Programmation 1

Licence 1 – Portail René Descartes

2018 – 2019

Aix*Marseille université
Initiative d'excellence

Première partie I

Introduction à la programmation

Aix*Marseille université
Initiative d'excellence

Qu'est-ce que la programmation ?

La programmation

C'est l'activité consistant à anticiper, planifier, organiser un ensemble de tâches, dans un langage compréhensible par l'humain et la machine, afin de fournir un service ou de répondre à une besoin.

Aix*Marseille université
Initiative d'excellence

Fonctionnement d'un programme typique

1. L'environnement ou un utilisateur produit des stimuli, via un *périphérique* (clavier, souris, carte réseau, capteurs, mémoire, un autre programme). Ce sont les *entrées* du programme.
2. Ces stimuli sont numérisés, représentés par du texte.
3. Le programme analyse ces données textuelles, en utilisant des représentations structurées, et produit des résultats représentés par du texte : les *sorties*.
4. Les sorties sont envoyées sur les périphériques, pour provoquer un affichage, un son, un processus mécanique ou l'envoi d'un message sur un réseau, ou pour être stockées.

Aix*Marseille université
Initiative d'excellence

Programmer : une activité de long terme.

- Analyser les besoins.
- Spécifier les comportements du programme.
- Concevoir des méthodes de résolution (des *algorithmes*).
- Implémenter le programme (vulgairement : *coder*).
- Vérifier le bon comportement du programme (*tester*).
- Déployer le programme dans son environnement.
- Maintenir le programme (corriger des défaillances, ajouter des fonctionnalités).

Aix*Marseille université
Initiative d'excellence

Une multitude de formes

Exemples de programmes :

- lecteur média,
- navigateur Web,
- application de smartphone,
- contrôleur de production industrielle,
- service web,
- système d'exploitation,
- compilateur, ...

Aix*Marseille université
Initiative d'excellence

Une multitude de langages

Différents langages existent pour répondre à tous les besoins, voici quelques familles de langages :

- procédural (séquence d'instructions),
- déclaratif (définitions de valeurs),
- logique (collection de contraintes),
- fonctionnel (composition de fonctions),
- objet (composants en interaction),

Aix*Marseille université
Initiative d'excellence

La programmation objet

Les objectifs :

- faciliter le développement et l'évolution des programmes,
- permettre le travail en équipe,
- augmenter la qualité des logiciels.

Comment ?

- découpler (séparer) les parties du programme,
- limiter et localiser les modifications lors des évolutions du programme,
- permettre de réutiliser les composants du programme.

Aix*Marseille université
Initiative d'excellence

Le langage Java

Le langage utilisé dans ce cours est Java :

- langage orienté objet,
- robuste et sûr, fonctionnant sur toute plateforme,
- performant,
- très populaire dans l'industrie,
- disposant d'excellents outils et bibliothèques.

Java intègre les mêmes concepts que la plupart des langages de programmation.

Aix*Marseille université
Initiative d'excellence

Un exemple d'application

On souhaite réaliser une application permettant d'affecter des étudiants à des cours d'option :

- chaque cours a une capacité maximale,
- chaque étudiant a une liste de préférences sur les cours qu'il souhaite suivre,
- chaque étudiant doit s'inscrire à un cours,
- la liste des cours, la liste des étudiants et de leurs préférences sont disponibles sous la forme de fichiers de données,
- nous souhaitons obtenir un fichier de données précisant pour chaque cours la liste des étudiants inscrits.

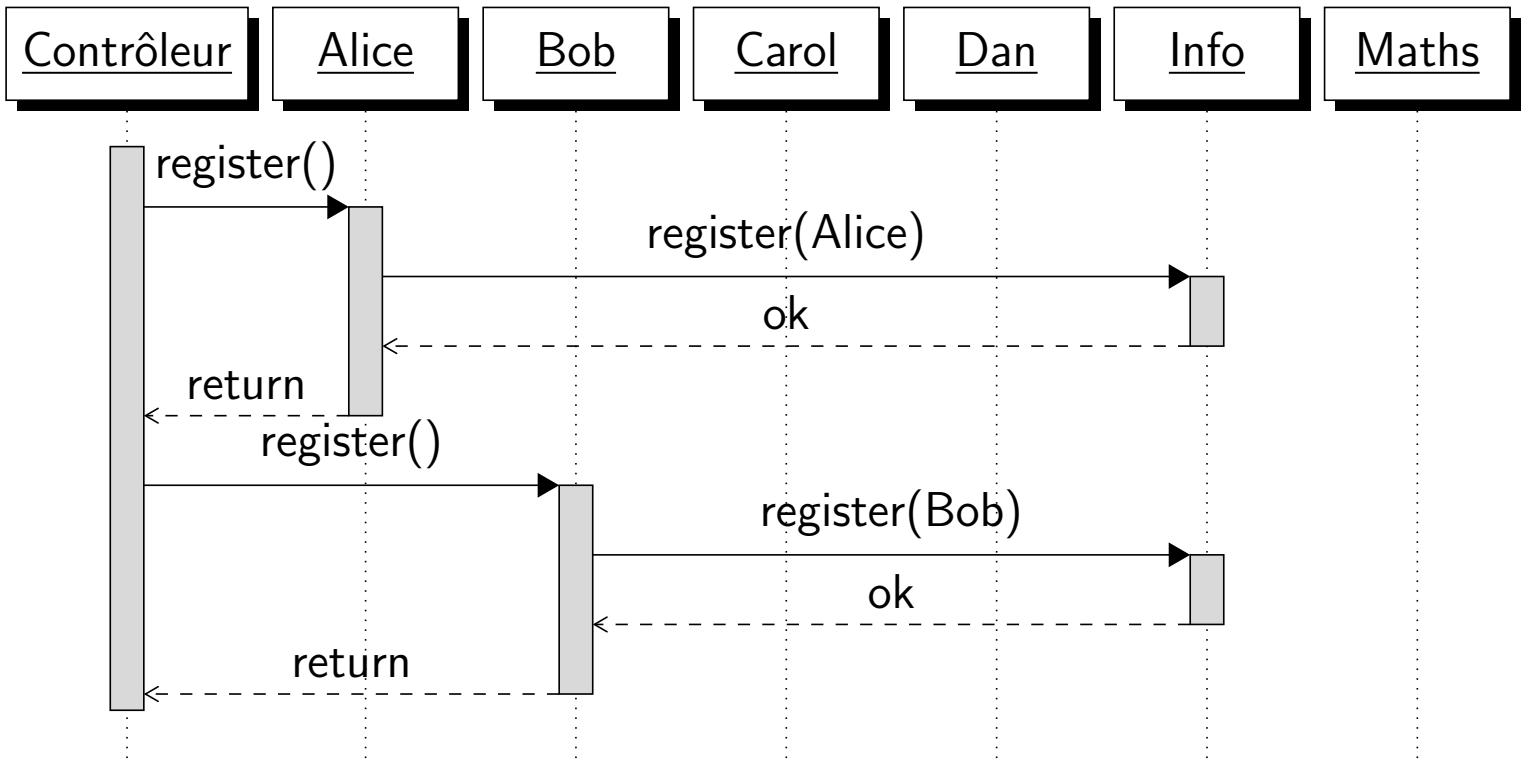
Aix*Marseille université
Initiative d'excellence

Les éléments de cette tâche

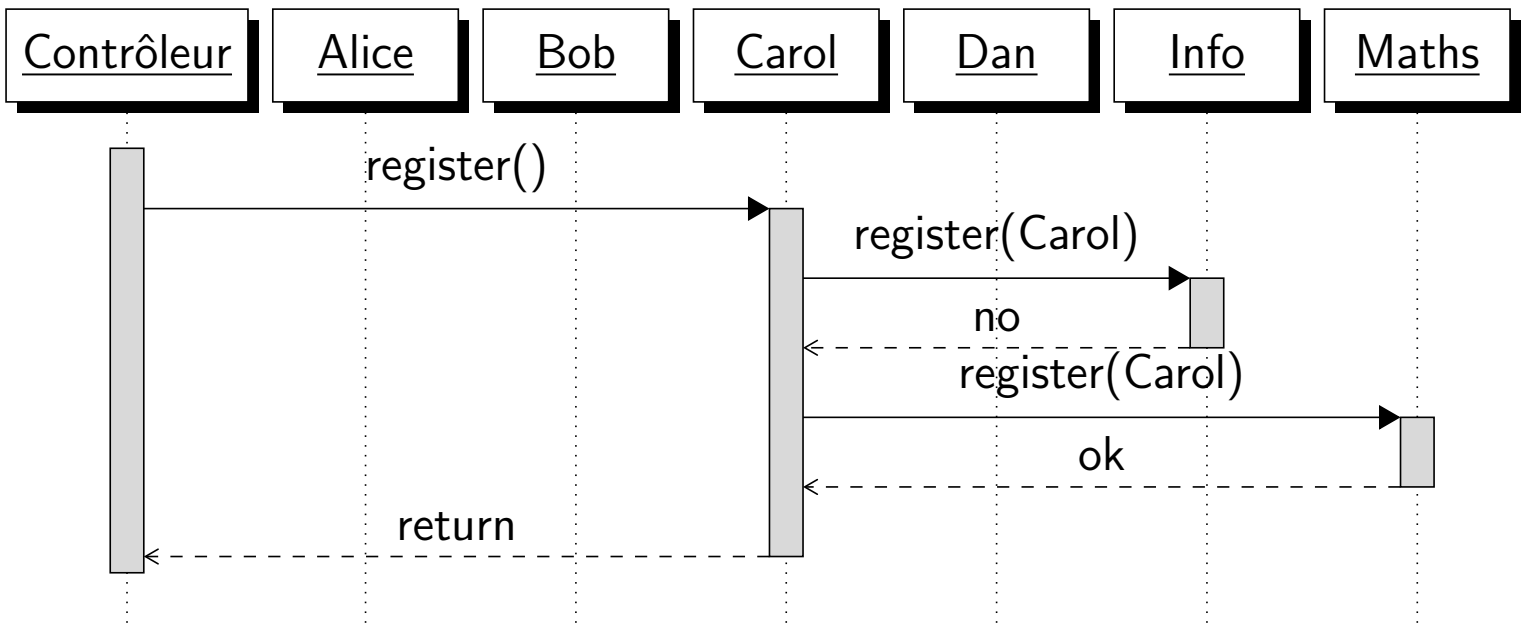
- des étudiants,
- des cours,
- des listes de préférences,
- des listes d'inscrits,
- des fichiers de données à lire ou écrire.

Chaque élément va être représenté comme un objet.

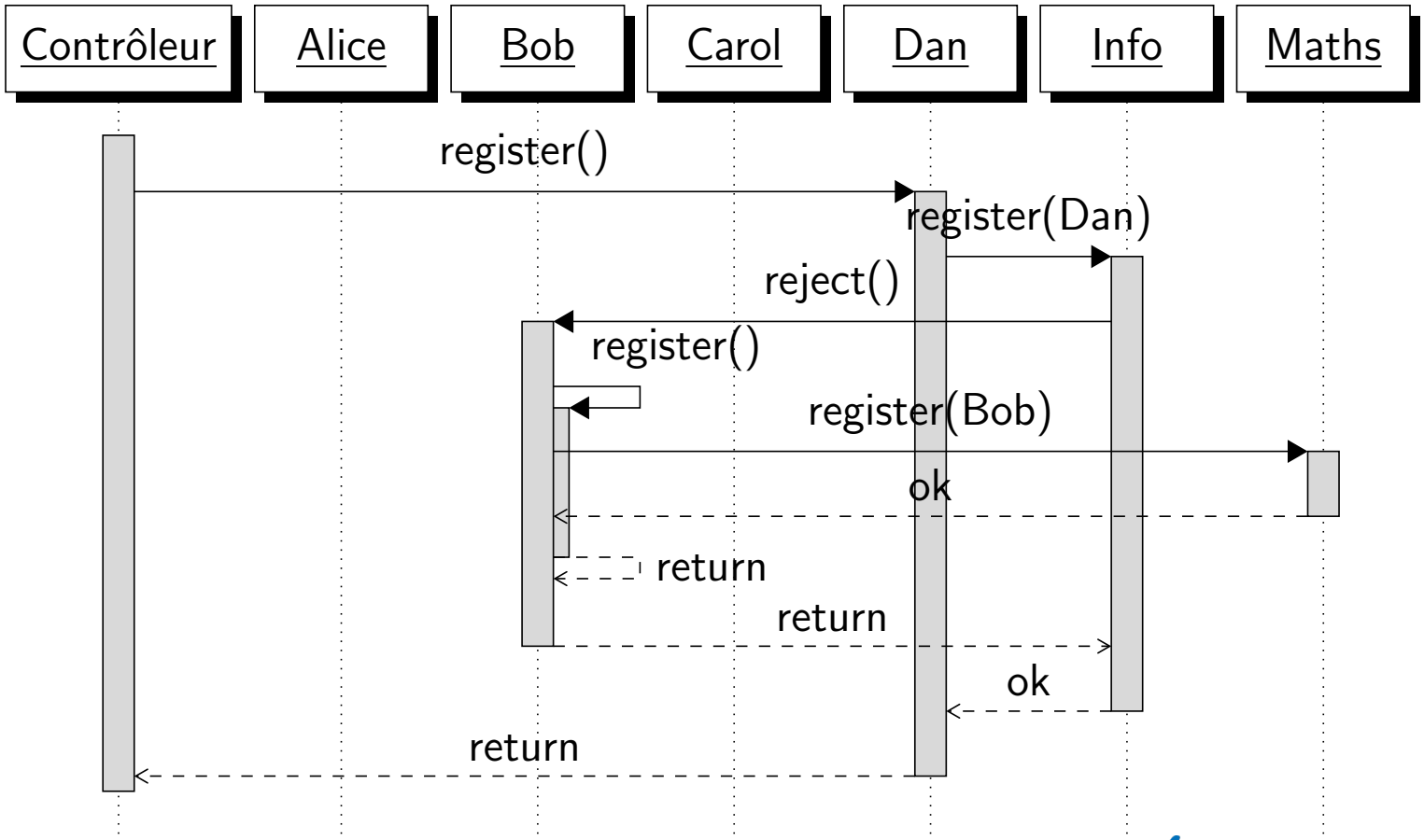
Exemple de processus



Exemple de processus



Exemple de processus



Comment maîtriser cette complexité ?

En programmation objet :

- Le comportement du programme est le résultat de la communication des objets entre eux.
- Les interactions produites sont en général trop complexes pour être appréhendées par un être humain.

Mais on réfléchit uniquement au niveau de l'objet :

- Chaque objet doit avoir un seul rôle, une seule responsabilité.
- Chaque objet doit avoir un comportement simple, facilement compréhensible par l'humain.
- Chaque objet doit limiter ses interactions avec les autres objets au nécessaire pour remplir son rôle.

Aix*Marseille université
Initiative d'excellence

Un étudiant

Un étudiant possède :

- un nom,
- une liste de préférences de cours.

Et il peut :

- s'inscrire à un cours,
- se faire désinscrire d'un cours.

Aix*Marseille université
Initiative d'excellence

Un cours

Un cours possède :

- une capacité,
- une liste d'étudiants inscrits.

Et il peut :

- traiter l'inscription d'un étudiant,
- fournir la liste des étudiants inscrits.

Aix*Marseille université
Initiative d'excellence

Une liste de préférences

Une liste de préférences de cours possède :

- une séquence ordonnée de cours.

Et elle peut :

- fournir le prochain cours dans la liste,
- déclarer qu'il n'y a plus de cours sur la liste.

Les objets

Étudiants, cours et listes de préférences ont donc :

- des propriétés (le nom de l'étudiant, la capacité du cours, la séquence de cours de la liste de préférences, . . .),
- des comportements (s'inscrire à un cours, fournir la liste des étudiants inscrits, fournir le prochain cours dans l'ordre de préférence).

Un objet

Un objet est une entité composée de propriétés (ou champs), et de comportements (ou méthodes).

Aix*Marseille université
Initiative d'excellence

L'étudiante Alice

Propriétés :

- **nom :** "Alice"
- **préférences :** Informatique, Mathématiques, Physique, Mécanique.

Comportements :

- **s'inscrire :** s'inscrit au premier cours de sa liste de préférences duquel elle n'a pas encore été rejetée, jusqu'à ce que son inscription soit acceptée (ou avoir épuisé tous les cours).
- **se faire rejeter d'un cours :** s'inscrire.

Aix*Marseille université
Initiative d'excellence

L'étudiant Bob

Propriétés :

- **nom :** "Bob"
- **préférences :** Informatique, Mécanique, Physique, Mathématiques.

Comportements :

- **s'inscrire :** s'inscrit au premier cours de sa liste de préférences duquel il n'a pas encore été rejeté, jusqu'à ce que son inscription soit acceptée (ou avoir épuisé tous les cours).
- **se faire rejeter d'un cours :** s'inscrire.

Aix*Marseille université
Initiative d'excellence

Le cours de mécanique

Propriétés :

- **titre :** “Mécanique”
- **capacité :** 40 étudiants
- **étudiants inscrits :** initialement aucun

Comportements :

- **inscrire l’étudiant** `student` **:** si le nombre d’étudiants inscrits est égal à la capacité, rejeter la demande d’inscription. Sinon ajouter `student` aux étudiants inscrits.
- **fournir la liste des étudiants :** retourner la liste des étudiants.

Aix*Marseille université
Initiative d'excellence

Le cours d'Informatique

Propriétés :

- **titre :** "Informatique"
- **capacité :** 60 étudiants
- **étudiants inscrits :** initialement aucun

Comportements :

- **inscrire l'étudiant** `student` **:** si le nombre d'étudiants inscrits est inférieur à la capacité, ajouter `student` aux étudiants inscrits. Sinon, inscrire `student` et désinscrire un étudiant choisi par tirage au sort.
- **fournir la liste des étudiants :** retourner la liste des étudiants.

Aix*Marseille université
Initiative d'excellence

Les classes

- Alice et Bob ont les mêmes comportements et les mêmes propriétés, seules les valeurs de leurs propriétés sont différentes. Alice et Bob appartiennent à la même classe.
- Mécanique et Informatique ne se comportent pas de la même façon, ils ne sont donc pas dans la même classe. Cependant ils proposent la même liste de services, ils ont donc une interface commune.

Aix*Marseille université
Initiative d'excellence

La classe des étudiants

Propriétés :

- **nom :** une chaîne de caractères.
- **préférences :** une liste de cours.

Comportements :

- **s'inscrire :** s'inscrit au premier cours de sa liste de préférences duquel `this` n'a pas encore été rejeté, jusqu'à ce que son inscription soit acceptée (ou avoir épuisé tous les cours).
- **se faire rejeter d'un cours :** s'inscrire.

Construction : nécessite de fournir un nom et une liste de préférence.

Aix*Marseille université
Initiative d'excellence

Qu'est-ce qu'une classe ?

Une classe

Une classe (d'objets) définit :

- une façon de créer des objets (constructeur),
- la structure de ces objets (propriétés),
- le comportement de ces objets (méthodes).

En Java, un programme est constitué de fichiers, chacun définissant une classe.

Aix*Marseille université
Initiative d'excellence

La classe `Student` : déclaration

```
public class Student {

   ... /* see next slides */

}
```

- `class` introduit une classe,
- `Student` est le nom donné à cette classe,
- `public` précise que cette classe est connue de tous les objets.
- les `{ }` délimitent le contenu de la classe.

Aix*Marseille université
Initiative d'excellence

La classe `Student` : propriétés

```
public final String name;
private final Iterator<Course> courses;
private Course currentCourse;
```

- `name`, `courses`, `currentCourse` sont les trois propriétés des objets de la classe `Student`
- `String`, `Iterator<Course>` et `Course` sont les classes des objets de chaque propriétés.
- `public` ou `private` indique si la propriété est connue de tous les objets ou seulement ceux de la classe.
- `final` indique que la propriété ne change jamais de valeur après la construction.

Aix*Marseille université
Initiative d'excellence

La classe `Student` : constructeur

```
public Student(String studentName,
               Iterator<Course> preferredCourses) {
  this.name = studentName;
  this.courses = preferredCourses;
}
```

- Entre parenthèses, les paramètres transmis lors de la création de l'objet.
- `this.name` est la propriété `name` de l'objet en construction `this`.
- `this.name = studentName` initialise la propriété `name` de l'objet en construction à la valeur `studentName` donnée au constructeur.

Aix*Marseille université
Initiative d'excellence

La classe Student : isRegistered

```
public boolean isRegistered() {
  return currentCourse != null;
}
```

- `isRegistered` est une méthode, elle retourne un message vrai ou faux (`boolean`) à l'appelant,
- `public` indique que tout objet peut l'appeler,
- `return` permet de définir le message retourné.

La classe Student : register

```
public void register() {
  if (isRegistered() || !courses.hasNext())
    return;
  currentCourse = courses.next();
  boolean isAccepted =
    currentCourse.registerStudent(this);
  if (!isAccepted) {
    reject();
  }
}
```

La classe Student : reject

```
public void reject() {
  currentCourse = null;
  register();
}
```

- `null` est une absence de valeur,
- un objet peut appeler l'une de ses méthodes (ici `register`).

Deuxième partie II

Objets et méthodes

Objet, classe : rappel

Un objet

Un objet est une entité composée de propriétés (ou champs), et de comportements (ou méthodes).

Une classe

Une classe (d'objets) définit :

- une façon de créer des objets (constructeur),
- la structure de ces objets (propriétés),
- le comportement de ces objets (méthodes).

Aix*Marseille université
Initiative d'excellence

Différence classe vs objet

Bob est un objet :

- c'est une entité concrète (il porte un nom),
- ses propriétés lui sont propres,
- on peut lui envoyer des messages,
- la plupart des objets sont instances d'une classe.

Student est une classe :

- c'est un concept abstrait (il n'a pas d'existence, on ne peut pas lui envoyer de messages),
- plusieurs entités (objets) peuvent provenir de cette classe (Alice, Bob, Carol,...). Ce sont ses instances.

Exemple : "Voiture" est un concept, la voiture immatriculée 123AB56 est un objet.

Aix*Marseille université
Initiative d'excellence

Définir une classe

- Établir son unique responsabilité.
- Lister ses services.
- Spécifier le comportement des services (par exemple en préparant des tests).
- Lister les propriétés nécessaires pour accomplir ses services.
- Choisir son nom, reflétant son rôle.

Ensuite on peut implémenter.

Services/méthodes

Un objet rend des services aux autres objets.

Plusieurs sortes de services :

- ordre,
- requête d'information (l'objet fournit de l'information),
- envoi d'information (l'objet reçoit de l'information),
- test.

Tous sont rendus par des appels de méthodes.

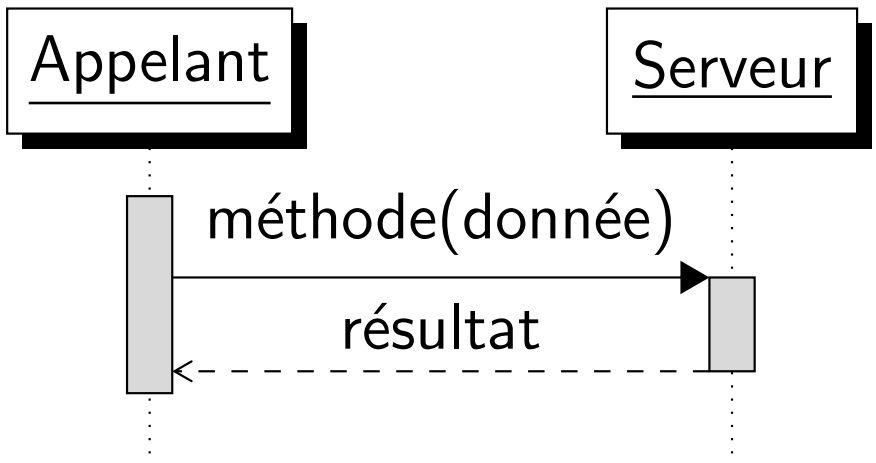
L'appel de méthode

Les étapes d'un appel de méthode. Deux objets en interaction, l'appelant (ou client) et l'appelé (ou serveur).

1. Une méthode du client a besoin d'un service,
2. le client précise qui est le serveur, quel service utiliser, et quelle information fournir au serveur pour rendre le service,
3. puis le client s'interrompt, se met en attente, pendant que le serveur travaille,
4. le serveur termine et renvoie un message en retour,
5. le client reçoit le message en retour et reprend son propre travail.

Aix*Marseille université
Initiative d'excellence

L'appel de méthode



Écrire un appel de méthode

Informations nécessaires :

- qui rend le service ?
- quel est le nom du service ?
- quelle information demande le service ?

Syntaxe Java :

```
serverName.methodName(data1,data2,...);
```

Aix*Marseille université
Initiative d'excellence

L'appel à soi-même

Un objet peut appeler l'un de ses services.

L'objet se désigne lui-même sous le nom `this`.

```
this.methodName(data1, data2);
```

Ou équivalent :

```
methodName(data1, data2);
```

Aix*Marseille université
Initiative d'excellence

Exemple d'utilisation de this

```java
// in Submarine.java

private void dive(int requiredFeetDepth) {
  if (this.feetDepth() >= requiredFeetDepth) {
    return;
  }
  this.moveDown();
  this.dive(requiredFeetDepth);
}
```

Aix*Marseille université
Initiative d'excellence

Paramètres des méthodes

Certaines méthodes ne nécessitent pas d'information :

```
list.isEmpty();
canvas.repaint();
robot.turnLeft();
```

D'autres sont paramétrables par des données :

```
phone.call(number);
robot.moveForward(distance);
submarine.launchMissile(target);
```

Aix*Marseille université
Initiative d'excellence

Paramètres des méthodes

```
submarine.launchMissile(target);
```

- `submarine` est le serveur,
- `launchMissile` est le service,
- `target` est un argument ou paramètre.

Un paramètre

Les paramètres permettent à l'appelant de préciser comment un service doit être rendu.

Paramètres multiples

Une méthode peut avoir zéro, un ou plusieurs paramètres.

```
canvas.repaint(); // no parameter
phone.call(number); // one parameter
mailer.send(address, message); // two parameters
```

L'ordre et le nombre des paramètres importent !

Aix*Marseille université
Initiative d'excellence

Connaître les paramètres

Quels sont les paramètres attendus, dans quel ordre ?

Dans la définition de la méthode :

```
public void send(Address address, String message) {
   ...
}
```

Deux paramètres : une adresse puis une chaîne de caractères.
L'IDE peut vous fournir cette information (`Ctrl-Q`)

Contrats

```
public void send(Address address, String message) {...}
```

Toute définition de méthode définit un contrat : le service est rendu seulement si tous les paramètres, de la bonne nature, sont fournis.

Erreur de types

Si les paramètres fournis ne sont pas de même nature que les paramètres requis, le programme est faux, il ne peut être ni compilé, ni exécuté.

Aix*Marseille université
Initiative d'excellence

Surcharge

Certains services peuvent avoir plusieurs contrats différents.

```
String text = "Hello World!";
// substring(int startIndex)
text.substring(6); // "World!";
// substring(int startIndex, int endIndex);
text.substring(6,10); // "World";
```

On parle alors de surcharge.

Retour d'une méthode

Pour des méthodes :

- de requête d'information,
- de test,

la méthode doit retourner une information.

La nature de l'information retournée fait partie du contrat : une méthode renvoie toujours des informations de même nature.

Exemples de retour de méthodes.

Certaines méthodes ne retournent pas d'information :

```
robot.moveForward(distance);
canvas.repaint();
```

D'autres retournent exactement une information :

```
Boolean shoppingIsOver = shoppingList.isEmpty();
String prefix = text.substring(0,10);
Double height = image.getHeight();
```

Connaître le retour d'une méthode.

De quelle nature est le message retourné par une méthode ?

Dans la définition de la méthode :

```
public void send(Address address, String message) {
  ...
  }
```

`void` : pas de message retourné.

```
public Boolean isEmpty() { ... }
```

`Boolean` : retourne un booléen (`true` ou `false`).

L'IDE peut vous la fournir (`Ctrl-Q`)

Aix*Marseille université
Initiative d'excellence

Message de retour et contrat

La nature du message retourné fait partie du contrat d'une méthode. Si les paramètres respectent le contrat, un message de la nature imposée par le contrat est retourné.

```
public String substring(int startIndex) { ... }
```

`substring` retourne une chaîne de caractères (`String`).

Définir une nouvelle méthode.

Pour définir une nouvelle méthode, il faut :

- déterminer le service rendu,
- trouver un nom correspondant au service rendu,
- lister les paramètres nécessaires,
- déterminer la nature du message retourné,
- décider si le service est utilisable par tous (`public`) ou seulement par l'objet lui-même (`private`).

Puis on implémente.

Aix*Marseille université
Initiative d'excellence

Syntaxe d'une définition de méthode.

```
public ReturnKind methodName(DataKind1 data1,
                             DataKind2 data2) {
    ...
}
```

- `public` (service disponible pour tous) ou `private` (seulement disponible pour cette classe),
- `ReturnKind` ou `void` : quelle nature de message retourné,
- `DataKind1`, `DataKind2` : nature des paramètres,
- `data1`, `data2` : paramètres formels

Exemple de méthode

```
public Boolean launchMissile(Target target) {
  if (!hasAvailableMissile()) return false; // abort
  if (isDiving()) surface();
  GPSCoordinate position = target.getPosition();
  targettingSystem.setTarget(position);
  tube.loadMissile();
  tube.open();
  tube.fire();
  tube.close();
  return true; // success!
}
```

Nommage des méthodes

Le nom d'une méthode doit indiquer :

- de quelle sorte de service il s'agit (ordre, requête d'information, test d'une propriété),
- quelle action est effectuée par le service,
- quelle information est retournée par le service.

(mais pas comment le service est rendu)

Pourquoi ?

- rendre le programme compréhensible par un humain,
- s'assurer que la méthode a un rôle clair, simple et bien défini,
- créer une cohérence grammaticale au niveau de l'appel de la méthode.

Aix*Marseille université
Initiative d'excellence

Règles de nommage des méthodes

- Interdiction d'utiliser des noms sans signification, des abbréviations incompréhensibles.
- Java et ses librairies sont en anglais, par cohérence on choisit des termes anglais.
- **méthode d'action** : groupe verbal à l'infinitif (`send`, `launchMissile`).
- **méthode de requête d'information** : groupe nominal désignant l'information requise (`height`, `substring`).
- **méthode de test de propriété** : groupe prédicatif au présent (`isEmpty`, `hasAvailableMissile`).

Aix*Marseille université
Initiative d'excellence

Troisième partie III

Propriétés, variables, constructeurs

Objet, classe : rappel

Un objet

Un objet est une entité composée de propriétés (ou champs), et de comportements (ou méthodes).

Une classe

Une classe (d'objets) définit :

- une façon de créer des objets (constructeur),
- la structure de ces objets (propriétés),
- le comportement de ces objets (méthodes).

Aix*Marseille université
Initiative d'excellence

Propriétés des objets

Un objet possède des propriétés.

Exemple : une voiture

- une marque, un modèle,
- un numéro de série (une chaîne de caractères),
- une position (une coordonnée),
- une direction (un angle),
- une vitesse (un nombre),
- un niveau d'essence (un nombre),
- l'état d'allumage des phares (un booléen),
- ...

Aix*Marseille université
Initiative d'excellence

Description d'une propriété.

Une propriété possède donc

- un rôle (quantifier la vitesse),
- un nom (*speed*),
- une nature (un nombre),
- une valeur ($60\mathrm{km.h^{-1}}$).

Une propriété peut

- soit changer au cours du temps (mutable),
- soit ne jamais être modifiée (immutable).

Pourquoi des propriétés ?

Un objet a une responsabilité. Ses propriétés lui permettent de satisfaire sa responsabilité.

- Un objet ne doit pas avoir de propriété qui ne serve pas à sa responsabilité.
- Certains objets ont pour responsabilité de représenter une valeur (par exemple un vecteur 2D, une couleur, ...) ; leur responsabilité est d'avoir des propriétés (les coordonnées x et y, une représentation RGB, ...).

Aix*Marseille université
Initiative d'excellence

Nommage des propriétés

Comme les méthodes, les propriétés doivent avoir des noms bien choisis.

- En anglais (cohérence),
- rappelant le rôle de la propriété,
- pas d'abbréviation inusuelle,
- éviter les variables numérotées `a1`, `a2`, `a3`... par manque d'imagination,
- essayez toujours d'être explicite.

Droits sur une propriété.

Comme les méthodes, une propriété peut être :

- **privée :** seul sa classe peut y accéder ou modifier sa valeur,
- **publique :** tout objet peut y accéder et modifier sa valeur.

Danger

Ce n'est pas à l'automobiliste derrière moi de fixer ma vitesse !

Portée d'une propriété

Une propriété est de la responsabilité de l'objet ayant cette propriété.

Principes généraux (sauf exception) :

- une propriété ne devrait être modifiée que par l'objet la possédant,
- une propriété mutable sera le plus souvent gardée privée,
- une propriété immutable peut être rendue publique, s'il est légitime pour un objet externe de la connaître.

Portée d'une propriété

Exemples :

- **couleur de la carrosserie :**
- **niveau d'essence :**
- **immatriculation :**
- **état des clignotants :**

Portée d'une propriété

Exemples :

- **couleur de la carrosserie :** publique, immutable,
- **niveau d'essence :** privé, mutable,
- **immatriculation :** publique, immutable,
- **état des clignotants :** privé, mutable,

Portée d'une propriété

Exemples :

- **couleur de la carrosserie :** publique, immutable,
- **niveau d'essence :** privé, mutable,
- **immatriculation :** publique, immutable,
- **état des clignotants :** privé, mutable,

Pour les clignotants, on offre en plus une méthode publique permettant de tester leur état.

Choix de portée d'une propriété

Les principaux choix sont donc :

- propriété immutable, publique ou privée (numéro d'immatriculation, modèle et couleur du véhicule),
- propriété mutable privée (niveau d'essence, température du moteur),
- propriété mutable privée avec méthode publique pour consulter sa valeur (position, vitesse, clignotants).

Pourquoi restreindre les portées ?

Principe de l'encapsulation : l'objet a sa responsabilité, et s'isole pour n'offrir que les services nécessaires.

- découper le programme en unités élémentaires simples,
- réduire les dépendances entre ces unités,
- garder une organisation compréhensible,
- cacher les détails internes non-pertinents pour le client,
- protéger l'objet d'interférences non-désirées.

Simplifier !

Aix*Marseille université
Initiative d'excellence

Nature des propriétés

Une propriété a toujours une nature : nombre, texte, couleur, objets rendant tel ou tel service.

Type

Un type désigne une catégorie de valeurs ou d'objets rendant les mêmes services (mais pas nécessairement avec le même comportement).

Aix*Marseille université
Initiative d'excellence

Exemples de types

- `int` (ou `Integer`), `long` (ou `Long`) : les entiers (précision bornée),
- `float` (ou `Float`), `double` (ou `Double`) : les nombres à virgule flottante,
- `Boolean` : les valeurs de vérités,
- `String` : les chaînes de caractères.

Toute classe définit un type ayant le nom de la classe (`Boolean`, `String` sont des classes).

(Plus tard : notion d'interface = liste de services)

Aix*Marseille université
Initiative d'excellence

Ajouter une propriété à une classe

```
public final String model = "Renault Twingo";
private double speed = 60; // in km/h
```

- `public` ou `private` pour indiquer qui a les droits d'accès,
- `final` si la propriété est immutable,
- le type (ici `String`, `double`) de la propriété,
- un identifiant (en `camelCase`),
- $=$ une valeur initiale (optionnel).

Aix*Marseille université
Initiative d'excellence

Accéder à la valeur d'une propriété

```
Boolean isMoving = car.speed > 0;
```

- `objectName.propertyName`
- si la propriété est `private`, seulement depuis cette classe,
- si la propriété est `public`, depuis toute classe,
- l'objet peut être `this`, alors `this.` peut être omis.

Aix*Marseille université
Initiative d'excellence

Modifier la valeur d'une propriété

```
if (this.isAccelerating) {
  this.speed = this.speed + this.acceleration * dt;
}
```

- = est l'opérateur d'assignation,
- son membre gauche est la propriété modifiée,
- son membre droit est la nouvelle valeur de la propriété,
- la nouvelle valeur peut être définie en utilisant l'ancienne valeur.

Avant : `speed = 50`, `acceleration = 10`, `dt = 1`,

Après : `speed = 60`.

Aix*Marseille université
Initiative d'excellence

Règles pour l'assignation

- Seul l'objet propriétaire de la propriété devrait la modifier (c'est sa responsabilité),
- Si un autre objet veut modifier la propriété, il doit passer par une méthode du propriétaire,
- les propriétés marquées `final` sont immutables : on ne peut les modifier.

= n'est pas le symbole mathématique

En Java, l'opérateur d'assignation `=` n'est pas symétrique : `speed = 42` est correct, `42 = speed` est incorrect. Le prédicat d'égalité est `==` (correct : `if (42 == speed)`).

Signification de l'assignation

- Une propriété est un nom, ce nom fait *référence* à un objet.
- Un objet peut avoir de multiples noms.
- Une assignation change ce à quoi se réfère le nom à gauche du symbole =.
- L'assignation ne crée pas un nouvel objet !

Référence

Une propriété est donc une référence sur un objet. Un objet peut être référencé plusieurs fois.

Aix*Marseille université
Initiative d'excellence

Références

```
gasStation.refill(car);
```

- les propriétes sont des références à des objets,
- lorsque l'argument d'une méthode est une propriété (`car`), c'est la référence qui est communiquée, et non l'objet,
- la voiture connue par le nom `car` aura donc son réservoir rempli,
- la référence elle-même n'est jamais modifiée par l'appel de méthode, `car` désigne toujours la même voiture après l'appel à `refill`.

Aix*Marseille université
Initiative d'excellence

Exemple de références multiples

```
People williamHenryMcCarthy = new People("outlaw");
People patGarett = new People("sheriff");
People henryAntrim = williamHenryMcCarthy;
People billyTheKid = williamHenryMcCarthy;
People williamBonney = billyTheKid;

patGarett.kill(billyTheKid);
```

- Qui est mort ?
- Combien de personnes sont mortes ?

Aix*Marseille université
Initiative d'excellence

Litéraux immutables

Identique pour toutes les Renault Twingo :

- `double fuelTankCapacity`,
- `double maxSpeed`,
- `String brand`,...

```
// BAD
public void fillTank() {
  availableFuel = 50;
}
```

Constantes

Solution :

```
// GOOD
private static final double FUEL_TANK_CAPACITY = 50;
public void fillTank() {
  availableFuel = FUEL_TANK_CAPACITY;
}
```

- `static` : partagé par tous les objets de cette classe,
- `final` : pas de modification, personne n'en est responsable.
- 50 n'a pas de sens sans lire le contexte, `FUEL_TANK_CAPACITY` a un sens immédiat.
- éviter les valeurs numériques non-nommées !

Aix*Marseille université
Initiative d'excellence

Variables

Les propriétés sont définies au niveau de l'objet, et vivent tant que l'objet vit.
On peut définir des propriétés au niveau des méthodes, appelées variables, vivant uniquement pendant un appel d'une méthode.

Aix*Marseille université
Initiative d'excellence

Exemple de variables

```
private void updatePosition(double dTime,
                            Vector2D acceleration) {
  Vector2D newVelocity =
    direction.times(speed).plus(acceleration);
  Vector2D dPosition = newVelocity.times(dTime);
  position =
    new Point2D(
      position.getX() + dPosition.x(),
      position.getY() + dPosition.y()
    );
  speed = newVelocity.dot(direction);
  direction =
    speed > 0 ? newVelocity.normalize() :
    speed < 0 ? newVelocity.normalize().times(-1) :
    direction;
}
```

Avantage des variables

- Donner un nom aux calculs intermédiaires : explique le calcul.
- La valeur calculée est utilisable plusieurs fois, mais calculée une seule fois.
- Une variable mutable peut servir à stocker les résultats intermédiaires d'un calcul en plusieurs étapes.

Syntaxe :

```
double availableFuel = 10;
availableFuel = availableFuel + jerrycanContent;
```

Aix*Marseille université
Initiative d'excellence

Instancier une classe

Rappel : classe $\neq$ objet

Une classe définit une façon de construire un objet, qui aura les mêmes comportements que tous les objets de cette classe.

Instanciation

Construire un objet d'une classe s'appelle instancier la classe. L'objet est une instance de cette classe. L'instanciation se fait avec le mot-clé new.

Aix*Marseille université
Initiative d'excellence

Exemples d'instanciation

```
Point2D origin = new Point2D(0,0);
Point2D corner = new Point2D(100,50);
double width = 200;
double height = 100;
Rectangle rectangle =
  new Rectangle(corner, width, height);
Car blueCar = new Car(Color.BLUE);
```

Comme les méthodes, l'instanciation peut prendre des données additionnelles.

```
ClassName objectName = new ClassName(data1,data2);
```

Aix*Marseille université
Initiative d'excellence

Constructeurs

Pour être instanciable, une classe doit possèder au moins un constructeur.

Constructeur

Un constructeur est un morceau d'implémentation d'une classe s'exécutant immédiatement après la création de l'objet. Le constructeur peut utiliser les propriétés et méthodes de l'objet.

(Par défaut les classes possèdent un constructeur ne faisant rien)

Aix*Marseille université
Initiative d'excellence

À quoi sert le constructeur ?

- il initialise les propriétés de l'objet,
- il reçoit des informations utiles à l'initialisation de l'objet,
- il peut appeler des méthodes de l'objet pour mettre l'objet en état de rendre ses services.

```
public Car(Point2D position, Vector2D direction) {
  this.position = position;
  this.direction = direction;
  this.availableFuel = FUEL_TANK_CAPACITY;
  this.startEngine();
}
```

Aix*Marseille université
Initiative d'excellence

Syntaxe

Constructeur :
Comme une méthode, mais sans type de retour, et avec le même nom que la classe !

```
public Car(Point2D position, Vector2D direction) {
  // anything inside
}
```

Instanciation :
Comme un appel de méthode, mais avec `new` devant. Les informations passées doivent correspondre au constructeur !

```
Car myCar =
  new Car(new Point2D(100,0), new Vector2D(0,1));
```

Aix*Marseille université
Initiative d'excellence

Constructeurs et propriétés finales

Souvent le constructeur initialise les propriétés finales de l'objet.

```
public final String name;
private final Date birthday;
private final Boolean isAPerson = true;

public Contact (String name, Date birthday) {
  this.name = name;
  this.birthday = birthday;
}
```

- Soit initialisation dans la déclaration (identique pour tous les objets de la classe),
- Soit initialisation dans le constructeur (selon les informations fournies au constructeur).

Aix*Marseille université
Initiative d'excellence

Quatrième partie IV

Décider : conditionnelles et booléens

Aix*Marseille université
Initiative d'excellence

Prendre des décisions

Toute tâche non-triviale demande de prendre des décisions :

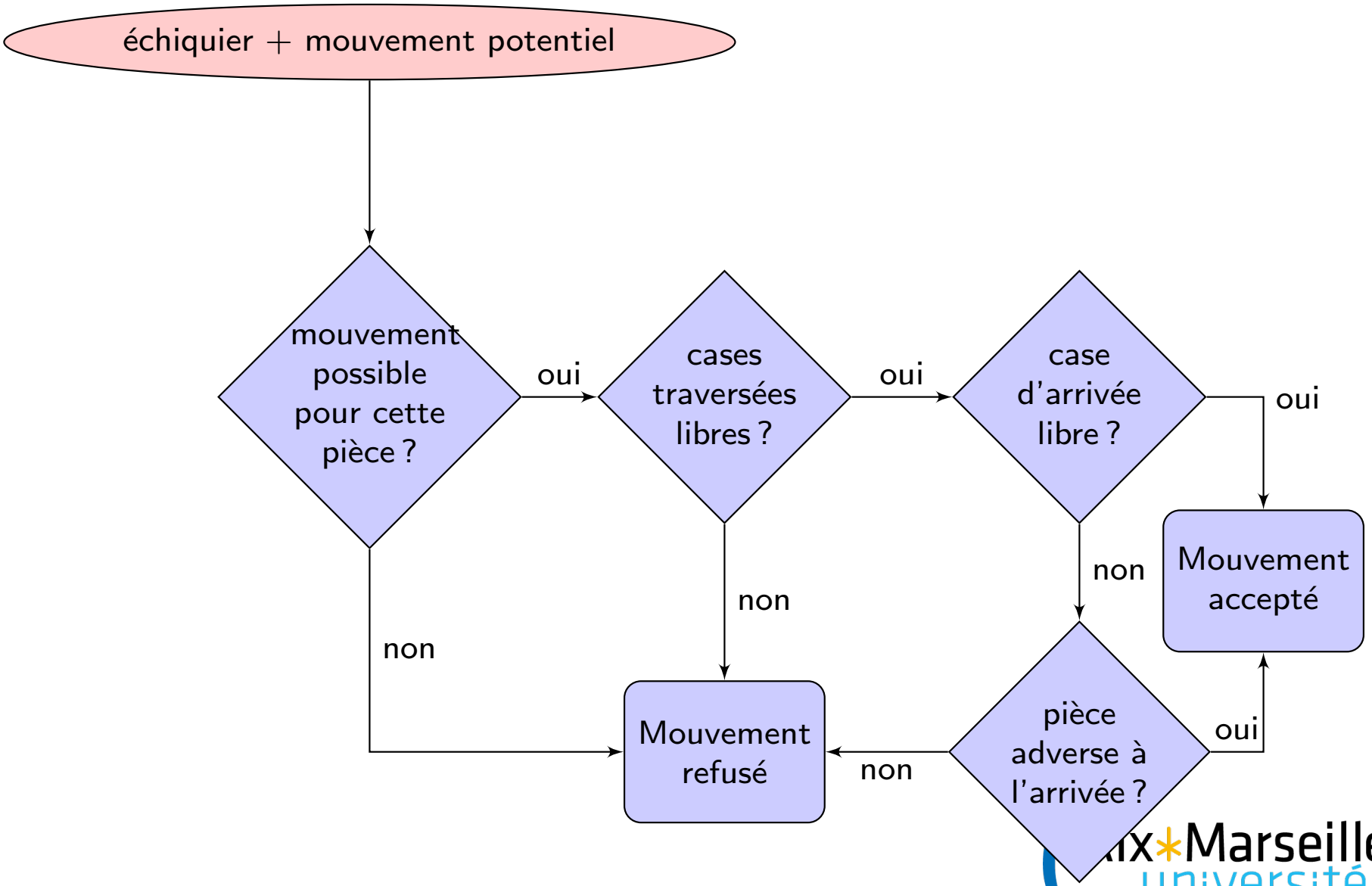
- Dois-je faire comme ceci ?
- Ou dois-je faire comme cela ?

Le choix est pris selon des critères :

- basés sur la situation au moment de la prise de décision,
- souvent exprimables par une réponse “oui” ou “non”,
- ou parfois par un nombre limité de choix (exemple : le jour de la semaine, le parfum d'une glace).

Aix*Marseille université
Initiative d'excellence

Validité d'un déplacement aux échecs



Plus complexe encore !

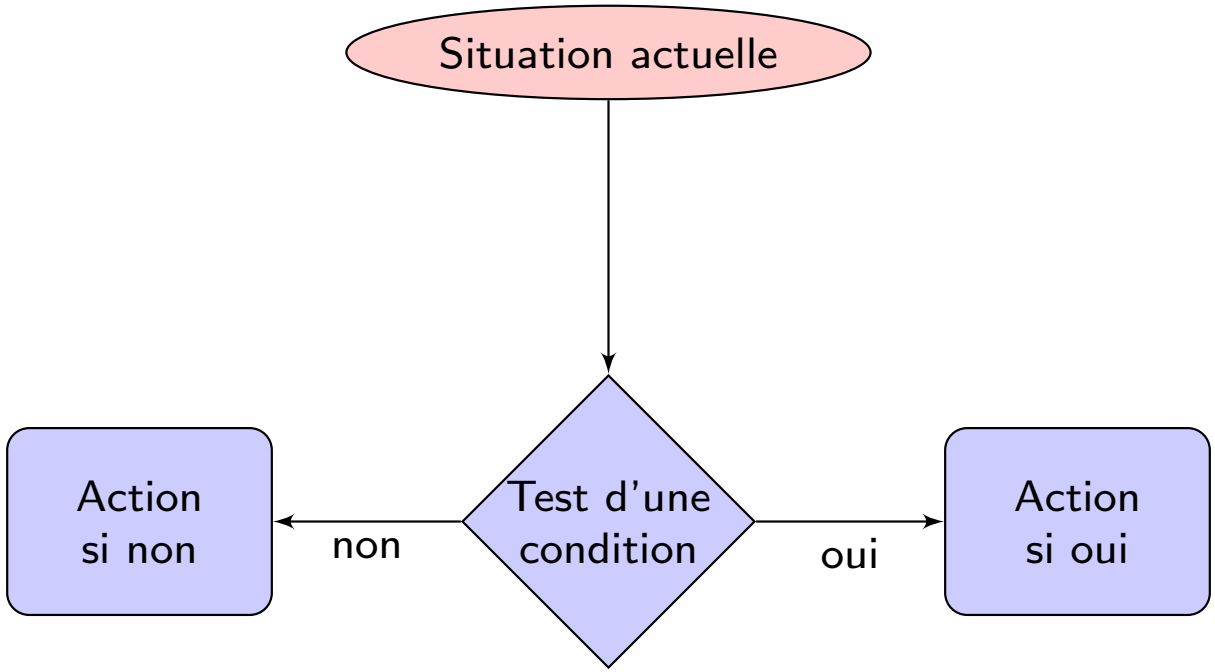
Ce schéma est une simplification :

- mouvements restreints en cas de mise en échec du roi,
- le déplacement et la capture par les pions ne suivent pas les mêmes règles de déplacement,
- règle de la prise en passant pour les pions,
- règle du roque pour les rois.

Bien comprendre et organiser les processus de prise de décision

Aix*Marseille université
Initiative d'excellence

L'instruction conditionnelle



Brique élémentaire : l'instruction conditionnelle.

- Si une condition est vraie, faire une chose,
- si elle est fausse, en faire une autre.

Proposition

Proposition

En mathématiques, une proposition est une phrase qui est soit vraie, soit fausse.

Propositions :

- Il pleut.
- Deux et deux font quatre.
- Deux et deux font cinq.
- Je suis le père Noël.

Pas des propositions :

- x est plus grand que 3.
- Quel jour sommes-nous ?
- Bob, passe-moi le sel.
- Cette phrase est fausse.

Aix*Marseille université
Initiative d'excellence

Prédicat

Prédicat

Un prédicat est une propriété sur un ou plusieurs objets, qui peut être vraie ou fausse selon les objets auxquels il est appliqué.

- est plus grand que 3.
- est bleu.
- peut se mouvoir.

Un prédicat n'a pas de valeur en lui-même ! (il n'est pas vrai, il n'est pas faux)

Aix*Marseille université
Initiative d'excellence

Valeurs booléennes

Booléens

Les valeurs booléennes sont *vrai* et *faux*.

Les valeurs booléennes permettent de représenter la véracité d'une proposition.

- Deux et deux font quatre : vrai.
- Deux et deux font cinq : faux.
- Je suis le père Noël : faux.

Booléens et expressions booléennes

Ces concepts se retrouvent en programmation :

- **valeurs booléennes :** type des booléens `boolean` (ou `Boolean`), valeurs `true` et `false`,
- **prédicats :** méthodes dont le type de retour est `boolean`,
- **propositions :** expressions s'évaluant en un booléen.

```
// from class Piece:
  public boolean mayEnter(Square destination) {
    Piece occupant = board.getPiece(destination);
    return (occupant == null)
           || (occupant.color != this.color);
  }
```

Aix*Marseille université
Initiative d'excellence

Expressions booléennes

Expressions arithmétiques :

```
int x = 2 + 3 * 5;
double y = Math.min(0.7, 3 * Math.cos(Math.PI / 2));
```

Expressions booléennes :

```
boolean b = true && (false || !false);
boolean castlingIsValid =
  kingHasNotMoved && rookHasNotMoved
  && eachInBetweenPositionIsEmpty
  && !someInBetweenPositionIsInCheck;
```

Aix*Marseille université
Initiative d'excellence

Opérateurs booléens

Litéraux :

- `true`, la valeur de vérité positive (vrai),
- `false`, la valeur de vérité négative (faux).

Opérateurs :

- la conjonction : `a && b`, vrai quand `a` est vrai et `b` est vrai.
- la disjonction : `a || b`, vrai quand `a` est vrai ou `b` est vrai.
- la négation : `!b`, vrai quand `b` est faux

Aix*Marseille université
Initiative d'excellence

Sources de booléens

Les booléens proviennent :

- des litéraux `true` et `false`,
- de variables de type `boolean`,
- de prédicats appliqués à leurs arguments (`shoppingList.isEmpty()`, `pawn.mayEnter(square)`,...),
- d'opérateurs de comparaisons :
  - égalité `a == b`,
  - différence `a != b`,
  - plus petit/grand que `a < b`, `a <= b`, `a > b`, `a >= b`.

```
// from class Square:
public boolean isValid() {
  return file >= 1 && file <= 8
         && rank >= 1 && rank <= 8;
}
```

Précédence des opérateurs

En arithmétique, $2 + 3 \times 5 = 2 + (3 \times 5) = 17$. La multiplication est prioritaire sur l'addition.

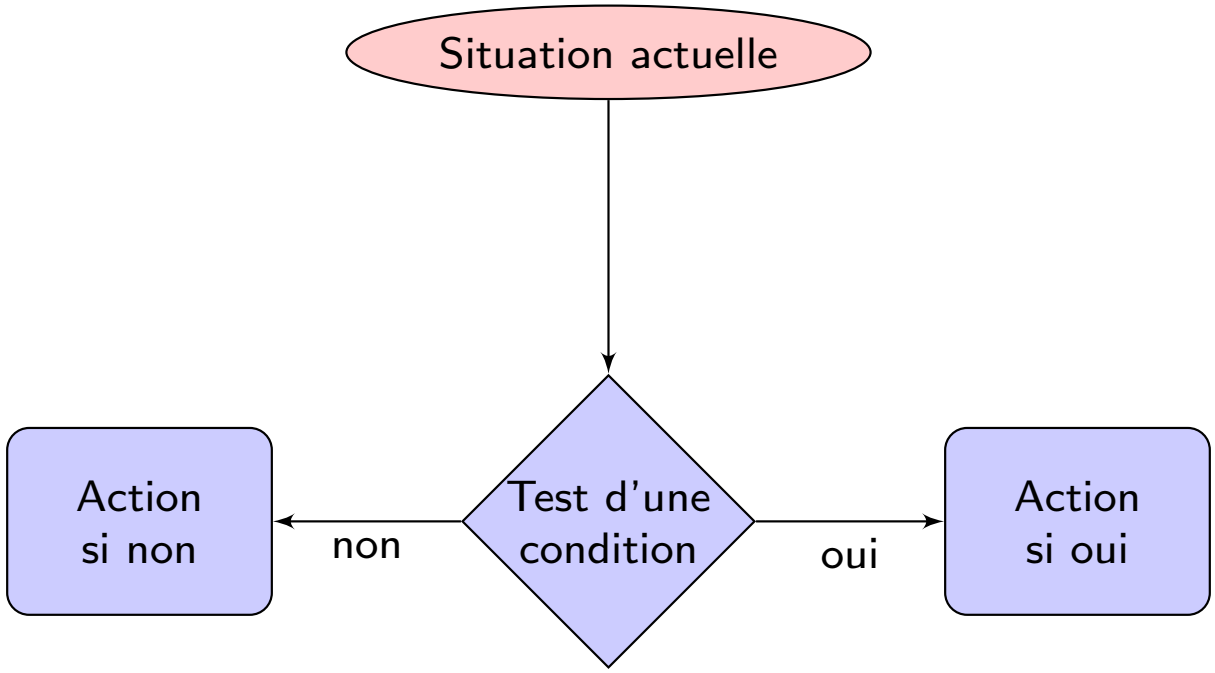
Pour les booléens :

- appels de méthodes (plus prioritaire)
- négation `!`
- comparaisons et égalités,
- conjonction `&&`,
- disjonction `||` (moins prioritaire).

En cas de doute, mettre des parenthèses.

Aix*Marseille université
Initiative d'excellence

Branchement conditionnel




```
if (condition) {
  // things to do when condition is true
} else {
  // things to do when condition is false
}
```

Garde

Parfois, on ne veut rien faire si la condition est fausse :

```
if (condition) {
  // things to do when condition is true
}
```

Un tel bloc n'est exécuté que si la condition est vérifiée.

```
if (iceCream.quantity() < ICECREAM_LOWER_THRESHOLD) {
  shoppingList.add(iceCream);
}
```

Garde et early exit

Un cas particulier de garde sert à quitter une méthode.

```
// from class King:
private void kingCastling(Game game, Square position) {
  if (this.hasMoved) return;
  Piece rook = ...;
    // get the rook initially in king's corner.
  if (rook.hasMoved) return;
  ...
```

Principe : par souci de simplification, quitter la méthode dès que possible (*early exit*).

Branchements multiples

```
// A car at a traffic light
if (signal.color() == Color.RED) {
  stop();
} else if (signal.color() == Color.YELLOW) {
  if (speedIsLow()) {
    stop();
  } else {
    goOn();
  }
} else { // signal.color() == Color.GREEN
  goOn()
}
```

Il n'y a pas de limite technique au nombre de branchements d'une méthode, mais une limite humaine.

Aix*Marseille université
Initiative d'excellence

Garder des branchements simples.

```
// A car at a traffic light
if (signal.color() == Color.RED
    || signal.color() == Color.YELLOW && speedIsLow()
    )
{
  stop();
} else {
  goOn();
}
```

- Identifier les conditions pour chaque comportement,
- les grouper à l'aide d'expressions booléennes.

Simplifier avec des prédicats

```
// A car at a traffic light
if (signal.isRed()
    || signal.isYellow() && speedIsLow()
    )
{
  stop();
} else {
  goOn();
}
```

- utiliser des prédicats permet d'exprimer les intentions
- et d'être proche d'une grammaire naturelle,
- simplifier la logique rend la condition plus compréhensible.

Aix*Marseille université
Initiative d'excellence

Extraction de prédicat

Si une condition est complexe, il est nécessaire de la transformer en prédicat.

```
if (rook != null && !rook.hasMoved
    && !inBetween.anyMatch(board::isOccupied)
    && !traversedByKing.anyMatch(board::isThreatened)
    )
{
  performCastling(rook);
}
```

```
if (canCastle(rook,inBetween,traversedByKing)) {
  performCastling(rook);
}
```

Prédicat extrait

```
private boolean canCastle(
  Piece rook,
  Stream<Square> inBetween,
  Stream<Square> traversedByKing
  )
{
  boolean rookIsStillThere =
    rook != null && !rook.hasMoved;
  boolean noPieceInBetween =
    !inBetween.anyMatch(board::isOccupied);
  boolean somePositionIsInCheck =
    traversedByKing.anyMatch(board::isThreatened);
  return rookIsStillThere
         && noPieceInBetween
         && !somePositionIsInCheck;
}
```

Intérêt de l'extraction de prédicat

Extraire le prédicat oblige à le nommer :

- donc à lui trouver un rôle,
- à expliciter ce rôle,
- à découper la condition en morceaux digestes pour nos cerveaux.

L'IDE extrait automatiquement les prédicats (`Refactor -> Extract -> Method`, Ctr-Alt-M).

Aix*Marseille université
Initiative d'excellence

Règles de nommages

Nommage d'un prédicat, d'un booléen :

- en anglais bien sûr !
- verbe conjugué au présent,
- commence typiquement par "has", "is" (état, propriété) ou "can" (capacité, possibilité) ou "must" (devoir, obligation), ou un verbe exprimant clairement une propriété (et pas une action),
- doit avoir un sens grammaticalement dans la lecture de :

```
if (myObject.isWhatever()) { ... }
if (myObject.canDoWhatever()) { ... }
```

Aix*Marseille université
Initiative d'excellence

Exemples de noms de prédicats

Noms de prédicats :

- isWhite
- isEmpty
- canMoveForward
- contains
- hasMoved

Noms inadaptés pour un prédicat :

- white
- empty
- moveForward
- member

Aix*Marseille université
Initiative d'excellence

Méthodes à conditionnelles (I)

Patron de méthodes séquentielles (séquence d'actions dont certaines sont optionnelles) :

```
public Result doCompositeAction(Arg argument) {
  doFirstAction();
  if (hasConditionFoo()) {
    doAdditionalFirstAction();
  }
  doSecondAction();
  if (hasConditionBar()) {
    doAdditionalSecondAction();
  }
  doThirdAction();
  return computeResult();
```

Méthodes à conditionnelles (II)

Patron de méthodes de répartition (choix d'une action parmi plusieurs selon le contexte) :

```
public Result dispatch(Arg argument) {
  if (hasConditionFoo()) {
    return processFoo();
  }
  if (hasConditionBar()) {
    return processBar();
  }
  if (hasConditionBaz()) {
    return processBaz();
  }
  return processDefaultCase();
```

Aix*Marseille université
Initiative d'excellence

À éviter ! (I)

```
if (condition()) {
  return true;
} else {
  return false;
}
```

Plutôt :

```
return condition();
```

À éviter ! (II)

```
if (condition1()) {
  if (condition2()) {
    if (condition3()) {
      doAction1();
    } else {
      doAction2();
    }
  } else {
    doAction3();
  }
} else {
  doAction4();
}
```

Aix*Marseille université
Initiative d'excellence

À éviter ! (II)

Plutôt :

```
if (!condition1()) {
  doAction4();
  return;
}
if (!condition2()) {
  doAction3();
  return;
}
if (!condition3() {
  doAction2();
  return;
}
doAction1();
```

À éviter ! (III)

```
if (condition()) {
  doThis()
} else {
  doThat();
  andDoThis();
  andDoThat();
  andDoThisToo();
}
```

À éviter ! (III)

Plutôt :

```
if (condition()) {
  doThis();
  return;
}
doThat();
andDoThis();
andDoThat();
andDoThisToo();
```

À éviter ! (IV)

```
if (condition()) {
  result = a;
} else {
  result = b
}
```

Plutôt :

```
result = condition() ? a : b;
```

À éviter ! (IV)

```
public static int min(int a, int b) {
  if (a < b) {
    return a;
  } else {
    return b
  }
}
```

Plutôt :

```
public static int min(int a, int b) {
  return a < b ? a : b;
}
```

Aix*Marseille université
Initiative d'excellence

Conseils généraux

Si la structure de votre méthode est complexe :

- donner des noms aux expressions manipulées,
- simplifier les conditions en prédicats,
- utiliser le *early exit* (quitter la méthode dès que possible),
- penser à utiliser les opérateurs booléens pour grouper les conditions,
- extraire des parties indépendantes en les transformant en méthodes (l'IDE peut vous aider), avec des noms explicites.

Cinquième partie V

Interfaces

Objet, classe : rappel

Un objet

Un objet est une entité composée de propriétés (ou champs), et de comportements (ou méthodes).

Une classe

Une classe (d'objets) définit :

- une façon de créer des objets (constructeur),
- la structure de ces objets (propriétés),
- le comportement de ces objets (méthodes).

Aix*Marseille université
Initiative d'excellence

Mêmes services, différents comportements

Tout :

- pion,
- tour,
- cavalier,
- fou,
- reine,
- roi

peut :

- se déplacer,
- menacer une autre pièce,
- capturer une autre pièce,
- être blanc ou noir.

Les pièces ont les mêmes services (méthodes), mais ne se comportent pas pareil : chaque pièce définit une classe.

Différents comportements, un seul usage

Pour le joueur :

1. choisir n'importe quelle pièce de sa couleur,
2. déplacer la pièce choisie,
3. ou capturer une pièce adverse avec la pièce choisie,
4. s'assurer que son roi n'est pas en échec.

L'usage d'une pièce est indépendant de sa nature précise (tour, pion,...), de sa méthode de déplacement exacte.

On peut séparer la description du mouvement de chaque type de pièce, et la description de l'usage des pièces par un joueur.

Interface

Interface

Une interface est une liste de services que peut rendre un objet. On dit alors que l'objet implémente l'interface.

L'interface `Pièce` :

- se déplacer,
- capturer,
- être blanc ou noir.

Implémentée par : pion, tour, cavalier, fou, reine et roi.

Aix*Marseille université
Initiative d'excellence

Gestion unifiée de plusieurs classes

Sans interface :

```
// from class Player
public void moveKing(King king) { ... }
public void moveQueen(Queen queen) { ... }
public void movePawn(Pawn pawn) { ... }
```

Avec l'interface `Piece` :

```
// from class Player
public void move(Piece piece) { ... }
```

Plus besoin de gérer chaque cas séparément.

objet, classe, interface

Un objet

Un objet est une entité concrète composée de propriétés et de comportements.

Une classe

Une classe (d'objets) définit :

- une façon de créer des objets (constructeur),
- la structure de ces objets (propriétés),
- le comportement de ces objets (méthodes).

Une interface

Une interface est une liste de services pouvant être rendu par les méthodes de divers objets, avec des comportements différents.

Exemple : dessins sur canvas

Nous disposons d'un canvas (une zone sur laquelle dessiner), nous voulons pouvoir dessiner des cercles et des rectangles.

```
// from class Canvas
private List<Circle> circles;
private List<Rectangle> rectangles;

public void paintCircle(Circle circle) {
  pencil.setColor(circle.color);
  pencil.fillCircle(circle.center,circle.radius);
}

public void paintRectangle(Rectangle rect) {
  pencil.setColor(rect.color);
  pencil.fillRectangle(rect.corner,rect.diagonal);
}
```

Questions :

- Que dois-je faire pour pouvoir dessiner des triangles, des pentagones ?
- Que dois-je faire si je veux aussi des cercles bicolores ?
- Comment je vais faire pour gérer la liste des objets que je veux dessiner ?

Aix*Marseille université
Initiative d'excellence

Mauvaise solution

- ajouter des classes `Triangle`, `BicolorCircle`,...
- ajouter des méthodes `paintTriangle`, ...
- ajouter des listes `triangles`, ...

Problèmes :

- `Canvas` ne sera jamais terminé,
- `Canvas` contiendra beaucoup de redondances (combien de listes, de méthodes `PaintFoo` similaires, ...),
- `Canvas` deviendra très long et difficile à appréhender.

Bonne solution (I)

Introduire une interface :

```
interface Paintable {
  public void paint(Pencil pencil);
}
```

```
// from class Canvas
List<Paintable> items;

public void paint(Paintable item) {
  item.paint(pencil);
}

public void paintAll() {
  for (Paintable item : items) {
    paint(item);
  }
}
```

Bonne solution (II)

```
// from class Circle
public class Circle implements Paintable {

   ...

   public void paint(Pencil pencil) {
      pencil.setColor(color);
      pencil.fillCircle(center,radius);
   }
}
```

Aix*Marseille université
Initiative d'excellence

Exemple : écriture de texte

Nous disposons d'un programme générant du texte. Ce texte peut être utilisé de différentes façons :

- affiché à l'écran,
- écrit dans un fichier,
- envoyer vers un serveur,
- envoyer vers une imprimante,
- reformatté, puis utilisé autrement,...

Aix*Marseille université
Initiative d'excellence

Deux écrivains

```java
public class BasicPrinter {
  public void print(String text) {
    System.out.println(text);
  }
}
```

```java
public class QuotedPrinter {
  public void print(String text) {
    System.out.println("\"" + text + "\"");
  }
}
```

Nous voulons facilement pouvoir passer de l'un à l'autre.

Aix*Marseille université
Initiative d'excellence

Changer d'écrivain facilement.

```
Printer printer = new BasicPrinter();
printer.print("Hello world!");
```

```
Printer printer = new QuotedPrinter();
printer.print("Hello world!");
```

Seule différence : le constructeur.

Pour cela, il faut un type `Printer`. Toute interface est aussi un type.

Interface Printer

```
public interface Printer {
  void print(String text);
}
```

```
public class BasicPrinter implements Printer {
    ...
}

public class QuotedPrinter implements Printer {
    ...
}
```

Tout BasicPrinter et tout QuotedPrinter est aussi un Printer.

Aix*Marseille université
Initiative d'excellence

Inversion de dépendances

Avantage de l'interface `Printer` :

- facile de passer d'un écrivain à un autre,
- facile d'intégrer de nouvelles façons d'utiliser le texte,
- le programme générant le texte ne dépend pas de la façon dont le texte est utilisé : il est donc plus facile à maintenir et à étendre (par exemple si on décide de changer la façon dont le texte est affiché).

Les interfaces permettent d'abstraire comment sont réalisés les services, ce qui :

- simplifie l'utilisation de ces services,
- crée une indépendance entre services et clients du service,

Intérêt des interfaces

Les interfaces permettent :

- de définir un nouveau type, sous la forme d'une liste de services,
- d'unifier plusieurs comportements (plusieurs classes) en un seul type,
- d'abstraire les comportements, c'est-à-dire la façon dont les services sont rendus,
- de réduire les dépendances entre classes,
- d'écrire des programmes plus faciles à modifier et étendre.

Aix*Marseille université
Initiative d'excellence

Usage des interfaces

Usage comme type

Dès que possible, le type d'une variable, d'une propriété, d'un paramètre, est une interface.

```
Collection<Piece> pieces = new ArrayList<Piece>();
// Collection<Piece> has a method:
// public void add(Piece piece) { ... }
pieces.add(new King(Color.BLACK));
pieces.add(new Queen(Color.BLACK));
```

Aix*Marseille université
Initiative d'excellence

Interfaces et contrats

Une interface est une liste de service. Un service est une méthode avec :

- un rôle,
- un nom qui reflète ce rôle,
- des paramètres, chacun ayant un type,
- des effets, selon son rôle (lire un fichier, contacter un serveur web, afficher une image,...),
- un résultat, ayant un type.

Le service promet de remplir son rôle. Un interface est donc aussi un contrat.

Aix*Marseille université
Initiative d'excellence

Spécifier le contrat d'une interface

Exemple de la librairie standard

```
    // java.lang.Comparable
public interface Comparable<T> {
  /* @param   o the object to be compared.
   * @return  a negative integer, zero, or a positive
   *          integer as this object is less than, equal
   *          to, or greater than the specified object.
   *
   * @throws NullPointerException if the specified
   *         object is null
   * @throws ClassCastException if the specified
   *         object's type prevents it from being
   *         compared to this object.
   */
  public int compareTo(T o);
}
```

Contrats

Le contrat d'un service est défini en précisant :

- le rôle général du service,
- le rôle de chaque paramêtres,
- l'influence de chaque paramêtre sur les effets produits,
- l'influence de chaque paramêtre sur le résultat produit,
- les erreurs possibles, les limites du service.

Ce qui n'est pas précisé est à la liberté de la classe implémentant l'interface.
Les spécifications peuvent être donnée formellement par des propositions mathématiques.

Aix*Marseille université
Initiative d'excellence

Exemple d'un compteur

On veut utiliser des objets capables de mémoriser un entier positif, avec des opérations pour augmenter ou diminuer de 1 cet entier.

Services :

- consulter la valeur de l'entier,
- incrémenter la valeur de l'entier,
- décrémenter la valeur de l'entier s'il est positif.

Aix*Marseille université
Initiative d'excellence

Interface du compteur

```java
/** An object having an integer property
 * that can be incremented or decremented
 */
public interface Counter {
  /** Access to the integer value.
   *
   * @return the current value of the integer
   */
  int getValue();

  /** Increment the integer value */
  void increment();

  /** Decrement the integer value, with minimum 0 */
  void decrement();
}
```

Spécification formelle (I)

Ces méthodes doivent toujours retourner `true` :

```
public boolean newCounterIsAtZero() {
  Counter counter = new Counter;
  return counter.getValue() == 0;
}
```

```
public boolean valueIsPositive(Counter counter) {
    return counter.getValue() >= 0;
}
```

Aix*Marseille université
Initiative d'excellence

Spécification formelle (II)

```
public boolean incrementAddsOne(Counter counter) {
    int value = counter.getValue();
    counter.increment();
    return counter.getValue() == value + 1;
}
```

```
public boolean decrementRemovesOne(Counter counter) {
    int value = counter.getValue();
    counter.decrement();
    return counter.getValue() == Math.max(0, value - 1);
}
```

Aix*Marseille université
Initiative d'excellence

Implémentation (I)

```
public class EasyCounter implements Counter {
  private int count = 0;

  public int getValue() { return count; }

  public void increment() { count++; }

  public void decrement() {
    if (count > 0) count--;
  }
}
```

Implémentation (II)

```
public class DoubleCounter implements Counter {
  private int incrementCount = 0;
  private int decrementCount = 0;

  public int getValue() {
    return incrementCount - decrementCount;
  }

  public void increment() { incrementCount++; }

  public void decrement() {
    if (getValue() > 0) decrementCount++;
  }
}
```

Aix*Marseille université
Initiative d'excellence

Interfaces usuelles

Quelques interfaces usuelles à connaître
(car très utiles) !

L'interface Comparable (I)

```
public interface Comparable<T> {
  public int compareTo(T other);
}

//typical usage:
public class MyOrderedClass
  implements Comparable<MyOrderedClass> {
  ...
}
```

Sert à définir une relation d'ordre entre les objets d'une classe (par exemple, pour pouvoir les trier).

L'interface Comparable (II)

```
public class People {
  public final String name;
  ...
  public int CompareTo(People other) {
    return name.compareTo(other.name);
  }
}

Collection<People> peoples;
...
Collections.sort(peoples); // sort by name
```

L'interface `Iterable` (I)

```
public interface Iterable<T> {
    Iterator<T> iterator();
    void forEach(Consumer<? super T> action);
    Spliterator<T> spliterator();
}
```

- définition très complexe,
- mais usage très utile,
- on ne définira pas de classes implémentant `Iterable` lors de ce cours (trop compliqué, niveau L3)

L'interface `Iterable` (II)

Usage : si une classe implémente `Iterable<Thing>`, c'est que ses objets contiennent une collection de `Thing`, et qu'on peut y accéder avec un `for`.

```
public void printAllPieces(Iterable<Piece> pieces) {
  for (Piece piece : pieces) {
    System.out.println(piece.toString());
  }
}
```

(refera l'objet d'un cours)

L'interface Collection

```
public interface Collection<T> extends Iterable<T> {
  boolean add(T element);
  boolean contains(T element);
  boolean isEmpty();
  boolean remove(Object o);

  ...
}
```

Sert à stocker plusieurs objets de même interface `T`.

Implémentations : `ArrayList`, `Vector`, `HashSet`, ...

L'interface `Set`

Comme `Collection`, mais spécification plus forte :

Ne contient pas deux fois le même élément

Implémentations : `HashSet`, `TreeSet`

Aix*Marseille université
Initiative d'excellence

D'autres collections

- `Set` : comme en mathématiques,
- `List` : éléments en séquence (avec un début, un milieu, une fin),
- `Deque` : séquence, mais accès uniquement par les extrémités (comme une file d'attente en magasin).
- ...

Plusieurs interfaces qui raffinent la spécification d'une collection.

Aix*Marseille université
Initiative d'excellence

Récapitulatif

- Une interface est une liste de service.
- Une interface définit un type, pouvant généraliser plusieurs classes.
- Une interface possède une spécification devant être respectée.
- Une interface permet de réduire les dépendances, d'abstraire les services utiles (ce qui rend le programme plus simple).

```
public class MyClass implements MyInterface {
   ...
}
```

Sixième partie VI

Collections et Générique

Avertissement

- Les programmes ont chacun une finalité précise (ou plusieurs).
- Ils sont réalisés par des machines extrêmement fiables et stupides.
- Ces machines ne comprennent pas l'objectif des programmes. Elles font ce qu'on leur demande, ni plus ni moins.

Avertissement

C'est donc au programmeur

- d'être précis dans ce qu'il demande à la machine,
- de bien prendre en compte les cas extrêmes ou exceptionnels, et les difficultés particulières de la tâche à accomplir.
- de s'assurer que son programme réalise les objectifs fixés.

Aucun programmeur n'en est naturellement capable !

Aix*Marseille université
Initiative d'excellence

Comment éviter les problèmes ?

Outils pour s'assurer qu'un programme est correct :

- simplifier le programme !
- travailler en équipe,
- pratiquer la relecture de programme,
- faire des tests,
- utiliser des outils d'analyse statique,
- spécifier et prouver formellement le programme (très complexe et donc très cher, donc réservé aux systèmes critiques).

Aix*Marseille université
Initiative d'excellence

Garder le programme simple

- Utiliser une approche modulaire de programmation (par exemple la programmation objet),
- Utiliser un langage adapté à la tâche à accomplir (si possible),
- Garder à l'esprit les règles de base :
  - DOT : *Do One Thing*
  - DRY : *Don't Repeat Yourself*
  - KISS : *Keep It Simple Stupid !*
- Utiliser une méthodologie de développement (incrémental par exemple),
- Avoir une stratégie de surveillance de la qualité du programme.

Aix*Marseille université
Initiative d'excellence

Spécification

Lors de la conception du programme, tous les comportements doivent être spécifiés.

- quel est le comportement normal de ma classe, de ma méthode ?
- quel sont les arguments autorisés de chaque méthode ?
- existe-t-il des cas exceptionnels, des cas extrêmes, et que se passe-t-il dans ce cas ?
- Y-a-t-il des contre-indications à l'usage de la classe ?
- Quel lien y-a-t-il entre la valeur retournée par la méthode et les arguments fournis ? Puis-je le spécifier mathématiquement, par une assertion logique ?

Comment spécifier ?

- spécification semi-formelle, texte d'explication,
- spécification formelle par des assertions logiques,
- exemples d'usage, avec traitement exhaustif des cas particuliers. Ces exemples peuvent être employés pour tester le programme.

Spécification et documentation

La spécification fournira une documentation pour aider les programmeurs à utiliser la classe.
Utilisation de `javadoc` :

```
/** computes an average of two colors.
 * @param color the color to mix this with.
 * @param t between 0 and 1
 * @return t * this + (1-t) * color
 */
public Color weightedAverage(Color color, double t) {
 ...
}
```

(`Alt+Enter` sur le nom de la méthode pour générer le commentaire javadoc)

Aix*Marseille université
Initiative d'excellence

Spécification et tests

La spécification sous forme d'exemples de comportement ou d'assertions mathématiques mènent naturellement à l'écriture de tests.

Aix*Marseille université
Initiative d'excellence

Pourquoi tester ?

- Vérifier que les spécifications sont respectées.
- Déceler un maximum d'erreurs.
- Déceler les erreurs le plus tôt possible.
- Vérifier que le programme assure son travail en un temps raisonnable.
- S'assurer que les modifications futures ne cassent pas les fonctionnalités du programme (test de non-régression).

Aix*Marseille université
Initiative d'excellence

Comment tester ?

- Le plus tôt possible ! Idéalement les tests sont écrits avant le programme (*cf. Test-Driven Development*).
- Si possible par un programmeur différent de celui qui écrit le programme. Certaines entreprises ont des employés spécialisés dans les tests.
- Exhaustivement. Tester tous les cas. Tester toutes les lignes de code (*couverture de code*).

Aix*Marseille université
Initiative d'excellence

Catégories de tests

- tests unitaires : test d'une seule fonctionnalité, d'un seul comportement simple du programme.
- tests d'intégration : test que deux composants interagissent comme prévu ensemble.
- tests *black-box* : tests écrits en se basant uniquement sur la spécification du programme, sans lire le programme, pour tester le comportement observable du programme.
- tests *white-box* : test écrits en se basant sur le programme, pour tester un traitement particulier, le comportement interne du programme.

Aix*Marseille université
Initiative d'excellence

Tests unitaires

- Test d'un seul comportement, d'une seule méthode avec un choix d'argument précis.
- permet de cibler précisément la source d'une erreur.
- détecte la grande majorité des erreurs, souvent facile à corriger.
- facile à mettre en place. Framework dédiés (Junit pour Java). Facile à automatiser.

Aix*Marseille université
Initiative d'excellence

Utilisation de Junit

Java possède des frameworks pour écrire des tests, dont JUnit.

```
@Test
public void scaleTest() {
  assertEquals(
    new Color(1,1,1),
    new Color(0.5,0.5,0.5).scale(2)
  );
}
```

méthodes de Junit

Assertions :

- `assertEquals`
- `assertTrue`
- `assertThat`
- ...

Hypothèses :

- `assumeTrue`
- `assumeThat(T value, Matcher<T> condition)`

Aix*Marseille université
Initiative d'excellence

Annotations pour `Junit`

- `@Test` marque un test,
- `@DisplayName(‘‘myName’’)` donne un nom au test,
- `@RepeatedTest(someInt)` combien de fois réaliser le test,
- `@ParameterizedTest` pour un test prenant un paramètre,
- `@ValueSource(ints = { 1,2,5,10,100 })` pour préciser les valeurs des arguments d'un test paramètrés (*cf.* la documentation).

Aix*Marseille université
Initiative d'excellence

Huitième partie VIII

Interface

Objets, classes, interfaces

- Les objets ont des propriétés et des comportements.
- Les classes regroupent des objets ayant :
  - des propriétés de même nature, mais pas de même valeur,
  - des comportements de même nature et de même valeur.
- les interfaces regroupent des objets ayant :
  - des comportements de même nature, mais différents.

Aix*Marseille université
Initiative d'excellence

Exemple des recyclables (I)

Imaginons les classes :

- Papier,
- Canette,
- Bouteille,
- Carton

Toutes ont des comportements différents, s'utilisent dans des contextes différents.
Mais aussi toutes sont recyclables et vont donc dans la poubelle des objets recyclables.

Exemple des recyclables (II)

Papier, Canette, Bouteille, Carton

- possèdent une méthode `recycle` (qui n'ont pas le même comportement, on ne recycle pas le verre comme le papier),
- possèdent aussi d'autres méthodes propres à chaque classe.

La Poubelle peut accepter n'importe quel objet recyclable.

Exemple des recyclables (II)

```
// class RecycleBin
public void empty() {
  for (int i = 0; i < firstEmptyCell; i++) {
     trash[i].recycle();
  }
  firstEmptyCell = 0;
}
```

Quel type pour `trash` ?

Aix*Marseille université
Initiative d'excellence

Tableau en Java

Les tableaux en Java sont homogènes : tous les éléments contenus dans le tableau sont du même type.

- parce que c'est plus simple !
- parce que l'expérience montre que c'est ce qu'on veut quasiment toujours,
- parce que sinon on ne sait pas comment utiliser un élément du tableau (on ne sait pas sa nature).

Donc `trash` doit être un tableau d'un type bien précis. Lequel ?

Aix*Marseille université
Initiative d'excellence

Exemple des cours (I)

Exemple de l'algorithme d'affectation d'étudiants sur les cours.
Chaque cours propose :

- `register(student)` inscrire un étudiant,
- `getListing()` fournir la liste des étudiants inscrits.

Pour l'inscription :

- **Informatique :** sélection par tirage au sort,
- **Mécanique :** sélection par ordre d'arrivée

Aix*Marseille université
Initiative d'excellence

Exemple des cours (II)

- Les cours de mécanique et d'informatique propose un même service (s'inscrire), mais ne le traite pas de la même façon.
- l'inscription en informatique et en mécanique correspondent donc à deux fragments de programmes différents (donc deux classes).
- Une fois le comportement de l'inscription défini, les deux cours peuvent s'utilisent de la même manière (pour les étudiants, on s'inscrit de la même façon). Les deux sont interchangeables.

Les interfaces sont des types !

Aix*Marseille université
Initiative d'excellence

Définition d'interface

Une interface

Une *interface* est la description d'une liste de services, sans préciser comment le service est accompli (sans implémentation).

- La définition d'interface ne contient pas d'instructions, seulement des déclarations.
- Une interface peut déclarer les en-têtes d'une ou plusieurs méthodes.
- Les propriétés n'apparaissent pas dans les interfaces.

Aix*Marseille université
Initiative d'excellence

Implémentation d'une interface

Implémenter une interface

Une classe (ou un objet) *implémente* une interface si toute méthode déclarée dans l'interface est déclarée et programmée (implémentée) dans la classe.

- Pour implémenter une méthode d'une interface, la méthode de la classe doit avoir le même nom, les mêmes paramètres, le même type de retour.
- Une classe peut implémenter des méthodes en plus par rapport à l'interface.
- Une classe peut implémenter plusieurs interfaces différentes.

Aix*Marseille université
Initiative d'excellence

Solution pour le recyclage

```
public interface Recyclable {
  void recycle();
}

public class Bottle implements Recyclable {
  ...
  public void recycle() { ... }
}
```

`trash` est de type `Recyclable[]`.

Neuvième partie IX

Collections

Aix*Marseille université
Initiative d'excellence

Collections

Pourquoi des collections d'objets ?

- pour manipuler des ensembles d'objets simultanément,
- pour ranger et accéder à des objets selon les besoins,

Pourquoi plusieurs interfaces de collections ?

- selon l'utilisation qu'on veut faire des objets stockés.

Aix*Marseille université
Initiative d'excellence

Exemples de collection

Exemples de collections dans la vie courante.

- Réfrigérateur : collection d'aliments mis au frais.
- File d'attente au guichet : collection de personnes attendant dans l'ordre d'arrivée.
- Pile de T-shirts : collection de vêtements avec un ordre spatial.
- Annuaire : collection de noms et de numéros de téléphone.

Réfrigérateur

Exemples d'opérations :

- prendre une bière,
- mettre une bière,
- compter le nombre de bières,
- tester s'il y a une bière.

Interface `Set`

L'interface Set

```
public interface Set<E> {
    boolean add(E e);
    boolean contains(E e);
    boolean isEmpty();
    boolean remove(E e);
    ...
}
```

Implémentations : TreeSet, HashSet

File d'attente

Exemples d'opérations :

- ajouter une personne,
- récupérer la personne suivante,
- tester si la file est vide,
- compter le nombre de personnes en attente.

Interface `Queue`

L'interface Queue

```
public interface Queue<E> {
  boolean offer(E e);
  E poll();
  E peek();
  boolean isEmpty();
  int size();
  ...
}
```

Implémentations : `ArrayDeque`, `LinkedList`, `PriorityQueue`

Pile de T-shirts

Exemples d'opérations :

- mettre in T-shirt en haut de la pile,
- récupérer le T-shirt du haut de la pile,
- tester s'il y a un T-shirt.

Interface : `Deque`, qui combine file et pile.

Aix*Marseille université
Initiative d'excellence

L'interface Deque

```
public interface Deque<E> {
  boolean offerFirst(E e);
  E pollFirst();
  E peekLast();
  boolean offerLast(E e);
  E pollLast();
  E peekLast();
  ...
}
```

Implémentations : `ArrayDeque`, `LinkedList`

Aix*Marseille université
Initiative d'excellence

Annuaire

Exemples d'opérations :

- ajouter un contact (nom + numéro),
- chercher le numéro d'un contact,
- retirer un contact.

Interface `Map`

L'interface Map

```
public interface Map<Key,Value> {
  Value put(Key k, Value v);
  boolean containsKey(Key k);
  Value get(Key k);
  Value remove(Key k);
  Set<Key> keySet();
  Collection<Value> values();
  ...
}
```

Implémentations : HashMap, TreeMap

Aix*Marseille université
Initiative d'excellence

L'interface `List`

Séquence ordonnée indexable par des entiers.

```
public interface List<E> {
  boolean add(E e);
  E get(int index);
  int indexOf(E e);
  ...
}
```

Implémentations : `Vector`, `LinkedList`, `ArrayList`

Aix*Marseille université
Initiative d'excellence

L'interface `Collection`

`Set`, `Queue`, `Deque`, `List` *étendent* l'interface `Collection`

```
public interface Set<E> extends Collection<E> {

}
```

Autrement dit, toutes ces interfaces possèdent aussi les services définies par l'interface `Collection`.

L'interface Collection

```
public interface Collection<E> {
  boolean add(E e);
  boolean remove (E e);
  int size();
  Iterator<E> iterator();
  ...
}
```

Définition d'une collection dans votre programme :

```
Collection<Item> myItems = new ArrayList<Item>();
```

Aix*Marseille université
Initiative d'excellence

Trier une collection

La classe `Collections` contient des fonctions (méthodes statiques) pour travailler sur les collections. Exemple :

```
List<Item> myList = ...;
Collections.sort(myList);
```

Il faut que la classe `Item` définisse une méthode `compareTo`, c'est-à-dire qu'elle implémente l'interface `Comparable<Item>`.

Aix*Marseille université
Initiative d'excellence

Itérer sur une collection

Comment faire une manipulation pour chaque item d'une collection ?

Facile !

```
Collection<Item> items = ...;
for (Item item : items) {
  ...
}
```

Comme un `for` classique sinon : possibilité de `break`, `return` et de faire toute sorte de calcul. Ordre des éléments : dépend de l'implémentation de la collection.

Aix*Marseille université
Initiative d'excellence

Dixième partie X

Généricité

Exemple 1 (I)

Écrire toutes les valeurs d'une collection d'entiers :

```
public String write(int[] ints) {
  boolean isFirst = true;
  StringBuilder builder = new StringBuilder();
  for (int i : ints) {
    if (isFirst) { isFirst = false; }
    else { builder.append(", "); }
    builder.append(i);
  }
  return builder.toString();
}
```

Exemple 1 (II)

Écrire toutes les valeurs d'une collection de chaînes :

```
public String write(String[] strings) {
  boolean isFirst = true;
  StringBuilder builder = new StringBuilder();
  for (String str : strings) {
    if (isFirst) { isFirst = false; }
    else { builder.append(", "); }
    builder.append(str);
  }
  return builder.toString();
}
```

Analyse de l'exemple 1

- Les deux fragments sont quasiment identiques,
- la seule différence concerne le type des éléments manipulés,
- la nature précise des éléments manipulés n'est pas utile : ce fragment fonctionnerait pour n'importe quel type,
- si on a 20 types différents, besoin d'écrire 20 fois la méthode ?

Besoin d'un mécanisme pour l'écrire une seule fois !

Aix*Marseille université
Initiative d'excellence

Abstraire le type

```
public <T> String write(T[] things) {
  boolean isFirst = true;
  StringBuilder builder = new StringBuilder();
  for (T thing : things) {
    if (isFirst) { isFirst = false; }
    else { builder.append(", "); }
    builder.append(thing);
  }
  return builder.toString();
}
```

T est une abstraction de type : même rôle qu'une variable en mathématique, représente une valeur (ici un type) non-précisée.

Aix*Marseille université
Initiative d'excellence

Utilisation de la méthode

```
int[] ints = { 1, 2, 5, 10, 100 };
System.out.println(write(ints));
  // output: 1, 2, 5, 10, 100
string[] strings { "Hello", "", "World", "!" };
System.out.println(write(strings));
  // output: Hello, , World, !
```

Utilisation transparente.

Syntaxe des méthodes génériques

```
public <T> String write(T[] things) {
   ...
}
```

- méthode avec des variables de types : *méthode générique*,
- `T` : *paramètre de généricité*, variable de type,
- déclaration de la variable de type, juste avant le type de retour de la méthode.
- plusieurs variables de type : `<S,T,U>`,
- le reste garde la syntaxe usuelle.

Aix*Marseille université
Initiative d'excellence

Variables de types

- `<T>` déclare une *variable de type*,
- les variables usuelles abstraient des valeurs, on leur substitue la valeur qui leur est donnée *à l'usage*,
- les variables de types abstraient des types, on leur substitue le type adéquat *à l'usage*.
- Différents usages : différentes valeurs, différents types.

Exemple 2 (I)

```
public int sum(int[] ints) {
  int sum = 0;
  for (int i : ints) {
    sum = sum + i;
  }
  return sum;
}
```

Exemple 2 (II)

```
public double product(double[] doubles) {
  double product = 1;
  for (double d : doubles) {
    product = product * d;
  }
  return product;
}
```

Aix*Marseille université
Initiative d'excellence

Analyse de l'exemple 2

```
public <T> T myMethod(T[] values) {
  T result = ___;
  for (T value : values) {
    result = result ___ value;
  }
  return result;
}
```

- fragments très similaires,
- mais les valeurs ne sont pas utilisées de la même façon : pas le même comportement,
- points communs : utilisation d'une valeur initiale, utilisation d'un opérateur binaire.
- services identiques, comportements différents : interface !

Aix*Marseille université
Initiative d'excellence

Interface générique

```
public interface Monoid<T> {
  T neutral();
  T operator(T left, T right);
}

public class IntWithAddition
  implements Monoid<Integer> {

  Integer neutral() { return 0; }
  Integer operator(Integer left, Integer right) {
   return left + right;
  }
}
```

Aix*Marseille université
Initiative d'excellence

Utilisation de l'interface générique

```
public <T> T reduce(T[] values, Monoid<T> monoid) {
  T result = monoid.neutral();
  for (T value : values) {
    result = monoid.operator(result, value);
  }
  return result;
}


int[] ints = { 1, 2, 5, 10, 100};
int sum = reduce(ints, new IntWithAddition());
```

Utilisation de l'interface générique

Les collections sont des interfaces génériques aussi :

```
public <T> T reduce(Collection<T> values,
                    Monoid<T> monoid) {
  T result = monoid.neutral();
  for (T value : values) {
    result = monoid.operator(result, value);
  }
  return result;
}

Collection<Integer> ints =
  Arrays.asList(1, 2, 5, 10, 100);
int sum = reduce(ints, new IntWithAddition());
```

Types génériques, types paramétrés

Type générique

Un *type générique* est un type admettant des paramètres formels de types.

Exemples : `ArrayList<E>`, `Group<T>`, `Iterable<T>`

Type paramétré

Un *type paramétré* est l'*instanciation* d'un type générique avec des paramètres de types concrets.

Exemples : `ArrayList<Integer>`, `Group<String>`, `Iterable<Double>`.

Aix*Marseille université
Initiative d'excellence

Exemple : Itérable

De nombreux objets sont itérables :

- les collections,
- les mots d'un texte,
- une séquence définie par une relation de récurrence $u_{n+1} = f(u_n)$,
- les pixels d'une image,
- ...

Besoin d'un mécanisme pour unifier le traitement.

Boucle `for` des itérables

```
Iterable<Pixel> image = ...;
for (Pixel pix : image) {
  doSomething(pix);
}
```

- Syntaxe :
  `for (EltType elt : providingObject) { ... }`,
- `providingObject` doit implémenter `Iterable<EltType>`,
- `break`, `continue` et `return` autorisés.

Aix*Marseille université
Initiative d'excellence

L'interface `Iterable`

```
public interface Iterable<E> {
  Iterator<E> iterator();
}
```

- Être itérable, c'est avoir la capacité de créer un objet chargé de l'itération (`Iterator`).
- Un objet peut être itérable sur plusieurs types différents (itérer les caractères ou les mots d'un texte).

Aix*Marseille université
Initiative d'excellence

L'interface `Iterator`

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

- `hasNext` permet de savoir s'il existe au moins un élément supplémentaire,
- `next` permet de récupérer le prochain élément.
- l'`iterator` est l'objet responsable de fournir les éléments un par un.

Aix*Marseille université
Initiative d'excellence

`Iterable` **vs** `Iterator`

- `Iterable` possède des éléments qui peuvent être examiné un par un (peut être itéré).
- `Iterator` fournit des éléments un par un (itère).

Généralement :

- Suffixe `able` : indique une capacité.
- Suffixe `ator` : indique un rôle.

Itérateur d'entiers

```
public class Range implements Iterator<Integer> {
  private int next = 1;
  private final int max;
  public Range(int min, int max) {
    this.next = min;
    this.max = max;
  }
  public boolean hasNext() { return next <= max; }
  public Integer next() {
    if (!hasNext()) throw new NoSuchElementException();
    return next++;
  }
}
```

Itérateurs d'entiers (utilisation, I)

```
Iterator<Integer> range = new Range(1,10);
range.hasNext(); // true
range.next(); // 1
range.next(); // 2
range.hasNext(); // true
range.next(); // 3
for (int i = 4; i < 10; i++) { range.next(); }
range.next(); // 10
range.hasNext(); // false
```

Aix*Marseille université
Initiative d'excellence

Itérateurs d'entiers (utilisation, II)

```
Iterator<Integer> range = new Range(1,10);
int sum = 0;
while (range.hasNext()) {
  int next = range.next();
  sum = sum + next;
  System.out.println(next);
}
```

Le fragment ci-dessous est faux (Range n'est pas itérable) !

```
for (Integer i : new Range(1,10);) { // NON !
  System.out.println(i);
}
```

Aix*Marseille université
Initiative d'excellence

Intervalle itérable (I)

```
public class Interval implements Iterable<Integer> {
  public final int start;
  public final int stop;
  public interval(int start, int stop) {
    this.start = start;
    this.stop = stop;
  }
  Iterator<Integer> iterator() {
    return new Range(start,stop);
  }
}
```

Intervalle itérable (II)

Utilisation :

```
Interval interval = new Interval(1,10);
int sum = 0;
for (Integer i : interval) {
  sum = sum + i;
  System.out.println(i);
}
```

Raccourci pour :

```
Iterator<Integer> iterator = interval.iterator;
while (iterator.hasNext()) {
  int i = iterator.next();
  sum = sum + i;
  System.out.println(i);
}
```

Suite (I)

```
public class Sequence implements Iterable<Double> {
  private final DoubleUnaryOperator f;
  private final Double initialValue;
  public Sequence(DoubleUnaryOperator f, Double x0) {
    this.f = f; this.initialValue = x0;
  }
  public Double term(int i) {
    for (Double d : this) {
      if (i == 0) return d;
      i--;
    }
    return 0;
  }
  Iterator<Double> iterator() {
     return SequenceIterator(f,initialValue);
  }
}
```

Suite (II)

```
public class SequenceIterator
implements Iterator<Double> {
  private final DoubleUnaryOperator f;
  private Double nextValue;
  public SequenceIterator(DoubleUnaryOperator f,
                          Double x0) {
    this.f = f; this.nextValue = x0;
  }
  public boolean hasNext() { return true; }
  public Double next() {
    Double current = nextValue;
    nextValue = f.applyAsDouble(nextValue);
    return current;
  }
}
```

Suite (III)

Utilisation :

```
public double sq(double x) { return x * x; }

public double sqrt(double target) {
  Sequence newtonRoot =
   new Sequence(x -> 0.5 * (x + target / x),
                target);
  for (Double possibleRoot : newtonRoot) {
    if (Math.abs(sq(possibleRoot) - target) < epsilon)
      return possibleRoot
  }
}

sqrt(139328.5387); // 373.26738231460837
Sequence powersOfTwo = new Sequence(x -> 2. * x, 1);
powersOfTwo.term(16); // 65536.0
```

Pourquoi séparer `Iterable` et `Iterator`

```
Interval range = new Interval(1,10);
for (Integer i : range) {
  for (Integer j : range) {
    context.fillEllipse(10. * i, 10. * j, 8, 8);
  }
}
```

- L'objet `range` possède un seul état pour gérer les deux itérations,
- Les deux boucles `for` interfèrent !
- les `Iterator` permettent de gérer l'état de l'itération.

Aix*Marseille université
Initiative d'excellence

L'interface Comparable

```
public interface Comparable<T> {
  int compareTo(T value);
}
```

- compareTo retourne une entier :
  - strictement positif si this > value,
  - strictement négatif si this < value,
  - nul si this = value,
- doit avoir un comportement cohérent avec equals,
- doit avoir un comportement cohérent avec equals,
- utile pour TreeMap, TreeSet, Collections.sort.

L'interface Comparator<T>

```
public interface Comparator<T> {
  int compare(T value1, T value2);
}
```

- compareTo retourne une entier :
  - strictement positif si value1 > value2,
  - strictement négatif si value1 < value2,
  - nul si value1 = value2,
- doit avoir un comportement cohérent avec equals,
- utile pour avoir des ordres autres que celui défini via Comparable.

Implémentation d'une pile d'entiers (I)

```
public class IntStack implements Iterable<Integer> {
  Integer[] values = new Integer[1000];
  Integer firstEmptyCell = 0;
  boolean isEmpty() { return firstEmptyCell == 0; }
  public void push(Integer value) {
    values[firstEmptyCell++] = value;
  }
  public Integer peek() {
    if (isEmpty()) throw new NoSuchElementException();
    return values[firstEmptyCell-1];
  }
  public Integer poll() {
    if(isEmpty()) throw new NoSuchElementException();
    return values[--firstEmptyCell];
  }
  ...
}
```

Implémentation d'une pile d'entiers (II)

```
public class IntStack implements Iterable<Integer> {
  ...
  public Iterator<Integer> iterator() {
    return new StackIterator(values, firstEmptyCell-1);
  }
  private class StackIterator
      implements Iterator<Integer> {
    private final Integer[] values;
    private Integer current = 0;
    private final Integer last;
    public StackIterator(...) { ... }
    public boolean hasNext() { return current <= last; }
    public Integer next() {
      if (!hasNext())
        throw new NoSuchElementException();
      return values[next++];
    }
  }
}
```

Implémentation d'une pile de `String` ?

- Même chose en substituant `String` à `Integer`,
- DRY : besoin d'un mécanisme pour ne pas dupliquer le programme,
- Généricité de classe : paramètrage de la classe par un type générique.

Notre classe `Stack` n'utilisait pas les méthodes des valeurs mises dans la pile : la restriction au type `Integer` n'est pas nécessaire.

Aix*Marseille université
Initiative d'excellence

Implémentation d'une pile générique (I)

```
public class Stack<T> implements Iterable<T> {
  T[] values = new T[1000];
  T firstEmptyCell = 0;
  boolean isEmpty() { return firstEmptyCell == 0; }
  public void push(T value) {
    values[firstEmptyCell++] = value;
  }
  public T peek() {
    if (isEmpty()) throw new NoSuchElementException();
    return values[firstEmptyCell-1];
  }
  public T poll() {
    if(isEmpty()) throw new NoSuchElementException();
    return values[--firstEmptyCell];
  }
  ...
}
```

Implémentation d'une pile générique (II)

```
public class Stack<T> implements Iterable<T> {
  ...
  public Iterator<T> iterator() {
    return new StackIterator(values, firstEmptyCell-1);
  }
  private class StackIterator implements Iterator<T> {
    private final T[] values;
    private T current = 0;
    private final T last;
    public StackIterator(...) { ... }
    public boolean hasNext() { return current <= last; }
    public T next() {
      if (!hasNext())
        throw new NoSuchElementException();
      return values[next++];
    }
  }
}
```

Classe générique

Déclaration d'une classe générique :

```
public class MyGenericClass<T> { ... }
```

Instanciation d'une classe générique :

```
MyGenericClass<Integer> obj =
  new MyGenericClass<Integer>(...);
// or when there is no ambiguity:
MyGenericClass<Integer> obj =
  new MyGenericClass<>(...);
```

Tout le reste identique (méthodes, déclaration du constructeur,...).

Aix*Marseille université
Initiative d'excellence

Classe Greatest

```
public class Greatest<E> {
  private E max;
  public Greatest(E initialMax) {
    this.max = initialMax;
  }
  public void add(E elt) {
    if (elt.compareTo(max) > 0) { max = elt; }
  }
  public E getMax() {
    return max;
  }
}
```

Il faut que E implémente Comparable<E>.

Aix*Marseille université
Initiative d'excellence

Classe Greatest

```
public class Greatest<E extends Comparable<E>> {
  private E max;
  public Greatest(E initialMax) { ... }
  public void add(E elt) {
    if (elt.compareTo(max) > 0) { max = elt; }
  }
  public E getMax() { ... }
}
```

- `<E ... >` car `E` est le paramètre de type,
- `<E extends Comparable<E>>` car `E` doit implémenté la méthode de `Comparable<E>`.
- c'est `Comparable<E>` : `Comparable` est générique aussi.
- `extends` (dans ce contexte) permet donc d'imposer une *contrainte* d'interface sur le type générique.

Récapitulatif génériques

- méthodes génériques,
- interfaces génériques,
- classes génériques,
- contraintes d'interface.

```
public <T> Result myMethod(...) { ... }
public interface myInterface<T> { ... }
public class myClass<T> { ... }

... <T extends MyInterface> ...
```

Aix*Marseille université
Initiative d'excellence

Récapitulatif itérables

- `Iterable` vs `Iterator`,
- `Iterator` = `next()` et `hasNext()`
- les deux classes sont génériques,
- `for (E elt : myIterableObject)` si `myIterableObject` implémente `Iterable<E>`.
- `iter.hasNext()` et `iter.next()` si `iter` implémente `Iterable`.

Aix*Marseille université
Initiative d'excellence

Onzième partie XI

Classes abstraites

Aix*Marseille université
Initiative d'excellence

Exemple (I)

```
public class SumList {
  List<Integer> ints;

  public SumList(List<integer> ints) {
    this.ints = ints;
  }

  public int eval() {
    int sum = 0;
    for (Integer i : ints) {
      sum = sum + i;
    }
    return sum;
  }
}
```

Exemple (II)

```
public class ProductList {
  List<Integer> ints;

  public ProductList(List<integer> ints) {
    this.ints = ints;
  }

  public int eval() {
    int product = 1;
    for (Integer i : ints) {
      product = product * i;
    }
    return product;
  }
}
```

Factorisation, étape 1

```
public class SumList {
  List<Integer> ints;

  public SumList(List<integer> ints) {
    this.ints = ints;
  }

  public int eval() {
    int value = neutral();
    for (Integer i : ints) {
      value = operator(value,i);
    }
    return value;
  }

  private int operator(int a, int b) { return a + b; }
  private int neutral() { return 0; }
}
```

Factorisation : délégation

```
public class SumList {
  List<Integer> ints;
  private IntMonoid monoid;

  public SumList(List<integer> ints, IntMonoid monoid) {
    this.ints = ints;
    this.monoid = monoid;
  }

  public int eval() {
    int value = monoid.neutral();
    for (Integer i : ints) {
      value = monoid.operator(value,i);
    }
    return value;
  }
}
```

Factorisation : délégation (usage)

```
public interface IntMonoid {
  int neutral();
  int operator(int a, int b);
}

public class SumMonoid implements IntMonoid {
  public int neutral() { return 0; }
  public int operator(int a, int b) { return a + b; }
}

SumList list = new SumList(ints,new SumMonoid());
int sum = list.eval();
```

Délégation

Délégation

Une *délégation* consiste pour un objet déléguant à faire faire un travail par un autre objet délégué, propriété du déléguant.

Intérêt :

- garder la classe simple en exportant une partie du travail dans une autre classe (chaque classe a une seul rôle),
- permettre de paramètrer la classe selon le délégué. C'est donc une technique de factorisation.
- on a ainsi une classe pour les parties communes, et une classe pour chaque comportement distinct.

Deuxième solution : classe abstraite

Idée similaire : une classe avec la partie commune, des classes pour chaque comportement distinct.

Réalisation :

- faire une classe *à trous*, certaines méthodes restent sans implémentation (classe abstraite).
- faire des classes bouchant les trous : on implémente uniquement les méthodes qui ne sont pas implémentée par la classe abstraite (extension).
- la même classe à trou peut être complétée de multiples fois (plusieurs extensions)

Aix*Marseille université
Initiative d'excellence

Exemple de classe abstraite

```
public abstract class OperatorList {
  public abstract int neutral();
  public abstract int operator(int a, int b);

  protected List<Integer> ints;

  public OperatorList(List<Integer> ints) {
    this.ints = ints;
  }

  public int eval() {
    int value = neutral();
    for (int i : ints) {
      value = operator(value,i);
    }
    return value;
  }
}
```

Exemple d'extension

```
public class SumList extends OperatorList {
  public int neutral() { return 0; }
  public int operator(int a, int b) { return a + b; }

  public SumList(List<Integer> ints) {
   super(ints);
  }
}

public class ProductList extends OperatorList {
  public int neutral() { return 1; }
  public int operator(int a, int b) { return a * b; }

  public ProductList(List<Integer> ints) {
   super(ints);
  }
}
```

Déclaration, super, extension

Une classe abstraite est définie avec `abstract` :

```
public abstract class MyAbstractClass { ... }
```

Une extension est définie avec `extends` :

```
public class MyConcreteClass extends MyAbstractClass {
   ...
}
```

`MyAbstractClass` est alors la `superclasse` de `MyConcreteClass`. `MyConcreteClass` est une *extension* (ou *sous-classe*) de `MyAbstractClass`.

Aix*Marseille université
Initiative d'excellence

Méthodes abstraites ou concrètes

Une classe abstraite peut contenir des méthodes concrètes et des méthodes abstraites :

```
public abstract MyReturnType myMethod(MyArgType arg);
```

Une méthode abstraite est non-implémenté, son corps est remplacé par un “;”

Une extension soit est elle-même abstraite, soit implémente chaque méthode abstraite de la superclasse.

```
public MyReturnType myMethod(MyArgType arg) { ... }
```

Une extension *hérite* des méthodes concrètes de la superclasse.

Aix*Marseille université
Initiative d'excellence

Classes abstraites et propriétés

Une classe abstraite peut avoir des propriétés, commune à toutes les extensions.

Une propriété privée d'une classe abstraite **est inaccessible** depuis les extensions.

Pour avoir une propriété accessible par les extensions, mais pas par toutes les classes, on utilise la visibilité `protected`.

`protected` : visible par la classe, les extensions et les classes du même `package`. Invisible pour toute autre classe.
`protected` peut aussi s'appliquer aux méthodes, constructeurs, ...

Aix*Marseille université
Initiative d'excellence

Classes abstraites et constructeurs

Une classe abstraite **ne peut pas** être instanciée.

Une classe abstraite **peut** néanmoins posséder un constructeur :

- pour factoriser une partie de la construction du nouvel objet, quelque soit l'extension.
- les extensions doivent aussi avoir un constructeur. Le constructeur de l'extension **doit** commencer par un appel au constructeur de la classe abstraite, **s'il existe**, pour initialiser l'objet. Mot-clé `super`. Par défaut le constructeur de `super` sans argument est appelé.

```
public SumList(List<Integer> ints) {
 super(ints);
}
```

Exemple : les listes revisitées (I)

Interface d'une liste :

```
public interface LinkedList<E> implements Iterable<E> {

  boolean isEmpty();

  E head();
  LinkedList<E> tail();

  int length();
  LinkedList<E> add(E elt);
}
```

Exemple : les listes revisitées (II)

Une méthode non-implémentée déclarée dans une interface est considérée abstraite, on pourrait donc ne pas mentionner `isEmpty`, `head`,...

```
public abstract class AbstractList<E>
    implements LinkedList<E> {

  public abstract boolean isEmpty();
  public abstract E head();
  public abstract AbstractList<E> tail();
  public abstract int length();

  public AbstractList<E> add(E elt) {
    return new NonEmptyList<E>(elt,this);
  }
  public Iterator<E> iterator() {
    return new ListIterator<E>(this);
  }
}
```

Exemple : les listes revisitées (III)

`iterator` et `add` sont héritées.

Pas besoin de constructeur : `List` n'en a pas, pas de `super()` à faire.

```
public class EmptyList<E> extends List<E> {

  public boolean isEmpty() { return true; }

  public E head() {
    throw new NoSuchElementException();
  }
  public AbstractList<E> tail() {
    throw new NoSuchElementException();
  }

  public int length() { return 0; }
}
```

Exemple : les listes revisitées (IV)

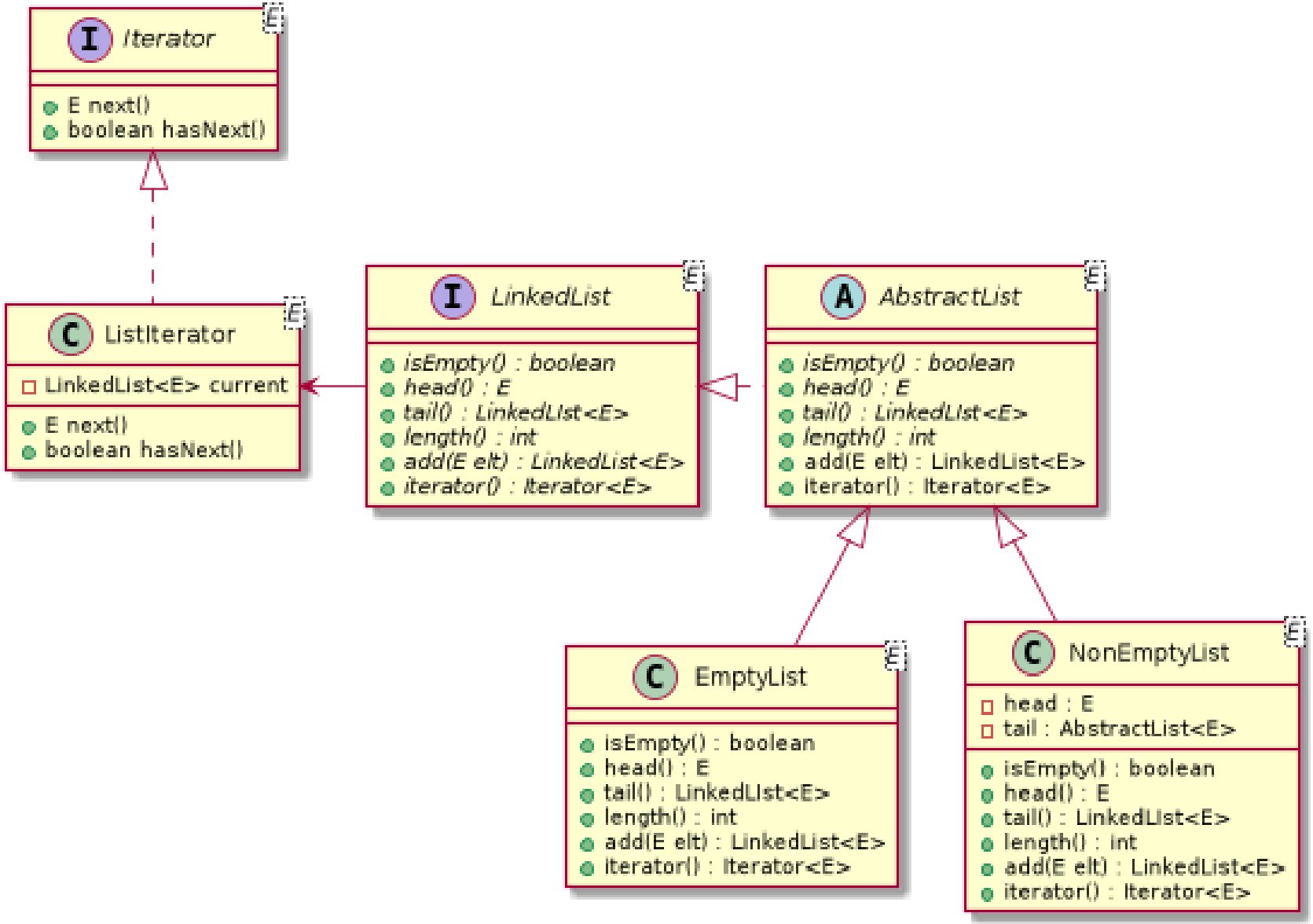
Ici non plus, pas de `super()` dans le constructeur. `iterator` et add sont hérités.

```
public class NonEmptyList<E> extends List<E> {
  private final E head;
  private final AbstractList<E> tail;

  public NonEmptyList(E head, AbstractList<E> tail) {
    this.head = head;
    this.tail = tail;
  }
  public boolean isEmpty() { return false; }
  public E head() { return head; }
  public AbstractList<E> tail() { return tail; }

  public int length() {
    return 1 + tail.length();
  }
}
```

Exemple : les listes revisitées (V)

```
public class ListIterator<E> implements Iterator<E> {
  private LinkedList<E> list;

  public ListIterator(LinkedList<E> list) {
    this.list = list;
  }
  public E next() {
    E current = list.head();
    list = list.tail();
    return current;
  }
  public boolean hasNext() {
    return !list.isEmpty();
  }
}
```

Diagramme de classes



Redéfinition de méthodes

L'extension peut redéfinir les méthodes implémentées de la classe abstraite (même arguments, même retour). Un objet de la classe étendue utilisera par défaut sa propre méthode (même dans un contexte où il possède le type de la classe abstraite).

```
abstract class A {
  public void foo() { System.out.println("A.foo"); }
}
class B extends A {
  public void foo() { System.out.println("B.foo"); }
}

B b = new B();
b.foo(); // "b.foo"
A a = b;
a.foo(); // "b.foo"
```

Redéfinition de méthode et `super`

L'extension peut faire appel à une méthode de sa superclasse avec le mot-clé `super`.

```
abstract class A {
  public void foo() { System.out.println "A.foo"; }
}
class B extends A {
  public void foo() {
    super.foo();
    System.out.println("B.foo");
  }
}

B b = new B();
b.foo(); // "a.foo", "b.foo"
```

Extensions générales

Les classes non-abstraites peuvent aussi être étendues :

- hérite des méthodes et propriétés de la superclasse,
- possède des nouvelles fonctionnalités (méthodes et propriétés),
- redéfinit des méthodes de la superclasse avec d'autres comportements.

Limitations :

- en Java, on ne peut étendre qu'une seule classe,
- on peut déclarer une classe finale, ce qui interdit de l'étendre,
- les extensions peuvent être à l'origine de problèmes subtils.

Aix*Marseille université
Initiative d'excellence

La classe `Object`

En Java, toute classe étend la classe `Object`.

```
boolean equals(Object obj);
int hashCode();
String toString();
...
```

```
Point point = new Point(3,2);
System.out.println(point.toString());
  // --> "test.Point@8c24e1"
System.out.println(point);
  // --> "test.Point@8c24e1"
```

Interface et méthodes par défaut

Une classe abstraite peut avoir des méthodes implémentées (non abstraite).

Une interface aussi peut avoir des méthodes implémentées : des méthodes par défaut (`default`).

- Une interface n'a pas de propriété (sauf constantes).
- Les méthodes par défaut peuvent appeler les méthodes non-implémentées de l'interface.
- Les méthodes par défaut sont signalées avec le mot-clé `default`.
- Une implémentation peut redéfinir les méthodes par défaut.

Aix*Marseille université
Initiative d'excellence

Méthode par défaut, exemple

```
public interface Monoid<E> {
  E neutral();
  E operator(E value1, E value2);

  default E reduce(List<E> values) {
    E accum = neutral();
    for (E value : values) {
      accum = operator(accum, value);
    }
    return accum;
  }
}
```

Redéfinition de méthode par défaut

```
public class ConjonctionMonoid
    implements Monoid<Boolean>
{
  public boolean neutral() { return true; }
  public boolean operator(boolean b1, boolean b2) {
    return b1 && b2;
  }

  public boolean reduce(List<Boolean> bools) {
    for (Boolean b : bools) {
      if (!b) return false;
    }
    return true;
  }
}
```

Classe abstraite vs interface

Que choisir entre classe abstraite et interface (et délégation) ?

- une classe abstraite correspond à une responsabilité (je suis . . .),
- une interface correspond à une capacité (je peux . . .).

Une *Renault twingo* est une *voiture* et peut être *conduite* :

- *twingo* est une classe,
- *voiture* est une classe abstraite,
- *conduisible* est une interface.

Aix*Marseille université
Initiative d'excellence

Classe abstraite vs interface

Classe abstraite :

- extension unique : chaque classe étend une seule autre classe.
- méthodes abstraites,
- possède des propriétés,
- visibilités variées (`protected`, `private`,...).

Interface :

- implémentation multiple : une classe implémente autant d'interfaces que nécessaire.
- méthodes par défaut,
- pas de propriétés (sauf constantes, implicitement `public static final`),
- tout est public.

Aix*Marseille université
Initiative d'excellence