();  
ungars = new Etudiant();
```

```
System.out.println(paul.toString());  
System.out.println(ungars.toString());
```

```
paul.setNumEtud("AA001");  
ungars.setNumEtud("AA001");
```

type statique Etudiant
type statique Personne

type dynamique Etudiant
type dynamique Etudiant

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
paul = new Etudiant();  
ungars = new Etudiant();
```

```
System.out.println(paul.toString());  
System.out.println(ungars.toString());
```

```
paul.setNumEtud("AA001");  
ungars.setNumEtud("AA001");
```

type statique Etudiant
type statique Personne

type dynamique Etudiant
type dynamique Etudiant

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Liaison tardive

```
Etudiant paul;  
Personne ungars;  
  
paul = new Etudiant();  
ungars = new Etudiant();  
  
System.out.println(paul.toString());  
System.out.println(ungars.toString());  
  
paul.setNumEtud("AA001");  
ungars.setNumEtud("AA001");
```

type statique Etudiant
type statique Personne

type dynamique Etudiant
type dynamique Etudiant

nom prenom formation

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
paul = new Etudiant();  
ungars = new Etudiant();
```

```
System.out.println(paul.toString());  
System.out.println(ungars.toString());
```

```
paul.setNumEtud("AA001");  
ungars.setNumEtud("AA001");
```

type statique Etudiant
type statique Personne

type dynamique Etudiant
type dynamique Etudiant

nom prenom formation
nom prenom formation

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
paul = new Etudiant();  
ungars = new Etudiant();
```

```
System.out.println(paul.toString());  
System.out.println(ungars.toString());
```

```
paul.setNumEtud("AA001");  
ungars.setNumEtud("AA001");
```

type statique Etudiant
type statique Personne

type dynamique Etudiant
type dynamique Etudiant

nom prenom formation
nom prenom formation

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Liaison tardive

```
Etudiant paul;  
Personne ungars;
```

```
paul = new Etudiant();  
ungars = new Etudiant();
```

```
System.out.println(paul.toString());  
System.out.println(ungars.toString());
```

```
paul.setNumEtud("AA001");  
ungars.setNumEtud("AA001");
```

type statique Etudiant
type statique Personne

type dynamique Etudiant
type dynamique Etudiant

nom prenom formation
nom prenom formation

ok
Erreur de type

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Liaison tardive

```
Etudiant paul;  
Personne ungars;  
  
paul = new Etudiant();  
ungars = new Etudiant();  
  
System.out.println(paul.toString());  
System.out.println(ungars.toString());  
  
paul.setNumEtud("AA001");  
ungars.setNumEtud("AA001");  
((Etudiant)ungars).setNumEtud("AA001");
```

type statique Etudiant
type statique Personne

type dynamique Etudiant
type dynamique Etudiant

nom prenom formation
nom prenom formation

ok
Erreur de type

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Liaison tardive

```
Etudiant paul;  
Personne ungars;  
  
paul = new Etudiant();  
ungars = new Etudiant();  
  
System.out.println(paul.toString());  
System.out.println(ungars.toString());  
  
paul.setNumEtud("AA001");  
ungars.setNumEtud("AA001");  
((Etudiant)ungars).setNumEtud("AA001");
```

type statique Etudiant
type statique Personne

type dynamique Etudiant
type dynamique Etudiant

nom prenom formation
nom prenom formation

ok
Erreur de type
possible, test à l'exécution

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Le compilateur ne veut pas prendre de risque, mais il tolère que vous en preniez.

Coercition de type

Forcer le compilateur à considérer une expression comme étant d'un sous-type de son type statique.

Syntaxe java : `((TYPE) expression)`

Peut échouer à l'exécution !

Le compilateur ne veut pas prendre de risque, mais il tolère que vous en preniez.

Coercition de type

Forcer le compilateur à considérer une expression comme étant d'un sous-type de son type statique.

Syntaxe java : `((TYPE) expression)`

Peut échouer à l'exécution !

Redéfinition et typage

On peut modifier le code d'une méthode héritée.
Et sa signature ?

Si on veut respecter le typage, il faut respecter les
contraintes de substituabilité.

il faut se souvenir envers qui est pris l'engagement de
type.

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition et typage

On peut modifier le code d'une méthode héritée.
Et sa signature ?

Si on veut respecter le typage, il faut respecter les
contraintes de substituabilité.

il faut se souvenir envers qui est pris l'engagement de
type.

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition et typage

On peut modifier le code d'une méthode héritée.
Et sa signature ?

Si on veut respecter le typage, il faut respecter les
contraintes de substituabilité.

il faut se souvenir envers qui est pris l'engagement de
type.

Protection, Typage

Le typage

Protection

Le sous typage

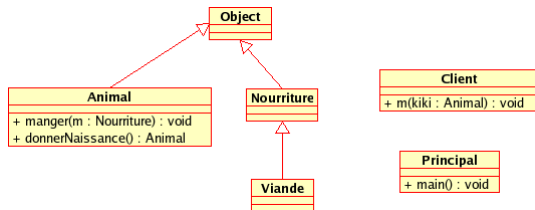
Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition : paramètres



```
public class Principal{
    public static void main(String[] args){
        Animal titi=new Animal();
        Client cli=new Client();
        cli.m(titi);}}

public class Client {
    public void m(Animal kiki){
        Nourriture n=new Nourriture();
        kiki.mange(n);
    }}
}
```

Protection, Typage

Le typage

Protection

Le sous typage

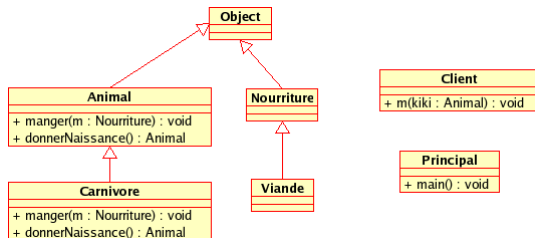
Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition : paramètres



```
public class Principal{
    public static void main(String[] args){
        Animal titi=new Animal();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Nourriture n=new Nourriture();
        kiki.mange(n);
    }
}
```

Protection, Typage

Le typage

Protection

Le sous typage

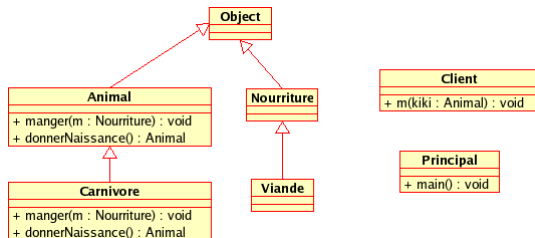
Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition : paramètres



```
public class Principal{
    public static void main(String[] args){
        Animal titi=new Carnivore();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Nourriture n=new Nourriture();
        kiki.mange(n);
    }
}
```

Protection, Typage

Le typage

Protection

Le sous typage

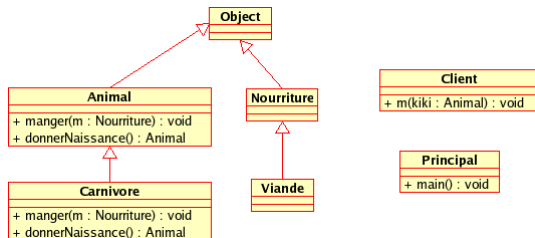
Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition : paramètres



```
public class Principal{
    public static void main(String[] args){
        Carnivore titi=new Carnivore();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Nourriture n=new Nourriture();
        kiki.mange(n);
    }
}
```

Protection, Typage

Le typage

Protection

Le sous typage

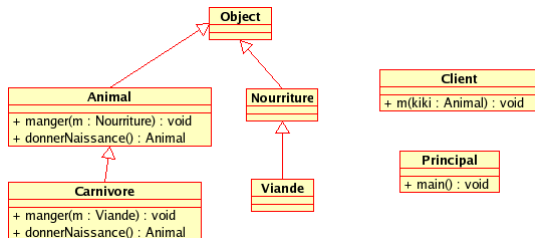
Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition : paramètres



```
public class Principal{
    public static void main(String[] args){
        Carnivore titi=new Carnivore();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Nourriture n=new Nourriture();
        kiki.mange(n);
    }
}
```

Protection, Typage

Le typage

Protection

Le sous typage

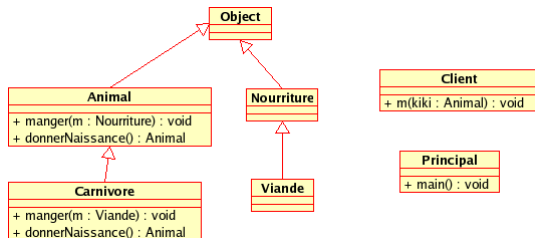
Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition : paramètres



```
public class Principal{
    public static void main(String[] args){
        Carnivore titi=new Carnivore();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Nourriture n=new Nourriture();
        kiki.mange(n); \\Erreur!
    }
}
```

Protection, Typage

Le typage

Protection

Le sous typage

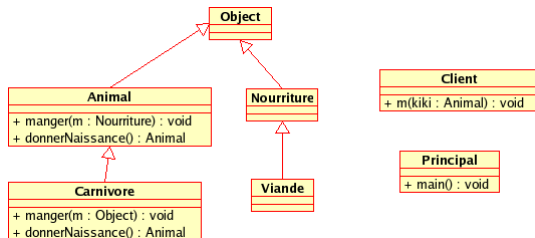
Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition : paramètres



```
public class Principal{
    public static void main(String[] args){
        Carnivore titi=new Carnivore();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Nourriture n=new Nourriture();
        kiki.mange(n); \\Ok, mais peu utile
    }
}
```

Protection, Typage

Le typage

Protection

Le sous typage

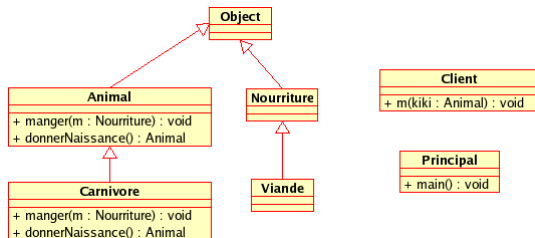
Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition : type de retour



```
public class Principal{
    public static void main(String[] args){
        Animal titi=new Animal();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Animal a;
        a=kiki.donnerNaissance();
    }
}
```

Protection, Typage

Le typage

Protection

Le sous typage

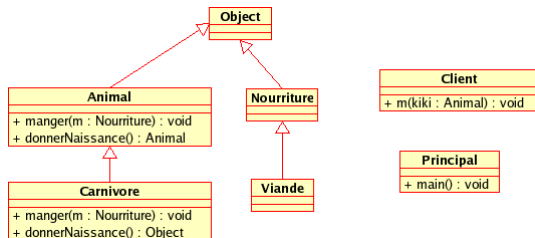
Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition : type de retour



```
public class Principal{
    public static void main(String[] args){
        Animal titi=new Animal();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Animal a;
        a=kiki.donnerNaissance();
    }
}
```

Protection, Typage

Le typage

Protection

Le sous typage

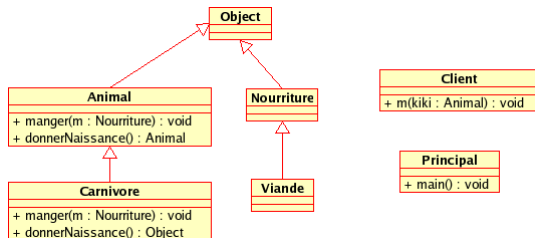
Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition : type de retour



```
public class Principal{
    public static void main(String[] args){
        Animal titi=new Carnivore();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Animal a;
        a=kiki.donnerNaissance();\\Erreur !
    }}
```

Protection, Typage

Le typage

Protection

Le sous typage

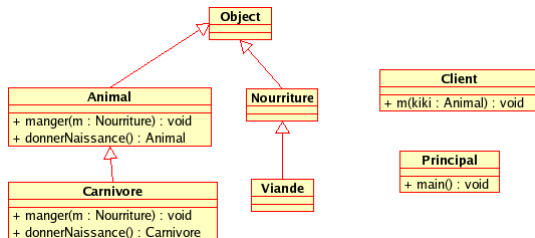
Héritage

Liaison tardive

Redéfinition et typage

généricité

Redéfinition : type de retour



```
public class Principal{
    public static void main(String[] args){
        Carnivore titi=new Carnivore();
        Client cli=new Client();
        cli.m(titi);}}
```

```
public class Client {
    public void m(Animal kiki){
        Animal a;
        a=kiki.donnerNaissance(); \\Ok
    }
}
```

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Les règles

Contraintes

Ce qu'on veut faire

Le typage

Java

Paramètres

Covariance

Contravariance

Invariance

Retour

Covariance

Covariance

Invariance

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Conclusion

- ▶ Typage statique -> Contrôle
- ▶ Sous-typage -> Extension avec de nouveaux objets
- ▶ Héritage -> Réutilisation des attributs et méthodes
- ▶ Redéfinition et liaison dynamique -> adaptation

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Deux sortes de généricité :

- ▶ polymorphique
 - ▶ Paramètre de type Object ex : Collections pré 1.5
- ▶ paramétrique
 - ▶ Paramètres d'un type à fixer à l'utilisation ex : Collections 1.5

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Deux sortes de genericité :

- ▶ polymorphique
 - ▶ Paramètre de type `Object` ex : Collections pré 1.5
- ▶ paramétrique
 - ▶ Paramètres d'un type à fixer à l'utilisation ex : Collections 1.5

Deux sortes de généricité :

- ▶ polymorphique
 - ▶ Paramètre de type `Object` ex : Collections pré 1.5
- ▶ paramétrique
 - ▶ Paramètres d'un type à fixer à l'utilisation ex : Collections 1.5

Deux sortes de généricité :

- ▶ polymorphique
 - ▶ Paramètre de type `Object` ex : Collections pré 1.5
- ▶ paramétrique
 - ▶ Paramètres d'un type à fixer à l'utilisation ex : Collections 1.5

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Généricité paramétrique

Paramétriser des types par des types.

Exemple : `Personne [] lesgens ;`

La variable `lesgens` est du type "tableau de `Personne`".

=> Le type tableau (`[]`) est un type paramétrique.

Dans l'exemple le paramètre est le type `Personne`.

- ▶ possède des méthodes `[]` d'accès à des éléments du type `T`
- ▶ `length`

Objectif : pouvoir créer et utiliser des types fonctionnant comme "tableau"

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Généricité paramétrique

Usage : `NomTypePrincipal<NomTypeParametre>`

Exemple : `Vector<Personne>`

Au lieu d'avoir une Liste d'Objets, on peut avoir une Liste de Personnes.

Avantages :

- ▶ On ne peut placer que des Personne (et ses sous-types) dans la liste
- ▶ On n'a pas besoin de cast quand on fait un `get()`, le compilateur "sait" que ce qui est retourné est une Personne

Un type peut être paramétré par plusieurs autres.

Exemple : `HashMap<Key, Value>`.

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Généricité paramétrique

Usage : `NomTypePrincipal<NomTypeParametre>`

Exemple : `Vector<Personne>`

Au lieu d'avoir une Liste d'Objets, on peut avoir une Liste de Personnes.

Avantages :

- ▶ On ne peut placer que des Personne (et ses sous-types) dans la liste
- ▶ On n'a pas besoin de cast quand on fait un `get()`, le compilateur "sait" que ce qui est retourné est une Personne

Un type peut être paramétré par plusieurs autres.

Exemple : `HashMap<Key, Value>`.

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Généricité paramétrique

Usage : `NomTypePrincipal<NomTypeParametre>`

Exemple : `Vector<Personne>`

Au lieu d'avoir une Liste d'Objets, on peut avoir une Liste de Personnes.

Avantages :

- ▶ On ne peut placer que des Personne (et ses sous-types) dans la liste
- ▶ On n'a pas besoin de cast quand on fait un `get()`, le compilateur "sait" que ce qui est retourné est une Personne

Un type peut être paramétré par plusieurs autres.

Exemple : `HashMap<Key, Value>`.

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Généricité paramétrique

Usage : `NomTypePrincipal<NomTypeParametre>`

Exemple : `Vector<Personne>`

Au lieu d'avoir une Liste d'Objets, on peut avoir une Liste de Personnes.

Avantages :

- ▶ On ne peut placer que des Personne (et ses sous-types) dans la liste
- ▶ On n'a pas besoin de cast quand on fait un `get()`, le compilateur "sait" que ce qui est retourné est une Personne

Un type peut être paramétré par plusieurs autres.

Exemple : `HashMap<Key,Value>`.

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Collection génériques

```
Vector<Personne> lesgens;  
lesgens= new Vector<Personne>();  
  
Personne pierre=new Personne("Pierre",18);  
Etudiant laure=new Etudiant("Laure",22,"LD654321");  
String paul="Jean";  
  
lesgens.add(pierre); //ok  
lesgens.add(laure); //ok, un Etudiant est une Personne  
lesgens.add(paul); // Non, String n'est pas une Personne  
  
lesgens.get(0).vieillirDe(5); //fait vieillir Pierre  
lesgens.get(1).vieillirDe(1); //fait vieillir Laure  
lesgens.get(1).affecteNumEtud("DL12345");  
    //non, affecteNumEtud n'est pas une methode de Personne  
( (Etudiant)lesgens.get(1)).affecteNumEtud("DL12345"); //ok
```

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Génériques :création

```
public class StockBizarre<T> {
    T unAttributDeTypeT;
    T unAutreAttributDeTypeT;

    public StockBizarre<T>(T unparamdetypeT){
        this.unAttributDeTypeT=unparamdetypeT;
    }
    public void add(T t) {
        this.unAutreAttributDeTypeT=this.unAttributDeTypeT;
        this.unAttributDeTypeT=t;
    }
    public T get(){
        T varlocale=this.unAutreAttributDeTypeT;
        this.unAutreAttributDeTypeT=this.unAttributDeTypeT;
        this.unAttributDeTypeT=varlocale;
        return varlocale;
    }
}
```

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Généricité contrainte

La généricité contrainte permet de limiter le type du paramètre à un type X de façon à pouvoir lui appliquer des méthodes de X

```
public class StockBizarre<T extends Personne> {  
    T unAttributDeTypeT;  
    T unAutreAttributDeTypeT;  
  
    public StockBizarre<T>(T unparamdetypeT){  
        this.unAttributDeTypeT=unparamdetypeT;  
    }  
  
    public String toString()  
    {return this.unAttributDeTypeT.getNom();}
```

Exemple : limiter le type à comparable permet de définir des listes triées génériques

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Iterator version générique

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();    // Optional  
}
```

Utilisation :

```
static void filter(Collection<String> collect) {  
    for (Iterator<String> i = collect.iterator(); i.hasNext(); )  
        if (!i.next().startsWith("#"))    i.remove();  
}
```

Pas besoin de cast pour utiliser la méthode de String !

Protection, Typage

- Le typage
- Protection
- Le sous typage
- Héritage
- Liaison tardive
- Redéfinition et typage
- généricité

Iterator version générique

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();    // Optional  
}
```

Utilisation :

```
static void filter(Collection<String> collect) {  
    for (Iterator<String> i = collect.iterator(); i.hasNext(); )  
        if (i.next().startsWith("#"))    i.remove();  
}
```

Pas besoin de cast pour utiliser la méthode de String !

Protection, Typage

- Le typage
- Protection
- Le sous typage
- Héritage
- Liaison tardive
- Redéfinition et typage
- généricité

Iterator version générique

```
public interface Iterator<E> {  
    boolean hasNext();  
    E next();  
    void remove();    // Optional  
}
```

Utilisation :

```
static void filter(Collection<String> collect) {  
    for (Iterator<String> i = collect.iterator(); i.hasNext(); )  
        if (i.next().startsWith("#"))    i.remove();  
}
```

Pas besoin de cast pour utiliser la méthode de String !

Protection, Typage

- Le typage
- Protection
- Le sous typage
- Héritage
- Liaison tardive
- Redéfinition et typage
- généricité

Lorsque l'on fait plusieurs usages de l'élément d'une collection, sans altérer la collection elle même :

```
static void filter(Collection<String> collect) {  
    for (Iterator<String> i = collect.iterator(); i.hasNext(); ){  
        String temp=i.next();  
        if (!temp.startsWith("#")) System.out.println(temp);  
    }  
}
```

On peut se contenter de la boucle simplifiée :

```
static void filter(Collection<String> collect) {  
    for (String temp : collect )  
        if (!temp.startsWith("#")) System.out.println(temp);  
}
```

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Lorsque l'on fait plusieurs usages de l'élément d'une collection, sans altérer la collection elle même :

```
static void filter(Collection<String> collect) {  
    for (Iterator<String> i = collect.iterator(); i.hasNext(); ){  
        String temp=i.next();  
        if (!temp.startsWith("#")) System.out.println(temp);  
    }  
}
```

On peut se contenter de la boucle simplifiée :

```
static void filter(Collection<String> collect) {  
    for (String temp : collect )  
        if (!temp.startsWith("#")) System.out.println(temp);  
}
```

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Généricité et Sous-Typage

Liste<Etudiant> est il un sous type de
Collection<Etudiant> ?

Oui : Elle possède au moins toutes les méthodes de
Collection<Etudiant> par héritage

Liste<Etudiant> est il un sous type de Liste<Personne> ?

Indice : La redéfinition covariante de méthode est elle
compatible avec le sous-typage ?

Non

Pour faciliter les boucles génériques, T<?> est un
super-type de tout les T<E>

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Généricité et Sous-Typage

Liste<Etudiant> est il un sous type de
Collection<Etudiant> ?

Oui : Elle possède au moins toutes les méthodes de
Collection<Etudiant> par héritage

Liste<Etudiant> est il un sous type de Liste<Personne> ?

Indice : La redéfinition covariante de méthode est elle
compatible avec le sous-typage ?

Non

Pour faciliter les boucles génériques, T<?> est un
super-type de tout les T<E>

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Généricité et Sous-Typage

Liste<Etudiant> est il un sous type de
Collection<Etudiant> ?

Oui : Elle possède au moins toutes les méthodes de
Collection<Etudiant> par héritage

Liste<Etudiant> est il un sous type de Liste<Personne> ?

Indice : La redéfinition covariante de méthode est elle
compatible avec le sous-typage ?

Non

Pour faciliter les boucles génériques, T<?> est un
super-type de tout les T<E>

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Généricité et Sous-Typage

Liste<Etudiant> est il un sous type de
Collection<Etudiant> ?

Oui : Elle possède au moins toutes les méthodes de
Collection<Etudiant> par héritage

Liste<Etudiant> est il un sous type de Liste<Personne> ?

Indice : La redéfinition covariante de méthode est elle
compatible avec le sous-typage ?

Non

Pour faciliter les boucles génériques, T<?> est un
super-type de tout les T<E>

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Généricité et Sous-Typage

Liste<Etudiant> est il un sous type de
Collection<Etudiant> ?

Oui : Elle possède au moins toutes les méthodes de
Collection<Etudiant> par héritage

Liste<Etudiant> est il un sous type de Liste<Personne> ?

Indice : La redéfinition covariante de méthode est elle
compatible avec le sous-typage ?

Non

Pour faciliter les boucles génériques, T< ?> est un
super-type de tout les T<E>

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Généricité et Sous-Typage

Liste<Etudiant> est il un sous type de
Collection<Etudiant> ?

Oui : Elle possède au moins toutes les méthodes de
Collection<Etudiant> par héritage

Liste<Etudiant> est il un sous type de Liste<Personne> ?

Indice : La redéfinition covariante de méthode est elle
compatible avec le sous-typage ?

Non

Pour faciliter les boucles génériques, T< ?> est un
super-type de tout les T<E>

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Méthodes génériques

On peut vouloir paramétrer une méthode par un type si elle doit fonctionner pour plusieurs types et

- ▶ sa classe n'a pas de raison d'être générique
- ▶ ou le type générique de sa classe n'est pas le seul type pour lequel la méthode est utile

```
public <T> void insertionGenerique(T[] tab; Collection<T> coll)
{ for (T elemtab: tab) coll.add(elemtab); }
```

On l'utilise comme une méthode normale, (sans la faire précéder de <String> ou <Integer>), le compilateur infère T du type des paramètres.

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Méthodes génériques

On peut vouloir paramétrer une méthode par un type si elle doit fonctionner pour plusieurs types et

- ▶ sa classe n'a pas de raison d'être générique
- ▶ ou le type générique de sa classe n'est pas le seul type pour lequel la méthode est utile

```
public <T> void insertionGenerique(T[] tab; Collection<T> coll)
{ for (T elemtab: tab) coll.add(elemtab); }
```

On l'utilise comme une méthode normale, (sans la faire précéder de <String> ou <Integer>), le compilateur infère T du type des paramètres.

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Méthodes génériques

On peut vouloir paramétrer une méthode par un type si elle doit fonctionner pour plusieurs types et

- ▶ sa classe n'a pas de raison d'être générique
- ▶ ou le type générique de sa classe n'est pas le seul type pour lequel la méthode est utile

```
public <T> void insertionGenerique(T[] tab; Collection<T> coll)
{ for (T elemtab: tab) coll.add(elemtab); }
```

On l'utilise comme une méthode normale, (sans la faire précéder de <String> ou <Integer>), le compilateur infère T du type des paramètres.

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Méthodes génériques

On peut vouloir paramétrer une méthode par un type si elle doit fonctionner pour plusieurs types et

- ▶ sa classe n'a pas de raison d'être générique
- ▶ ou le type générique de sa classe n'est pas le seul type pour lequel la méthode est utile

```
public <T> void insertionGenerique(T[] tab; Collection<T> coll)
{ for (T elemtab: tab) coll.add(elemtab); }
```

On l'utilise comme une méthode normale, (sans la faire précéder de <String> ou <Integer>), le compilateur infère T du type des paramètres.

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Méthodes génériques

On peut vouloir paramétrer une méthode par un type si elle doit fonctionner pour plusieurs types et

- ▶ sa classe n'a pas de raison d'être générique
- ▶ ou le type générique de sa classe n'est pas le seul type pour lequel la méthode est utile

```
public <T> void insertionGenerique(T[] tab; Collection<T> coll)
{ for (T elemtab: tab) coll.add(elemtab); }
```

On l'utilise comme une méthode normale, (sans la faire précéder de <String> ou <Integer>), le compilateur infère T du type des paramètres.

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

On essaie ?

Modélisation **simple** :

Un Chef, de la Nourriture et une Cuisine.

Objectif : générer un Menu.

On ne touche pas aux machines avant d'avoir fait son diagramme de classes

Pour lancer eclipse :

```
eclipse -data chemindemonworkspace
```

Puis créer nouveau projet java, nouveau package et vos classes

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Protection, Typage

Le typage

Protection

Le sous typage

Héritage

Liaison tardive

Redéfinition et typage

généricité

Utilisation de l'héritage

Refactoring

Héritage multiple

Interfaces

Coder vite sans se soucier de demain c'est prendre un risque

Métaphore du prêt :

- ▶ On veut le résultat *tout de suite*
- ▶ On paye les intérêts en maintenance
- ▶ En cas de surendettement, on fait faillite

⇒ On renégocie son prêt, on fait des remboursements anticipés quand on le peut (*après une release*).

Coder vite sans se soucier de demain c'est prendre un risque

Métaphore du prêt :

- ▶ On veut le résultat *tout de suite*
- ▶ On paye les intérêts en maintenance
- ▶ En cas de surendettement, on fait faillite

⇒ On renégocie son prêt, on fait des remboursements anticipés quand on le peut (*après une release*).

Coder vite sans se soucier de demain c'est prendre un risque

Métaphore du prêt :

- ▶ On veut le résultat *tout de suite*
- ▶ On paye les intérêts en maintenance
- ▶ En cas de surendettement, on fait faillite

⇒ On renégocie son prêt, on fait des remboursements anticipés quand on le peut (*après une release*).

Coder vite sans se soucier de demain c'est prendre un risque

Métaphore du prêt :

- ▶ On veut le résultat *tout de suite*
- ▶ On paye les intérêts en maintenance
- ▶ En cas de surendettement, on fait faillite

⇒ On renégocie son prêt, on fait des remboursements anticipés quand on le peut (*après une release*).

Coder vite sans se soucier de demain c'est prendre un risque

Métaphore du prêt :

- ▶ On veut le résultat *tout de suite*
- ▶ On paye les intérêts en maintenance
- ▶ En cas de surendettement, on fait faillite

⇒ On renégocie son prêt, on fait des remboursements anticipés quand on le peut (*après une release*).

Revoir régulièrement sa conception pour identifier les "besoins en nouveaux types"

- ▶ lorsque l'on veut qu'une méthode renvoie plusieurs valeurs
- ▶ lorsqu'on utilise des paramètres "en groupe"
- ▶ lorsqu'une méthode devrait être appliquée à plusieurs types d'objets : un supertype commun
- ▶ lorsqu'on peut factoriser du code

Revoir régulièrement sa conception pour identifier les "besoins en nouveaux types"

- ▶ lorsque l'on veut qu'une méthode renvoie plusieurs valeurs
- ▶ lorsqu'on utilise des paramètres "en groupe"
- ▶ lorsqu'une méthode devrait être appliquée à plusieurs types d'objets : un supertype commun
- ▶ lorsqu'on peut factoriser du code

Revoir régulièrement sa conception pour identifier les "besoins en nouveaux types"

- ▶ lorsque l'on veut qu'une méthode renvoie plusieurs valeurs
- ▶ lorsqu'on utilise des paramètres "en groupe"
- ▶ lorsqu'une méthode devrait être appliquée à plusieurs types d'objets : un supertype commun
- ▶ lorsqu'on peut factoriser du code

Revoir régulièrement sa conception pour identifier les "besoins en nouveaux types"

- ▶ lorsque l'on veut qu'une méthode renvoie plusieurs valeurs
- ▶ lorsqu'on utilise des paramètres "en groupe"
- ▶ lorsqu'une méthode devrait être appliquée à plusieurs types d'objets : un supertype commun
- ▶ lorsqu'on peut factoriser du code

Revoir régulièrement sa conception pour identifier les "besoins en nouveaux types"

- ▶ lorsque l'on veut qu'une méthode renvoie plusieurs valeurs
- ▶ lorsqu'on utilise des paramètres "en groupe"
- ▶ lorsqu'une méthode devrait être appliquée à plusieurs types d'objets : un supertype commun
- ▶ lorsqu'on peut factoriser du code

Refactoring : méthodes

Favoriser les méthodes courtes et nombreuses pour offrir des points d'extension.

Extraire la partie générique des opérations

- ▶ trop longues
- ▶ intérieur des boucles
- ▶ créations d'objets

Procédure incrémentale : *cent fois sur le métier...*

Si toutes les méthodes ne sont pas implantées, la classe est abstraite.

Refactoring : méthodes

Favoriser les méthodes courtes et nombreuses pour offrir des points d'extension.

Extraire la partie générique des opérations

- ▶ trop longues
- ▶ intérieur des boucles
- ▶ créations d'objets

Procédure incrémentale : *cent fois sur le métier...*

Si toutes les méthodes ne sont pas implantées, la classe est abstraite.

Favoriser les méthodes courtes et nombreuses pour offrir des points d'extension.

Extraire la partie générique des opérations

- ▶ trop longues
- ▶ intérieur des boucles
- ▶ créations d'objets

Procédure incrémentale : *cent fois sur le métier...*

Si toutes les méthodes ne sont pas implantées, la classe est abstraite.

Favoriser les méthodes courtes et nombreuses pour offrir des points d'extension.

Extraire la partie générique des opérations

- ▶ trop longues
- ▶ intérieur des boucles
- ▶ créations d'objets

Procédure incrémentale : *cent fois sur le métier...*

Si toutes les méthodes ne sont pas implantées, la classe est abstraite.

Favoriser les méthodes courtes et nombreuses pour offrir des points d'extension.

Extraire la partie générique des opérations

- ▶ trop longues
- ▶ intérieur des boucles
- ▶ créations d'objets

Procédure incrémentale : *cent fois sur le métier...*

Si toutes les méthodes ne sont pas implantées, la classe est abstraite.

Protection et héritage

Lorsque l'on crée de nouvelles méthodes pour favoriser l'extension par redéfinition, elles ne doivent être

- ▶ ni publiques
- ▶ ni privées

⇒ un nouveau niveau de protection

Protection et héritage

Lorsque l'on crée de nouvelles méthodes pour favoriser l'extension par redéfinition, elles ne doivent être

- ▶ ni publiques
- ▶ ni privées

⇒ un nouveau niveau de protection

Utilisation de
l'héritage

Refactoring

Héritage multiple

Interfaces

Protection et héritage

Lorsque l'on crée de nouvelles méthodes pour favoriser l'extension par redéfinition, elles ne doivent être

- ▶ ni publiques
- ▶ ni privées

⇒ un nouveau niveau de protection

Protection et héritage

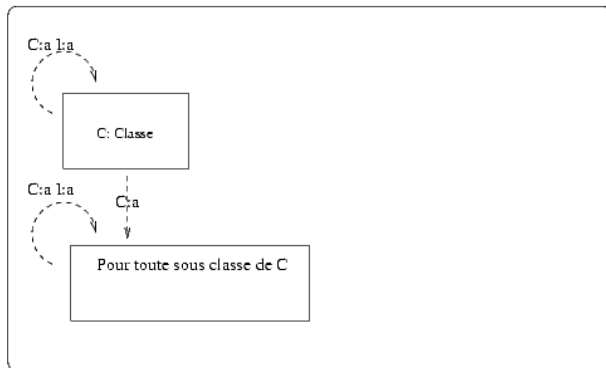
Lorsque l'on crée de nouvelles méthodes pour favoriser l'extension par redéfinition, elles ne doivent être

- ▶ ni publiques
- ▶ ni privées

⇒ un nouveau niveau de protection

protected

C définit la propriété "a" protected



Utilisation de
l'héritage

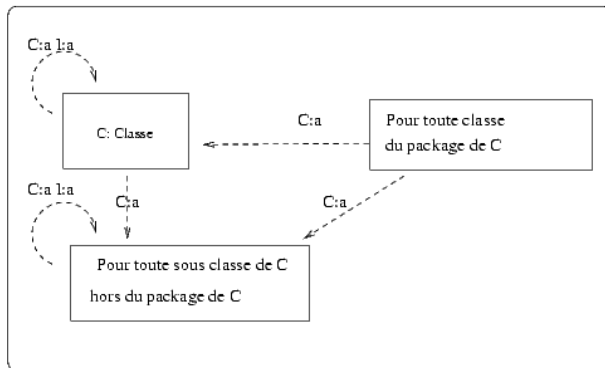
Refactoring

Héritage multiple

Interfaces

protected

C définit la propriété "a" protected



Utilisation de
l'héritage

Refactoring

Héritage multiple

Interfaces

Classe abstraite

Une classe abstraite ne peut avoir d'instances. Car

- ▶ trop générale (animal, nombre, comparable)
- ▶ ou possédant des méthodes non exprimables
abstraites

Toutes ses méthodes ne sont pas forcément abstraites.

Ex : La méthode `supérieur` de la classe `Comparable` peut se définir à partir d'`inférieur` et `égal`, qui ne peuvent être définies que dans les sous-classes de `comparable`

Héritage multiple

Avantage : plus de code hérité (en faisant attention à la spécialisation) Inconvénients : conflits

- ▶ conflit de nom (département géographique ou administratif)
 - ⇒ renommage en amont
 - ▶ suppose l'accès au code
 - ▶ impact sur les clients
- ▶ conflit de valeur (comment choisir la bonne)
 - ⇒ désambiguïsation
 - ▶ désignation explicite (C++) ("casse" la liaison tardive et le principe de découplage)
 - ▶ renommage local et sélection du nom (Eiffel)

La solution de Java : interdiction de l'héritage multiple

Héritage multiple

Avantage : plus de code hérité (en faisant attention à la spécialisation) Inconvénients : conflits

- ▶ conflit de nom (département géographique ou administratif)
 - ⇒ renommage en amont
 - ▶ suppose l'accès au code
 - ▶ impact sur les clients
- ▶ conflit de valeur (comment choisir la bonne)
 - ⇒ désambiguisation
 - ▶ désignation explicite (C++) ("casse" la liaison tardive et le principe de découplage)
 - ▶ renommage local et sélection du nom (Eiffel)

La solution de Java : interdiction de l'héritage multiple

Héritage multiple

Avantage : plus de code hérité (en faisant attention à la spécialisation) Inconvénients : conflits

- ▶ conflit de nom (département géographique ou administratif)
 - ⇒ renommage en amont
 - ▶ suppose l'accès au code
 - ▶ impact sur les clients
- ▶ conflit de valeur (comment choisir la bonne)
 - ⇒ désambiguïsation
 - ▶ désignation explicite (C++) ("casse" la liaison tardive et le principe de découplage)
 - ▶ renommage local et sélection du nom (Eiffel)

La solution de Java : interdiction de l'héritage multiple

Héritage multiple

Avantage : plus de code hérité (en faisant attention à la spécialisation) Inconvénients : conflits

- ▶ conflit de nom (département géographique ou administratif)
 - ⇒ renommage en amont
 - ▶ suppose l'accès au code
 - ▶ impact sur les clients
- ▶ conflit de valeur (comment choisir la bonne)
 - ⇒ désambiguisation
 - ▶ désignation explicite (C++) ("casse" la liaison tardive et le principe de découplage)
 - ▶ renommage local et sélection du nom (Eiffel)

La solution de Java : interdiction de l'héritage multiple

Avantage : plus de code hérité (en faisant attention à la spécialisation) Inconvénients : conflits

- ▶ conflit de nom (département géographique ou administratif)
 - ⇒ renommage en amont
 - ▶ suppose l'accès au code
 - ▶ impact sur les clients
- ▶ conflit de valeur (comment choisir la bonne)
 - ⇒ désambiguisation
 - ▶ désignation explicite (C++) ("casse" la liaison tardive et le principe de découplage)
 - ▶ renommage local et sélection du nom (Eiffel)

La solution de Java : interdiction de l'héritage multiple

Interfaces

Les conflits de valeur disparaissent s'il n'y a pas de code. On peut renoncer à l'héritage mais conserver le typage multiple avec une nouvelle catégorie de types : les interfaces.

- ▶ pas d'attributs
- ▶ pas de constructeurs
- ▶ des méthodes publiques

Interfaces

Les conflits de valeur disparaissent s'il n'y a pas de code. On peut renoncer à l'héritage mais conserver le typage multiple avec une nouvelle catégorie de types : les interfaces.

- ▶ pas d'attributs
- ▶ pas de constructeurs
- ▶ des méthodes publiques

Interfaces

Les conflits de valeur disparaissent s'il n'y a pas de code. On peut renoncer à l'héritage mais conserver le typage multiple avec une nouvelle catégorie de types : les interfaces.

- ▶ pas d'attributs
- ▶ pas de constructeurs
- ▶ des méthodes publiques

Interfaces

Les conflits de valeur disparaissent s'il n'y a pas de code. On peut renoncer à l'héritage mais conserver le typage multiple avec une nouvelle catégorie de types : les interfaces.

- ▶ pas d'attributs
- ▶ pas de constructeurs
- ▶ des méthodes publiques

Une classe peut hériter d'une seule autre, mais peut *implémenter* plusieurs interfaces.

Ex: `public class Etudiant extends Personne implements localise`

La classe s'engage à fournir un code pour chaque méthode de l'interface : Elle doit respecter le type !

- ▶ soit en définissant la méthode
- ▶ soit en en héritant

⇒ Toute classe possède les types de ses superclasses.

Interfaces

Une classe peut hériter d'une seule autre, mais peut *implémenter* plusieurs interfaces.

Ex: `public class Etudiant extends Personne implements localise`

La classe s'engage à fournir un code pour chaque méthode de l'interface : Elle doit respecter le type !

- ▶ soit en définissant la méthode
- ▶ soit en héritant

⇒ Toute classe possède les types de ses superclasses.

Interfaces

Une classe peut hériter d'une seule autre, mais peut *implémenter* plusieurs interfaces.

Ex: `public class Etudiant extends Personne implements localise`

La classe s'engage à fournir un code pour chaque méthode de l'interface : Elle doit respecter le type !

- ▶ soit en définissant la méthode
- ▶ soit en héritant

⇒ Toute classe possède les types de ses superclasses.

Interfaces

Une classe peut hériter d'une seule autre, mais peut *implémenter* plusieurs interfaces.

Ex: `public class Etudiant extends Personne implements localise`

La classe s'engage à fournir un code pour chaque méthode de l'interface : Elle doit respecter le type !

- ▶ soit en définissant la méthode
- ▶ soit en en héritant

⇒ Toute classe possède les types de ses superclasses.

Interfaces

Une classe peut hériter d'une seule autre, mais peut *implémenter* plusieurs interfaces.

Ex: `public class Etudiant extends Personne implements localise`

La classe s'engage à fournir un code pour chaque méthode de l'interface : Elle doit respecter le type !

- ▶ soit en définissant la méthode
- ▶ soit en en héritant

⇒ Toute classe possède les types de ses superclasses.

Extensibilité

Programmer vers les interfaces, pas vers les classes.

Utiliser un type "abstrait" plus général, facilite l'extension par des objets qui en sont des sous-types.

Si le type d'un paramètre est celui d'une classe, alors seuls des objets issus de sous classes seront acceptés.

Les contraintes de l'héritage par rapport à l'implémentation d'interfaces sont :

- ▶ récupération de code pas forcément utile
- ▶ si héritage simple, la meilleure superclasse n'est pas forcément celle utilisée

A contrario, l'implémentation ne coûte presque rien. Il est facile de créer des objets qui sont du type d'une interface

Extensibilité

Programmer vers les interfaces, pas vers les classes.

Utiliser un type "abstrait" plus général, facilite l'extension par des objets qui en sont des sous-types.

Si le type d'un paramètre est celui d'une classe, alors seuls des objets issus de sous classes seront acceptés.

Les contraintes de l'héritage par rapport à l'implémentation d'interfaces sont :

- ▶ récupération de code pas forcément utile
- ▶ si héritage simple, la meilleure superclasse n'est pas forcément celle utilisée

A contrario, l'implémentation ne coûte presque rien. Il est facile de créer des objets qui sont du type d'une interface

Extensibilité

Programmer vers les interfaces, pas vers les classes.

Utiliser un type "abstrait" plus général, facilite l'extension par des objets qui en sont des sous-types.

Si le type d'un paramètre est celui d'une classe, alors seuls des objets issus de sous classes seront acceptés.

Les contraintes de l'héritage par rapport à l'implémentation d'interfaces sont :

- ▶ récupération de code pas forcément utile
- ▶ si héritage simple, la meilleure superclasse n'est pas forcément celle utilisée

A contrario, l'implémentation ne coûte presque rien. Il est facile de créer des objets qui sont du type d'une interface

Extensibilité

Programmer vers les interfaces, pas vers les classes.

Utiliser un type "abstrait" plus général, facilite l'extension par des objets qui en sont des sous-types.

Si le type d'un paramètre est celui d'une classe, alors seuls des objets issus de sous classes seront acceptés.

Les contraintes de l'héritage par rapport à l'implémentation d'interfaces sont :

- ▶ récupération de code pas forcément utile
- ▶ si héritage simple, la meilleure superclasse n'est pas forcément celle utilisée

A contrario, l'implémentation ne coûte presque rien. Il est facile de créer des objets qui sont du type d'une interface

Extensibilité

Programmer vers les interfaces, pas vers les classes.

Utiliser un type "abstrait" plus général, facilite l'extension par des objets qui en sont des sous-types.

Si le type d'un paramètre est celui d'une classe, alors seuls des objets issus de sous classes seront acceptés.

Les contraintes de l'héritage par rapport à l'implémentation d'interfaces sont :

- ▶ récupération de code pas forcément utile
- ▶ si héritage simple, la meilleure superclasse n'est pas forcément celle utilisée

A contrario, l'implémentation ne coûte presque rien. Il est facile de créer des objets qui sont du type d'une interface

La décision d'utiliser des interfaces est soit prévisionnelle, soit un raffinement.

1. création d'interface

regroupant des méthodes fréquemment rencontrées et utilisées de concert

2. déclaration d'implémentation dans les classes qui possèdent les méthodes

3. utilisation de l'interface plutôt que les classes pour typer \Rightarrow clients plus flexibles

4. créer une classe abstraite implémentant l'interface \Rightarrow en prévision d'une factorisation

La décision d'utiliser des interfaces est soit prévisionnelle, soit un raffinement.

1. création d'interface

regroupant des méthodes fréquemment rencontrées et utilisées de concert

2. déclaration d'implémentation dans les classes qui possèdent les méthodes

3. utilisation de l'interface plutôt que les classes pour typer \Rightarrow clients plus flexibles

4. créer une classe abstraite implémentant l'interface \Rightarrow en prévision d'une factorisation

La décision d'utiliser des interfaces est soit prévisionnelle, soit un raffinement.

1. création d'interface

regroupant des méthodes fréquemment rencontrées et utilisées de concert

2. déclaration d'implémentation dans les classes qui possèdent les méthodes

3. utilisation de l'interface plutôt que les classes pour typer \Rightarrow clients plus flexibles

4. créer une classe abstraite implémentant l'interface \Rightarrow en prévision d'une factorisation

La décision d'utiliser des interfaces est soit prévisionnelle, soit un raffinement.

1. création d'interface

regroupant des méthodes fréquemment rencontrées et utilisées de concert

2. déclaration d'implémentation dans les classes qui possèdent les méthodes

3. utilisation de l'interface plutôt que les classes pour typer \Rightarrow clients plus flexibles

4. créer une classe abstraite implémentant l'interface \Rightarrow en prévision d'une factorisation

La décision d'utiliser des interfaces est soit prévisionnelle, soit un raffinement.

1. création d'interface

regroupant des méthodes fréquemment rencontrées et utilisées de concert

2. déclaration d'implémentation dans les classes qui possèdent les méthodes

3. utilisation de l'interface plutôt que les classes pour typer \Rightarrow clients plus flexibles

4. créer une classe abstraite implémentant l'interface \Rightarrow en prévision d'une factorisation

La décision d'utiliser des interfaces est soit prévisionnelle, soit un raffinement.

1. création d'interface

regroupant des méthodes fréquemment rencontrées et utilisées de concert

2. déclaration d'implémentation dans les classes qui possèdent les méthodes

3. utilisation de l'interface plutôt que les classes pour typer \Rightarrow clients plus flexibles

4. créer une classe abstraite implémentant l'interface \Rightarrow en prévision d'une factorisation

La décision d'utiliser des interfaces est soit prévisionnelle, soit un raffinement.

1. création d'interface

regroupant des méthodes fréquemment rencontrées et utilisées de concert

2. déclaration d'implémentation dans les classes qui possèdent les méthodes

3. utilisation de l'interface plutôt que les classes pour typer \Rightarrow clients plus flexibles

4. créer une classe abstraite implémentant l'interface \Rightarrow en prévision d'une factorisation

Exemple : iterators

L'interface `Iterator` permet de parcourir les éléments d'une collection sans se soucier du type de la collection

```
public interface Iterator {  
    boolean hasNext();  
    Object next();  
    void remove();    // Optional  
}
```

Utilisation :

```
static void filter(Collection collect) {  
    for (Iterator i = collect.iterator(); i.hasNext(); )  
        if (!cond(i.next())) i.remove();  
}
```

Utilisation de
l'héritage

Refactoring

Héritage multiple

Interfaces

Interfaces avec eclipse

Pour définir une interface :

- ▶ clic-droit sur le nom de la classe,
- ▶ Refactor->Extract Interface
- ▶ choisir les méthodes
- ▶ cocher Change references to the class into references to the Interface (where possible).

Pour permettre son utilisation

- ▶ clic-droit sur le nom d'une classe implémentant l'interface,
- ▶ Refactor->Use Supertype where possible
- ▶ sélectionner l'interface

Interface ou classe Abstraite

Les deux !

- ▶ Créer une ou des interfaces
pour que les clients soient plus flexibles.
- ▶ Créer une classe abstraite implémentant ces interfaces
afin de factoriser du code.

Utilisation de
l'héritage

Refactoring

Héritage multiple

Interfaces

La librairie de collections de Java offre :

- ▶ des structures de données
- ▶ des interfaces pour faciliter la manipulation
- ▶ des algorithmes

Exemple : collections

```
public interface Collection {  
    // Basic Operations  
    int size();  
    boolean isEmpty();  
    boolean contains(Object element);  
    boolean add(Object element);    // Optional  
    boolean remove(Object element); // Optional  
    Iterator iterator();  
  
    // Bulk Operations  
    boolean containsAll(Collection c);  
    boolean addAll(Collection c);    // Optional  
    boolean removeAll(Collection c); // Optional  
    boolean retainAll(Collection c); // Optional  
    void clear();                    // Optional  
  
    // Array Operations  
    Object[] toArray();  
    Object[] toArray(Object a[]);  
}
```

Interfaces et implémentations

Dans java.util on trouve

Interface	Impl.	nature et méthodes)
Set	HashSet	ensembles (union, unicité ...)
List	ArrayList, LinkedList	trié indicé get(int), set(int,valeur), subList(int,int) ListIterator
Map	HashMap TreeMap	Dictionnaires put(clé, valeur), get(clé)

Dans la classe Collections on trouve :

- ▶ `sort(List)`
- ▶ `sort(List,Comparator)`
- ▶ `shuffle(List)`
- ▶ `reverse(List)`
- ▶ `fill(List,Object)`
- ▶ `binarySearch(List,Object)`
- ▶ `singleton(Object)`

conclusion

Pour produire un code maintenable (correction et évolution)

- ▶ Décomposez, factorisez et rendez le code générique
- ▶ Programmez vers les interfaces
- ▶ Retravaillez votre code régulièrement

Utilisation de
l'héritage

Refactoring

Héritage multiple

Interfaces

Exception : Emission

Une exception signale une anomalie dans une exécution à l'aide d'un objet de la classe Exception.

Utilisation :

```
\\Declaration
public void loadFile(String fname) throws Exception{
\\ signifie : cette methode peut declencher une Exception

if ((fname==null)|| (fname.equals(""))){
\\Creation de l'objet
Exception e=new Exception("pas de fichier");
\\Emission de l'Exception
throw(e);
\\ Interruption de la methode
}
...
}
```

Exceptions

Emission

Réception

Types d'exceptions

Utilisation

Flux

Introduction

Exceptions : réception

L'appelant de la méthode doit soit :

- ▶ déclarer l'exception

```
public void test() throws FormatException  
{load("truc.txt");}
```

- ▶ traiter l'exception

```
public void test(){  
    try{  
        load("truc.txt");  
        System.out.println("Bon format");  
    }  
    catch (FormatException fe) {  
        System.out.println("Mauvais format :"+fe);  
    }  
}
```

Exceptions

Emission

Réception

Types d'exceptions

Utilisation

Flux

Introduction

Exceptions : réception

L'appelant de la méthode doit soit :

- ▶ déclarer l'exception

```
public void test() throws FormatException  
{load("truc.txt");}
```

- ▶ traiter l'exception

```
public void test(){  
    try{  
        load("truc.txt");  
        System.out.println("Bon format");  
    }  
    catch (FormatException fe) {  
        System.out.println("Mauvais format :"+fe);  
    }  
}
```

Exceptions

Emission

Réception

Types d'exceptions

Utilisation

Flux

Introduction

Exceptions : réception

Exceptions

Emission

Réception

Types d'exceptions

Utilisation

Flux

Introduction

L'appelant de la méthode doit soit :

- ▶ déclarer l'exception

```
public void test() throws FormatException  
{load("truc.txt");}
```

- ▶ traiter l'exception

```
public void test(){  
    try{  
        load("truc.txt");  
        System.out.println("Bon format");  
    }  
    catch (FormatException fe) {  
        System.out.println("Mauvais format :"+fe);  
    }  
}
```

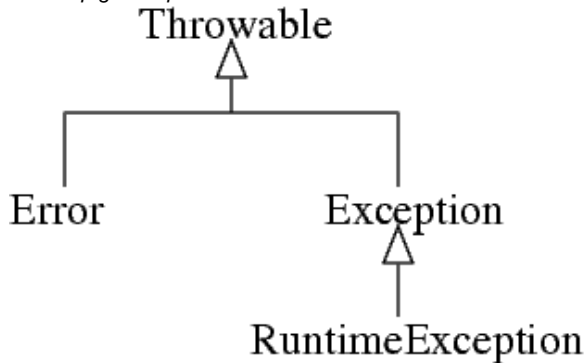
Runtime Exceptions et Erreurs

Les Exceptions sous classes de RuntimeException

ne nécessitent pas de throws ou de try/catch car elles peuvent être déclenchées par la plupart des actions "normales" de la JVM.

Les Erreurs

sont trop graves pour être traitées.



Exceptions

Emission

Réception

Types d'exceptions

Utilisation

Flux

Introduction

Exceptions classiques

Exception

- ▶ FileNotFoundException
- ▶ ClassNotFoundException

RuntimeException

- ▶ NullPointerException
- ▶ ArrayIndexOutOfBoundsException
- ▶ ClassCastException

Errors

- ▶ ClassFormatError
- ▶ VerifyError

Exceptions

Emission

Réception

Types d'exceptions

Utilisation

Flux

Introduction

Exceptions personnalisées

Pour créer d'autres classes d'exception, il suffit de créer une sous classe de la classe `Exception` et de définir ses deux constructeurs.

```
public class FormatException extends Exception{  
    public FormatException(){super();}  
    public FormatException(String msg){super(msg);}  
}
```

On peut aussi y ajouter des attributs et des méthodes d'accès si nécessaire.

Exceptions

Emission

Réception

Types d'exceptions

Utilisation

Flux

Introduction

Exceptions multiples

S'il y a plusieurs Exceptions récupérables

- ▶ il faut plusieurs blocs `catch`
- ▶ du plus spécifique au plus général

Exceptions

Emission

Réception

Types d'exceptions

Utilisation

Flux

Introduction

Exceptions multiples

S'il y a plusieurs Exceptions récupérables

- ▶ il faut plusieurs blocs `catch`
- ▶ du plus spécifique au plus général

Exceptions

Emission

Réception

Types d'exceptions

Utilisation

Flux

Introduction

Exceptions multiples

S'il y a plusieurs Exceptions récupérables

- ▶ il faut plusieurs blocs `catch`
- ▶ du plus spécifique au plus général

Exceptions

Emission

Réception

Types d'exceptions

Utilisation

Flux

Introduction

Déclenchement d'exceptions

- ▶ Lorsqu'un type de retour est déjà présent
- ▶ Dans un constructeur
- ▶ Lorsque l'erreur invalide l'exécution

Exceptions

Emission

Réception

Types d'exceptions

Utilisation

Flux

Introduction

Traitement d'exceptions

- ▶ par un try catch quand on peut traiter
 - ▶ adopter un comportement par défaut dans les classes de base
 - ▶ signaler le problème dans les classes d'interfaces
- ▶ en faisant remonter si on ne peut traiter
- ▶ ou en requalifiant l'exception

Exceptions

Emission

Réception

Types d'exceptions

Utilisation

Flux

Introduction

Entrées sorties non événementielles

- ▶ Lecture Ecriture Fichiers
- ▶ Lecture Ecriture Entrée/Sortie standard/erreur
- ▶ Lecture Scanner
- ▶ Ecriture carte son
- ▶ ...

Exceptions

Emission
Réception
Types d'exceptions
Utilisation

Flux

Introduction

Pour les flux d'octets : Input/Output Streams
Pour les flux de caractères Unicode Readers/Writers
Très grande variété, mais de nombreux traitements
communs et optionnels : Découplage

- ▶ Readers et Writers : 16 bit characters
- ▶ InputStream et OutputStream : 8 bit bytes

Sous classes : (pour Reader/Writers)

- ▶ lecture ecriture : CharArray, File, Piped, String
- ▶ opérations : Buffered, LineNumbered, Input/OutputStream, Filter

Exceptions

Emission
Réception
Types d'exceptions
Utilisation

Flux

Introduction

Composition des readers

- ▶ Transformer un flot d'octets en flot de caractères ;

```
InputStreamReader in =new InputStreamReader(System.in);
```

- ▶ Convertir un flot de caractères en flot bufferisé pour lire des String

```
BufferedReader br= new BufferedReader(new FileReader("myfile.txt"));
```

- ▶ Convertir un flot de caractères pour le parser

```
StreamTokenizer st= new StreamTokenizer(new  
FileReader("myfile.txt"));
```

Exceptions

- Emission
- Réception
- Types d'exceptions
- Utilisation

Flux

- Introduction

Programmation événementielle

Programmation dans laquelle les exécutions sont déclenchées par des événements, principalement initiés par l'utilisateur.

La modularité repose sur la distinction entre les

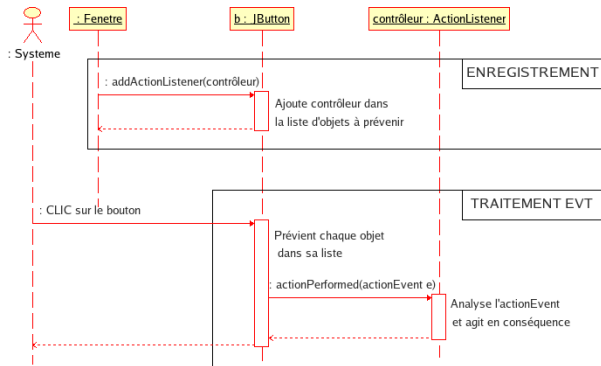
- ▶ Récepteurs d'évènements système (Boutons Clavier)
- ▶ Contrôleurs
- ▶ Reste de l'application

Pour greffer une gestion événementielle sur un programme objet il faut :

- ▶ Créer des contrôleurs qui analysent les évènements et appellent le code objet de base
- ▶ Collecter des sources d'évènements (les Recepteurs) et leur affecter des contrôleurs
- ▶ Reste de l'application

Attention : les récepteurs d'évènements système sont considérés comme des sources d'évènements par les contrôleurs.

Exemple



Le modèle événementiel

Introduction

événements

Ecouteurs

exemples

Les composants réactifs émettent des "événements" pour prévenir leurs contrôleurs ("écouters") qui réagissent en conséquence. pour chaque catégorie d'événement, on a :

- ▶ une classe événement

i.e MouseEvent

- ▶ un type d'écouteur associé

interface MouseListener

- ▶ une classe d'écouteur

class MouseAdapter implements MouseListener

Les composants réactifs doivent posséder une méthode permettant de leur associer un écouteur

addMouseListener

Les composants réactifs émettent des "événements" pour prévenir leurs contrôleurs ("écouters") qui réagissent en conséquence. pour chaque catégorie d'événement, on a :

- ▶ une classe événement

i.e MouseEvent

- ▶ un type d'écouteur associé

interface MouseListener

- ▶ une classe d'écouteur

class MouseAdapter implements MouseListener

Les composants réactifs doivent posséder une méthode permettant de leur associer un écouteur

addMouseListener

Les composants réactifs émettent des "événements" pour prévenir leurs contrôleurs ("écouters") qui réagissent en conséquence. pour chaque catégorie d'événement, on a :

- ▶ une classe événement
- ▶ un type d'écouteur associé

i.e MouseEvent

interface MouseListener

- ▶ une classe d'écouteur

class MouseAdapter implements MouseListener

Les composants réactifs doivent posséder une méthode permettant de leur associer un écouteur

addMouseListener

Les composants réactifs émettent des "événements" pour prévenir leurs contrôleurs ("écouters") qui réagissent en conséquence. pour chaque catégorie d'événement, on a :

- ▶ une classe événement

i.e MouseEvent

- ▶ un type d'écouteur associé

interface MouseListener

- ▶ une classe d'écouteur

class MouseAdapter implements MouseListener

Les composants réactifs doivent posséder une méthode permettant de leur associer un écouteur

addMouseListener

On associe un composant réactif et son "écouteur" :

```
monComposant.addMouseListener(monObjetEcouteur)
```

L'écouteur doit implementer l'interface :

```
public class mafenetre implements WindowListener{  
    public void windowOpened(WindowEvent e) {};  
    public void windowClosing(WindowEvent e) {};  
    public void windowClosed(WindowEvent e) {};  
    public void windowIconified(WindowEvent e) {};  
    public void windowDeiconified(WindowEvent e) {};  
    public void windowActivated(WindowEvent e) {};  
    public void windowDeactivated(WindowEvent e) {};  
    ...  
}
```

Il peut aussi être une sous-classe de WindowAdapter

pour ne redéfinir que la méthode qui nous intéresse.

Exemples

Quelques événements et leur nom (à suffixer par Event, Listener, Adapter)

ClicBouton	Action
ClicSouris	Mouse
Drag'n Drop	MouseMotion
ToucheClavier	Key
Fenêtre	Window

Le modèle événementiel

Introduction

événements

Ecouteurs

exemples

Un application graphique possède

- ▶ Un support (ex : une fenêtre)
- ▶ Des éléments réactifs (boutons, menus. . .)
- ▶ Une organisation des éléments sur la fenêtre (Layout)
- ▶ Des contrôleurs faisant le lien avec le reste de l'application

bibliothèques graphiques

- ▶ Abstract Windowing Toolkit
*java.awt.**
- ▶ Swing (Lightweight components)
*javax.swing.**
- ▶ SunWindowingToolkit :
platform specific implementations (pas de gc)
*org.eclipse.swt.widget.**
- ▶ versions propriétaires (pour des générateurs d'interface)

Interfaces Graphiques

Introduction

Composition

Exercices

Java 5.0

Threads

base

Multithreading

Composition

Principe : une ou des fenêtres (Frame/JFrame/Display) :
Dans lesquels on peut placer des composants :

- ▶ Button
- ▶ Label
- ▶ Text
- ▶ List
- ▶ Menu
- ▶ Containers...

Selon un `Layout` (placement)

les containers ont des méthodes `add(Component)`

les composants ont des méthodes `setSize(int, int)`

dans swing, il faut ajouter les composants à la `ContentPane` de la `JFrame`

dans SWT, il faut ajouter les composants à un `Shell` du `Display`

Interfaces Graphiques

Introduction

Composition

Exercices

Java 5.0

Threads

base

Multithreading

A faire

Créer les classes permettant

- ▶ l'affichage d'une fenetre d'Etat du Stock à partir d'un objet Cuisine.
Elle affichera une liste d'ingredients et leur quantité, ainsi qu'un bouton quitter.
- ▶ l'affichage d'une fenêtre présentant un Menu à partir d'un objet Menu.
Elle affichera un menu et son prix.
- ▶ l'affichage d'une fenêtre de Création de menu à partir d'un Chef
elle comportera un label avec le nom du chef et un bouton composer qui
générera des fenetres affichant le menu composé.

Créer un main comportant une phase d'initialisation des données puis
affichant fenêtre de stock et fenêtre de création de menu .

Interfaces Graphiques

Introduction
Composition
Exercices

Java 5.0

Threads

base
Multithreading

Objets réactifs

On veut que la création d'un menu "préviene" la fenetre d'Etat du stock pour qu'elle se mette à jour. **Mais sans toucher au code de Menu, Ingrédient, Chef, Cuisine, ni à celui de FenetreChef et FenetreMenu.** La solution suppose une réponse à ces questions :

- ▶ quels objets doivent devenir "réactifs" et emettre les messages ?
- ▶ Les objets réactifs sont ils obtenus par composition ou héritage ?
- ▶ Quand doivent ils prévenir l'écouteur et comment ?
- ▶ Comment introduire les nouveaux objets ?
- ▶ Quand les lier à l'écouteur ?
- ▶ Qui est l'écouteur ? que doit il faire ?

Permettre ensuite de prévenir un nombre quelconque d'"écouteurs"

Interfaces Graphiques

Introduction
Composition
Exercices

Java 5.0

Threads

base
Multithreading

nouvelle boucle

nouvelle boucle for sur les collections et les tableaux

for (TypeContenu nomvariable :
variablecollection<TypeContenu>) { }

```
for (Personne each: lesgens) each.vieillirDe(4);
```

Attention, limitations

- ▶ On ne peut modifier la collection dans la boucle, mais seulement les éléments qu'elle contient.
- ▶ On ne peut parcourir qu'une collection à la fois pour chaque **for**.

Interfaces
Graphiques

Introduction
Composition
Exercices

Java 5.0

Threads

base
Multithreading

On ne peut utiliser des types primitifs (i.e. non objets) comme `int`, `boolean`, là où le programme attend des objets.

Mais pour faciliter l'usage des collections, ils sont convertis automatiquement en leur équivalent `Objet` (et réciproquement).

```
int i=82;
int j;
Vector<Integer> notes= new Vector<Integer>( );

notes.add(i);
j=notes.get(0);
```

C'est une facilité à n'utiliser que dans ce cas car il est bien plus rapide d'utiliser les types primitifs pour du calcul.

```
public enum Jours LUNDI, MARDI, MERCREDI, JEUDI,  
VENDREDI, SAMEDI, DIMANCHE
```

utilisation classique comme constantes mais avec les avantages suivants :

- ▶ type spécifique (pas de conflits)
- ▶ `toString()` renvoie leur nom (et pas un int)
- ▶ la méthode de classe `.values()` renvoie une collection de tous les éléments

Utilisation avancée : on peut définir un enum comme une classe

- ▶ avec des attributs (ce qui oblige à faire un constructeur, souvent `private`)
- ▶ avec des méthodes, éventuellement redéfinies pour chaque élément

- ▶ EnumSet génère des ensembles d'enums
 - ▶ EnumSet.of(Style.BOLD,Style.ITALIC).contains(monstyle)
vérifie que la variable monstyle est soit BOLD soit ITALIC
 - ▶ EnumSet.range(Jours.LUNDI,Jours.VENDREDI)
renvoie la collection correspondante
- ▶ EnumMap permet d'associer des objets à un enums
 - ▶ EnumMap<Jour,Nourriture> platdujour =new EnumMap<Jour,Nourriture>();
 - ▶ platdujour.put(Jour.LUNDI,new Nourriture("Ravioli"));

Threads

Un programme Java peut comporter plusieurs threads, c'est à dire :

- ▶ plusieurs séquences d'instructions
- ▶ ayant chacune un point d'exécution propre
- ▶ partageant un environnement

Interfaces Graphiques

Introduction
Composition
Exercices

Java 5.0

Threads

base

Multithreading

Threads

- ▶ création et utilisation
- ▶ cycle de vie d'un thread
- ▶ priorités
- ▶ partage de données, synchronisation

Interfaces Graphiques

Introduction
Composition
Exercices

Java 5.0

Threads

base

Multithreading

Création et Utilisation

- ▶ sous classe de `java.lang.thread`
- ▶ implanter l'interface `java.lang.Runnable`

La boucle de vie d'un thread se trouve dans la méthode `run`, qui est appelée par la méthode `start`.

Exercice : Créer deux threads affichant leur nom suivi de nombres croissants. Lancez les à partir d'un programme principal. Pour que ça n'aille pas trop vite, faites un très grand nombre d'itérations mais n'affichez que toutes les 1000.

Interfaces Graphiques

Introduction
Composition
Exercices

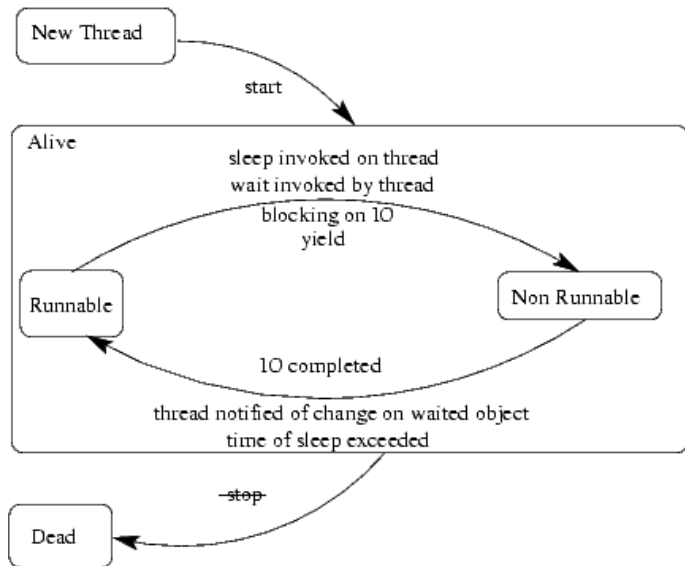
Java 5.0

Threads

base

Multithreading

Cycle de vie



Interfaces Graphiques

Introduction
Composition
Exercices

Java 5.0

Threads

base
Multithreading

Multi-threading

Il ne peut y avoir plus de threads en exécution que de processeurs. Le temps est donc partagé entre les threads. Le système (ici la machine virtuelle)

- ▶ élit un thread (en général en se basant sur sa priorité et sur un historique),
- ▶ alloue un quantum de temps à un thread
- ▶ jusqu'à ce que son temps soit écoulé ou que le thread devienne non-Runnable

Interfaces Graphiques

Introduction
Composition
Exercices

Java 5.0

Threads

base
Multithreading

Politesse

Un thread doit prévoir de rendre la main avant que l'on ne l'y oblige en utilisant la méthode `yield`, qui rend la main au système afin qu'il décide à qui attribuer le prochain quota de temps.

Inclure des `yields` conditionnels dans la boucle de vie d'un thread est un moyen non centralisé d'influer sur la politique d'allocation des threads. **Exercice** : Ajoutez une méthode `ajustePolitesse(int)` à vos threads précédents qui leur fait céder la place plus régulièrement. Testez.

Interfaces Graphiques

- Introduction
- Composition
- Exercices

Java 5.0

Threads

- base
- Multithreading

Partage et Synchronisation

Si deux threads manipulent le même objet en même temps, ce dernier peut se retrouver dans un état incohérent car l'entrelacement des instructions qui composent ses méthodes n'a pas les mêmes propriétés que des invocations successives de méthodes.

int j=attribut ; attribut=j+1 ; int j=attribut ; attribut=j+1 ;	int j=attribut ; int j=attribut ; attribut=j+1 ; attribut=j+1 ;
--	--

Il faut

- ▶ empêcher deux exécutions simultanées
- ▶ gérer l'attente provoquée

Interfaces Graphiques

Introduction
Composition
Exercices

Java 5.0

Threads

base
Multithreading

Méthodes synchronisées

L'appel à une méthode synchronisée bloque l'objet qui la reçoit durant son exécution. On ne peut faire appel à une méthode synchronisée sur un objet bloqué, sauf si on est à l'origine du blocage.

Syntaxe : synchronized

Exemple :

```
public void synchronized raiseQte(int n){  
    int j=this.qte; this.qte=j+n;  
}
```

Interfaces Graphiques

- Introduction
- Composition
- Exercices

Java 5.0

Threads

- base
- Multithreading

gestion de l'attente

Lorsque l'exécution d'une méthode dépend d'un état de l'objet, il faut :

- ▶ la rendre synchronized
- ▶ attendre avec `wait()` dans une boucle testant l'état
- ▶ rendre les méthodes de changement d'état synchronized
- ▶ faire un `notify` ou un `notifyAll` dans les méthodes de changement d'état

Le framework java a beaucoup évolué depuis la version 1.5 et va nettement plus loin maintenant.

Interfaces Graphiques

Introduction
Composition
Exercices

Java 5.0

Threads

base
Multithreading