

Rappels programmation orientée objet et Java : partie 1

Arnaud Labourel arnaud.labourel@univ-amu.fr



Section 1

Objectif des rappels

Apprendre à programmer proprement

Un programme propre :

- respecte les attentes des utilisateurs
- est fiable
- peut évoluer facilement/rapidement
- est compréhensible par tous

Méthode pour programmer proprement

- nommer correctement les éléments du code
- écrire du code lisible (par un autre humain)
- relire et améliorer le code

S'initier au penser objet

Connaître les éléments de base de la programmation objet :

- maîtriser le vocabulaire de la programmation objet : *classe*, *instance*, *méthode*, *attribut*, *constructeur*, *interface*, ...
- être capable d'utiliser les objets des classes du JDK (Java Development Kit)
- être capable de coder de nouvelles classes pour définir des nouveaux types d'objets
- savoir décomposer un problème simple en classes et objets
- savoir tester le comportement d'objets

Section 2

Premiers pas en Java et différences avec Python

Python

Lors du premier semestre, vous avez codé en Python dans le cours de mise en œuvre informatique :

Exemple de code Python :

```
def is_even(n):  
    return n % 2 == 0
```

```
>>> is_even(10)
```

```
True
```

```
>>> is_even("toto")
```

```
...
```

```
TypeError: not all arguments converted during  
    string formatting
```

```
static boolean isEven(int n){  
    return n % 2 == 0;  
}
```

En java, il faut définir le type des paramètres

```
>>> isEven(10)
```

True

```
>>> isEven("toto")
```

Error:

```
| incompatible types: java.lang.String cannot be  
| converted to int isEven("toto")
```

En cas de type incompatible donné en argument, une erreur se produit avant l'exécution de la fonction.

Définition d'un type

Un type de données définit :

- l'ensemble des valeurs possibles pour les données du type
- les opérations applicables sur ces données

Java vs Python

- Python :
 - ▶ toute donnée/valeur a un type
 - ▶ les variables/paramètres **n'ont pas** de types
- Java :
 - ▶ toute donnée a un type, toute variable a un type
 - ▶ les variables/paramètres ont un type

Typage dynamique

En Python, le type d'une valeur d'une variable est défini lors de son affectation, et le type de la valeur de la variable peut changer au cours d'une exécution.

```
>>> x = 8
>>> x
8
>>> type(x)
<class 'int'>
>>> x = "toto"
>>> type(x)
<class 'str'>
```

⇒ Le typage est dit **dynamique**

Typage statique

En Java, les variables ont chacune un type défini à sa déclaration.

Il détermine les valeurs que peut prendre la variable et ce type ne peut pas changer.

Le compilateur vérifie le typage.

```
int x = 8;  
x; /* 8 */  
x = "timoleon"; // interdit ! ne compile pas
```

⇒ Le typage est dit **statique**

Java est un langage **typé statiquement** :

- les variables, paramètres et retours de méthode ont des types
- une vérification de compatibilité des types est effectué à la compilation

Python est un langage **typé dynamiquement** :

- les variables, paramètre et retour de méthode **n'ont pas** de types
- les valeurs ont des types
- des erreurs liés au types se produisent à l'exécution

Analyse du typage statique

Avantages du typage statique

- séparation entre erreurs de types à la compilation et autres erreurs à l'exécution
- aide à la programmation (surtout avec la complétion automatique)
- aide à l'utilisation de bibliothèque (les types aident à comprendre l'usage des fonctions)
- exécutable généralement plus rapide (code plus facile à optimiser)

Désavantages du typage statique

- verbeux (code plus long à écrire)
- généricité plus difficile (difficile de définir des fonctions acceptant plusieurs types)
- code plus difficile à modifier

Autre exemple Python

```
def even_integers(bound):  
    list_even_integers = []  
    for i in range(0, bound):  
        if is_even(i):  
            list_even_integers += [i]  
    return list_even_integers  
  
>>> print(even_integers(10))  
[0, 2, 4, 6, 8]
```

Même exemple en Java

```
static List<Integer> evenIntegers(int bound) {  
    List<Integer> evenIntegers = new ArrayList<>();  
    for(int i = 0; i < bound; i++){  
        if(isEven(i)){  
            evenIntegers.add(i);  
        }  
    }  
    return evenIntegers;  
}  
  
>>> System.out.println(evenIntegers(10));  
[0, 2, 4, 6, 8]
```

Points communs Java/Python

- concept de fonctions similaire :
 - ▶ une fonction est définie par un bloc d'instruction
 - ▶ une fonction a des paramètres
 - ▶ une fonction peut retourner une valeur (instruction `return` similaire dans les deux langages)
- concept de variables/arguments similaire (sauf au niveau du typage)
- concept de bloc d'instructions similaire
- concept de structures de contrôle similaire : `for`, `if`, `while` présents en Java et Python

Différences Java/Python

- Même si les concepts sont similaires, la syntaxe (manière d'écrire est souvent différente) :
 - ▶ les blocs en python sont définis par l'indentation alors qu'en java il le sont par les accolades { }.
 - ▶ instructions séparées par des ; en java.
- Typage statique vs typage dynamique $\Rightarrow$ il faut préciser le type des éléments d'une liste en Java.
- Très peu d'opérateurs disponibles sur les objets en Java
 - ▶ pas de + sur des listes
 - ▶ pas de * sur les String (mais quand même le +)
- Les for sur les entiers *old school* en Java.
- Normes de nommage différentes pour les variables/arguments/fonctions :
 - ▶ snake_case en Python
 - ▶ camelCase en Java

Section 3

Types et structures de données

Types simples

Un même type basique peut correspondre à des données très différentes.

Par exemple, un entier peut correspondre à :

- un âge exprimé en année
- une température exprimée en Celsius
- un prix exprimé en €

```
int temperatureInCelsius = 8;
```

```
int age = 8;
```

```
int price = 8;
```

Fonction sur les types simples (1/2)

```
public boolean isSenior(int age) {  
    return age >= 70;  
}
```

Techniquement, on peut utiliser cette fonction sur toutes les données entières :

```
ageIsSenior(age);  
ageIsSenior(temperatureInCelsius);  
ageIsSenior(price);
```

Cela n'a pas de sens pour les entiers ne correspondant pas à un âge:

⇒ la fonction ne devrait pouvoir être appliquée qu'à des « âges »

Fonction sur les types simples (2/2)

```
public float toFahrenheit(int temperatureInCelsius) {  
    return 9 / 5 * temperatureInCelsius + 32;  
}
```

Techniquement, on peut utiliser cette fonction sur toutes les données entières :

```
toFahrenheit(temperatureInCelsius);  
toFahrenheit(age);  
toFahrenheit(price);
```

La fonction ne devrait pouvoir être appliquée qu'à des températures.

⇒ les types de base ne suffisent pas

On a besoin de distinguer âges et températures même si les deux données sont représentées par un entier

Types composites

Parfois, on a besoin de plusieurs données pour représenter des notions.

On souhaite travailler sur des personnes caractérisées par leur nom et année de naissance (données).

```
int a1 = 2004;  
String a2 = "Bob";  
int b1 = 2001;  
String b2 = "Alice";
```

Des variables séparées ne conviennent pas car on souhaiterait que les données soient regroupées.

Solution : types composites

Il faut un type composite (type contenant des éléments de différents types), par exemple en Python un dictionnaire

```
>>> alice = { "year" : 2001, "name" : "Alice" }
>>> bob = { "year" : 2006, "name" : "Bob" }
def is_adult(person) :
    return 2022 - person['year'] >= 18
>>> is_adult(alice)
True
>>> is_adult(bob)
False
```

Définir une fonction commune pour les variables bob et alice est possible car elles partagent la même structure de clés.

Partager des clés n'est pas suffisant

On pourrait aussi considérer des livres caractérisés par leur année de parution et leur nom :

```
>>> lotr = { "year" : 1954,  
             "name" : "Seigneur des Anneaux" }  
  
>>> is_adult(lotr)  
True
```

La fonction `is_adult` n'a pas de sens pour des livres et ne devrait pouvoir être appliqué qu'à des « personnes ». On a le même problème que pour les types simples.

⇒ partager les mêmes clés ne suffit pas

- Les types de base (simples ou composites) ne suffisent pas
 - ⇒ il faut pouvoir distinguer un « int age » et un « int temperature »
- Des données composites de même nature doivent partager la même structure
- La possibilité d'appliquer une opération à une donnée doit être contrôlée selon la nature de la donnée

⇒ il faut pouvoir définir ses propres **types**

Type : besoin

Un type définit les valeurs possibles et les opérations autorisées.

Exemples:

- un type `Temperature`
 - ▶ valeurs possibles : la valeur en degré Celsius de type `int`
 - ▶ opérations autorisées : obtenir la valeur en Fahrenheit
- un type `Age`
 - ▶ valeurs possibles : une donnée de type `int`
 - ▶ opérations autorisées : savoir si l'âge correspond à un âge sénior
- un type `Personne`
 - ▶ valeurs possibles : une année de naissance de type `int` et un nom de type `String`
 - ▶ opérations autorisées : savoir si la personne est majeure, obtenir l'âge

Section 4

Objet et classe

Principe programmation objet

- En programmation objet, les données sont représentées par des objets
- La programmation objet consiste à définir les types des objets

Objet

- un objet est composé de données (= d'autres objets)
- un objet peut exécuter des opérations
- un objet a un type et le type d'un objet définit :
 - ▶ la **structure** des données qui composent cet objet: **les attributs**
 - ▶ les **opérations** (= **traitements**) que peut exécuter cet objet: **les méthodes**

Une **classe** est un type objet.

Une classe définit :

- les **méthodes** et les traitements associés $\Rightarrow$ le **comportement** des objets
- la structure des **attributs** nécessaires à la réalisation des traitements $\Rightarrow$ l'**état** des objets

le **comportement** agit sur l'**état** et l'**état** influence le **comportement**

Méthodes et attributs

Méthode

Une **méthode** est une fonction qui appartient à une classe :

« *function member* »

- une méthode ne peut être utilisée que par les objets de la classe
- un objet ne peut exécuter que les méthodes de sa classe

Attribut

Un **attribut** est une variable qui appartient à un objet :

« *data member* » (synonymes : *property*, *field*, ...)

- les attributs sont définis par la classe de l'objet
- chaque attribut a un type

Instance

Une classe permet de **créer** des objets.

Ces objets sont du type de cette classe.

Définition d'instance

On appelle **instance** un objet créé par une classe.

Tout objet est l'instance d'une classe.

Exemples

- "toto" est une instance de la classe String (un objet issu de la classe String).
- `new ArrayList()` est une instance de la classe ArrayList (un objet issu de la classe ArrayList).

Définition d'une classe

❶ Définir les **attributs** des objets de la classe

- Tous les objets d'une même classe ont **la même structure d'attributs**
- Les valeurs des attributs sont « personnelles » à chaque objet, elles peuvent être **différentes** d'un objet à l'autre.

❷ Définir les **méthodes** que pourront exécuter les objets de la classe

- tous les objets d'une même classe peuvent exécuter les mêmes méthodes
- appeler (ou invoquer) une méthode sur un objet n'affecte que l'objet en question
- appeler une méthode sur un objet peut changer la valeurs de ses attributs (effet de bord)

Exemple d'attributs d'une classe Person

On peut donc définir une classe correspondant à une personne et contenant deux attributs :

- une année de naissance: `int` nommé `year`
- un nom : `String` (chaîne de caractères) nommé `name`

On utilise la syntaxe suivante dans un fichier `Person.java`

```
public class Person {  
    int year;  
    String name;  
}
```


Accès aux attributs

Pour accéder aux attributs d'un objet il faut utiliser la syntaxe suivante (notation pointée) :

`nomDeLaVariable.nomDeLAttribut`

Si on est dans le code d'une méthode, on peut seulement utiliser:

- `nomDeLAttribut` ou
- `this.nomDeLAttribut`

pour accéder à l'attribut de l'objet courant.

Exemples

- `person.name` pour accéder à l'attribut `name` d'une instance `person` de type `Person`
- `array.length` pour accéder à l'attribut `length` d'un tableau `array` de type

Exemple de méthodes d'une classe Person

On peut définir des **méthodes** qu'on peut appeler à partir d'une personne.

```
public class Person {  
    int year;  
    String name;  
    int getAge(){  
        return 2022 - year;  
    }  
    boolean isAdult(){  
        return getAge() >= 18;  
    }  
}
```

Les méthodes sont liés aux objets de la classe et se définissent sans le mot-clé `static`.

Accès aux méthodes

Une méthode s'appelle avec une instance de la classe.

Pour accéder aux méthodes d'un objet il faut utiliser la syntaxe suivante (notation pointée) :

`nomDeLaVariable.nomDeLaMéthode(arguments)`

Si on est dans le code d'une méthode, on peut seulement utiliser `nomDeLaMéthode(arguments)` ou `this.nomDeLaMéthode(arguments)` pour appeler une méthode sur l'objet courant.

Exemples

- `person.getAge()` pour appeler la méthode `getAge` sans argument sur l'objet `person` de type `Person`
- `string.charAt(i)` pour appeler la méthode `charAt` avec un argument `i` sur l'objet `string` de type `String`

À l'exécution, il faut créer les objets conformément au modèle.

Constructeur

Pour créer un objet il faut utiliser un **constructeur**.

Chaque appel à un constructeur crée un **nouvel** objet (instance) qui obéit au modèle défini par la classe du constructeur.

Rôles du constructeur

Un constructeur a deux rôles :

- ❶ créer les attributs de l'objet (la structure de l'état) $\Rightarrow$ réserver l'espace mémoire (automatique en Java)
- ❷ donner les valeurs initiales aux attributs (« initialiser l'objet » : code du constructeur)

Chaque classe doit définir comment sont initialisés les attributs

Exemple de constructeurs d'une classe Person

```
public class Person {  
    int year;  
    String name;  
    // Méthodes getAge et isAdult  
    Person(int year, String name) {  
        this.year = year;  
        this.name = name;  
    }  
}
```

Accès aux constructeurs d'une classe

Appel du constructeur

`new + nomDeLaClasse (+ arguments)`

Exemples

- `new Person("Alice", 2001)` crée un objet de type `Person`
- `new ArrayList()` crée un objet de type `ArrayList`

Constructeur implicite

En Java, si une classe ne définit pas de constructeur, alors il y a un constructeur par défaut (constructeur sans paramètre n'ayant pas d'instruction réservant juste l'espace mémoire pour l'objet)

→ il existe seulement s'il n'y a pas d'autre constructeur déclaré

- L'appel à un constructeur a pour résultat une référence vers l'objet créé.
- Cette référence = un pointeur vers l'identité de l'objet.

Elle peut être stockée dans une variable (de type objet).

Important

- La référence permet d'accéder à l'objet, mais n'est pas l'objet lui-même.
- Une variable objet contient l'information pour accéder à l'objet (et potentiellement le modifier).
- Deux variables ayant la même référence d'un objet (par exemple après une affectation `a = b;`) pointe vers un objet unique $\Rightarrow$ toute modification de l'objet est visible via les deux variables.

Ajout d'autres méthodes dans Person

```
public class Person {  
    int year;  
    String name;  
  
    //Code des transparents précédents  
  
    boolean isOlder(Person person){  
        return getAge() > person.getAge();  
    }  
  
    @Override  
    public String toString() {  
        return name + " (" + year + ")";  
    }  
}
```


Diagramme de classe

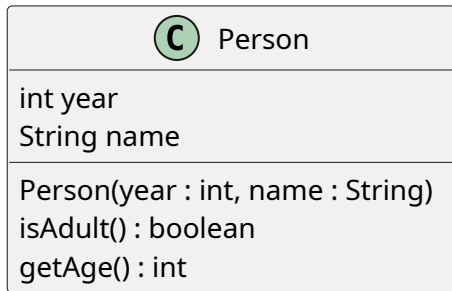


Diagramme de classe

Manière “simple” de représenter une classe avec :

- son nom
- ses attributs
- ses constructeurs (types des paramètres)

Diagramme d'objets

alice : Person
year = 2001 name = "Alice"

bob : Person
year = 2006 name = "Bob"

Diagramme d'objet

Manière “simple” de représenter un objet avec :

- son type
- les valeurs de ses attributs.

Section 5

Classes : à retenir absolument

Définition d'une classe

Une **classe** (d'objet) définit des :

- **constructeurs** : des façons de construire/instancier les objets (**instances** de la classe)
- **attributs** (champs, propriétés ou données membres) : la structure des objets de la classe
- **méthodes** : le comportement des objets de la classe

Syntaxe de définition d'une classe

```
public class NameOfTheClass{  
    Type1 attribute1;  
    Type2 attribute2;  
  
    NameOfTheClass(Type3 argument1, Type4 argument2){  
        // constructor code  
    }  
  
    Type2 getAttribute2(){  
        return this.attribute2;  
    }  
}
```

Syntaxe d'utilisation d'une classe

```
>>> Person p = new Person(2000, "Bob");  
>>> p.year;  
2000  
>>> p.name = "Alice";  
>>> p.name  
Alice  
>>> p.getAge();  
22  
>>> p.isAdult();  
true
```

Section 6

Appel de méthodes

Utilisation d'objet

On interagit avec un objet uniquement via :

- ses méthodes
- ses attributs

Appel/invocation de méthode

Appel/invocation de méthode = « envoi de message »

L'objet a le contrôle sur le message qui lui est envoyé :

⇒ seulement les méthodes définies dans la classe de l'objet sont autorisées.

Utilisation de la notation pointée :

`aPerson.isAdult()`

Exemple : classe Thermometer

Classe Thermometer

- structure attributs : value de type int
- méthodes : getValue(), toFahrenheit(), toString()

t1 et t2 deux instances de la classe Thermometer

⇒ même structure d'attributs, mais valeurs différentes

t1 : Thermometer	
value	12

t2 : Thermometer	
value	20

Exemple de code : classe Thermometer

```
public class Thermometer {  
    private float temperature; // attribut  
    public Thermometer(float initialTemp) {  
        // constructeur  
        this.temperature = initialTemp;  
    }  
    public float temperatureInCelsius() {  
        // méthode  
        return this.temperature;  
    }  
    public void changeTemperature(float newTemp) {  
        // méthode  
        this.temperature = newTemp;  
    }  
}
```

Exemple de code : classe Thermometer

```
// suite
public String toString() { // méthode
    return "Current temperature is : "
        + this.temperature + "°C";
}
public float temperatureInFahrenheit() { // méthode
    return (9.0f/5.0f)*this.temperature+32;
}
}
```

Exemple : classe Person

Classe Person :

- structure attributs : birthYear de type int, name de type String
- méthodes : isAdult(), isOlderThan(Person), getBirthYear(), getName(), toString()

alice et bob deux instances de la classe Person

alice : Person	
name	"Alice"
birthYear	2001

bob : Person	
name	"Bob"
birthYear	2006

Exemples d'utilisations

Thermometer

value : int

getValue() : int toFahrenheit() : int toString() : String

Person

name : String birthYear : int

getName() : String

getBirthYear() : int

isAdult() : boolean

isOlderThan(p : Person) : boolean toString() : String

```
t1.toFahrenheit() // -> 53.6
```

```
t2.toFahrenheit() // -> 68
```

```
t1.toString() // -> "Current temperature is : 18°C"
```

```
t1.isAdult() // impossible (erreur à la compilation)
```

```
alice.getBirthYear() // -> 19
```

```
alice.isAdult() // -> true
```

```
alice.toString() // -> "Alice is born in 2001"
```

```
alice.toFahrenheit()
```

Envoi de message

Conformément à sa classe (son type), l'objet « contrôle » le message :

- sa légalité
- pas d'ambiguïté pour deux messages (méthodes) de même nom dans des classes différentes (cf. `toString()`)

Règle d'appel de méthode

- Une méthode ne peut pas être utilisée autrement qu'en étant appelée/invoquée sur un objet via un envoi de message à cet objet
- La validité du message pour le type de la référence est vérifié à la compilation

L'objet invoquant = le **receveur** du message

Il fait partie du contexte d'exécution de la méthode

⇒ dans le code d'une méthode, on peut :

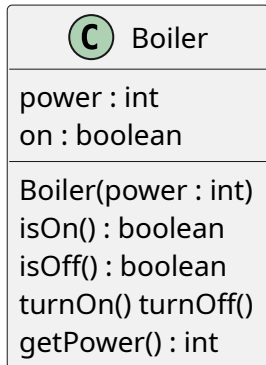
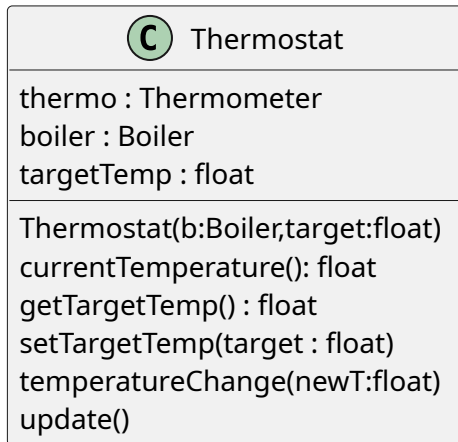
- appeler une méthode sur le receveur
- accéder aux attributs du receveur

Messages en cascade

Les attributs d'un objet étant eux-mêmes des objet : il peut y avoir des messages en cascade.

⇒ Appel d'une méthode d'un receveur peut entraîner l'appel d'une méthode sur un de ses attributs.

Exemple thermomètres, chaudières et thermostats



```
Boiler b = new Boiler(1000);  
Thermostat t = new Thermostat(b, 20);
```


Classe et instance

Classe = définition/modèle

- décrit la structure de l'état des objets : les attributs et leurs types
- définit les envois de messages possibles : les méthodes
⇒ **interface** d'une classe

Abstrait

Instance = objet conforme au modèle de la classe qui l'a créé

- son état correspond à la structure définie par sa classe
→ association de valeurs aux attributs
- n'accepte que les messages définis par la classe
= n'exécute que les méthodes définies par sa classe

Concret

Section 7

Le penser objet

Analyse d'un problème en programmation objet

Penser objet : décomposer le programme en objets

- Quels sont les objets nécessaires à la résolution du problème ?
⇒ décomposition du problème en objets
- À quels modèles des objets correspondent-il ?
⇒ Quelles sont les classes ?
- Quels sont les fonctionnalités/opérations dont on doit/veut pouvoir disposer sur ces objets ?
⇒ Quelles sont les méthodes des classes ?
- Quelle est la structure des données de l'objet ?
⇒ Quelles sont les attributs des classes ?

Exemple de problème

- un catalogue regroupe des articles, il permet de trouver un article à partir de sa référence.
- un article est caractérisé par un prix et une référence que l'on peut obtenir. On veut aussi pouvoir déterminer si un article est plus cher qu'un autre
- une commande est créée pour un client et un catalogue donnés, on peut ajouter des articles à une commande, accéder à la liste des articles commandés ainsi que prix total des articles et le montant des frais de port de la commande.
- un client peut créer une commande pour un catalogue et commander dans cette commande des articles à partir de leur références.

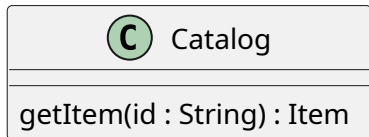
Classes du problèmes

- un **catalogue** regroupe des **articles**, il permet de trouver un article à partir de sa **référence**.
- un **article** est caractérisé par un prix et une référence que l'on peut obtenir. On veut aussi pouvoir déterminer si un article est plus cher qu'un autre
- une **commande** est créée pour un **client** et un **catalogue** donnés, on peut ajouter des **articles** à une commande, accéder à la liste des articles commandés ainsi que prix total des articles et le montant des frais de port de la commande.
- un **client** peut créer une **commande** pour un catalogue et commander dans cette commande des **articles** à partir de leur références.

- un catalogue regroupe des articles, il permet de **trouver** un article à partir de sa référence.
- un article est caractérisé par un prix et une référence que l'on **peut obtenir**. On veut aussi pouvoir **déterminer** si un article est plus cher qu'un autre
- une commande est créée pour un client et un catalogue donnés, on peut **ajouter** des articles à une commande, **accéder** à la liste des articles commandés ainsi que prix total des articles et le montant des frais de port de la commande.
- un client peut **créer** une commande pour un catalogue et **commander** dans cette commande des articles à partir de leur références.

Description d'un catalogue

un catalogue regroupe des articles, il permet de **trouver** un article à partir de sa référence.



Description d'un article

un article est caractérisé par un prix et une référence que l'on **peut obtenir**. On veut aussi pouvoir **déterminer** si un article est plus cher qu'un autre

 Item

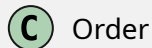
getPrice() : double

getReference() : String

isMoreExpensiveThan(other : Item) : boolean

Description d'une commande

une commande est **créée** pour un client et un catalogue donnés, on peut **ajouter** des articles à une commande, **accéder** à la liste des articles commandés ainsi que prix total des articles et le montant des frais de port de la commande.



```
Order(client : Client, catalog : Catalog)
addItem(item : Item)
allItems() : List<Item>
getTotalPriceOfItems() : double
getShippingCost() : double
getClient() : Client
getCatalog() : Catalog
```

Description d'un client

un client peut **créer** une commande pour un catalogue et **commander** dans cette commande des articles à partir de leur références.



Client

```
createOrder(Catalog catalog) : Order  
orderItem(Order order, String reference)
```

Utilisation des classe

Créer une commande pour un client, faire commander 2 articles par le client, obtenir le prix total des articles de la commande

on suppose les références disponibles et initialisées :

```
Client client = new Client(...)  
Catalogue cata = new Catalogue(...)  
  
Order order = client.createOrder(cata);  
client.orderItem(order, "A0527");  
client.orderItem(order, "B3879");  
float price = order.getTotalPrice();
```

Code des méthodes

La méthode `orderItem` de `Client` permet d'ajouter un article à une commande à partir de son identifiant quel code pour cette méthode ?

```
public void orderItem(Order order, String id) {  
    // récupérer le catalogue de la commande  
    Catalog cata = order.getCatalog();  
    // récupérer l'article dans le catalogue  
    // à partir de la référence  
    Item item = cata.getItem(id);  
    // ajouter l'article à la commande  
    order.addItem(item);  
}
```

Section 8

Compilation

Compilation/interprétation

- En Java, le code est généralement compilé puis ensuite exécuté.
- En Python, le code est généralement directement interprété.

Compilation

code source (langage de programmation) → code exécutable (langage machine)

Plusieurs phases :

- analyse lexicale (découpe le code en *tokens*)
- analyse syntaxique (vérifie la syntaxe)
- analyse sémantique (vérifie le typage)

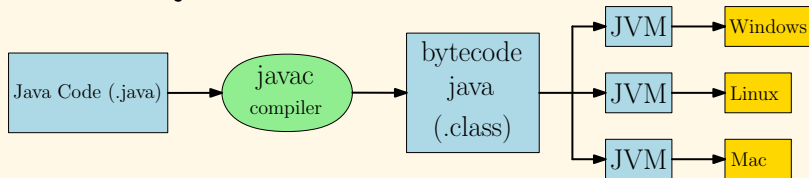
Commandes

- `javac MonFichier.java` pour compiler le fichier `MonFichier.java`
- `java MonFichier` pour exécuter le code de `MonFichier.java`

Compilation et machine virtuelle

Compilation Java

- le compilateur java génère du **bytecode java** = langage machine virtuel
- le code java s'exécute dans une **Java Virtual Machine (JVM)**



Machine virtuelle Java

- la JVM interprète le *bytecode* pour l'exécuter
 - la JVM rend le code java indépendant de l'OS et de la machine hôte
- ⇒ *Compile once, run everywhere*

Exemple compilation → exécution

Pour être exécuté un code Java doit définir une méthode main.

```
public class App {  
    public static void main(String[] args) {  
        System.out.println("Hello world !");  
    }  
}
```

- la fichier doit avoir le même nom que la classe (ici App et donc fichier App.java)
- La commande `javac App.java` compile le fichier et crée l'exécutable `App.class`.
- La commande `java App` exécute `App.class` et affiche donc `Hello world !`.

Exemple compilation → exécution avec arguments

```
public class App2 {  
    public static void main(String[] args){  
        for(String arg : args)  
            System.out.println(arg);  
    }  
}
```

- `String[] args` correspond aux arguments de l'appel d'exécution.
- La commande `java App2 to ta ti` exécute `App2.class` avec les arguments `to`, `ta` et `ti` et affiche donc :

```
to  
ta  
ti
```

Section 9

Encapsulation

Exemple classe Client

Pour le moment tout code ayant accès à un Item, a accès à son attribut price en lecture et écriture.

```
public class Client{  
    void orderItem(Order order, String id) {  
        Catalogue cata = order.getCatalogue();  
        Item item = cata.getItem(id);  
        item.price = 0; // l'item devient gratuit  
        order.addItem(item);  
    }  
}
```

Comment « empêcher » cela ?

Comment interdire la modification des attributs depuis l'extérieur ?

- **Restreindre** la visibilité des attributs ou méthodes d'une classe.
- En Java : **modificateurs** d'accès précisés lors de la définition d'attributs ou méthodes.

Mot-clés `private` et `public`

Utilisables à la déclaration des attributs, méthodes et constructeurs.

Deux modificateurs d'accès possibles :

- `private` : accessible uniquement pour les instances de la classe c'est-à-dire uniquement depuis le code des méthodes de la classe
- `public` : accessible pour tout le monde c'est-à-dire dans le code de n'importe quelle méthode

La classe Item

```
public class Item {  
    private float price;  
    private String id;  
    public float getPrice() {  
        return this.price;  
    }  
    public float getId() {  
        return this.id;  
    }  
    public boolean moreExpensiveThan(Item otherItem) {  
        return this.price > otherItem.price;  
    }  
    public Item(float p, String id) {  
        this.price = p;  
    }  
}
```

Méthode getTotalPrice dans Order

```
public class Order {  
    List<Item> items;  
    public float getTotalPrice() {  
        float total = 0;  
        // cumuler les prix de tous les articles  
        for(Item item : this.items) {  
            total += item.price; // interdit private !  
        }  
        return total;  
    }  
}
```

Erreur à la compilation : price étant un attribut privé, il n'est pas possible d'y accéder depuis une autre classe (ici Order).

Méthode getTotalPrice dans Order

```
public class Order {  
    List<Item> items;  
    public float getTotalPrice() {  
        float total = 0;  
        // cumuler les prix de tous les articles  
        for(Item item : this.items) {  
            total += item.getPrice();  
            // autorisé public !  
        }  
        return total;  
    }  
}
```

On peut accéder à la valeur de price grâce à un getter qui lui est public.

Règle d'encapsulation

Règle

Rendre privés les attributs caractérisant l'état de l'objet et fournir si besoin des méthodes publiques permettant de modifier/accéder à l'attribut

get et set

accesseur/mutateur = getter/setter

Pour un attribut `float price` :

- `getPrice()` : accesseur
- `setPrice(float newPrice)` : mutateur/modificateur

Exemple getter/setter

```
private float price;
```

```
public float getPrice() {  
    // accès en lecture  
    return this.price;  
}
```

```
public void setPrice(float newPrice) {  
    // accès en écriture  
    this.price = newPrice;  
}
```

Principe d'encapsulation

Ne pas laisser l'état d'un objet en libre accès en modification

Intérêt de l'encapsulation

- **masquer l'implémentation**

→ toute la décomposition du problème n'a pas besoin d'être connue du « programmeur utilisateur »

- **protéger**

→ l'objet a le contrôle sur son état (contrôle sur les modifications)

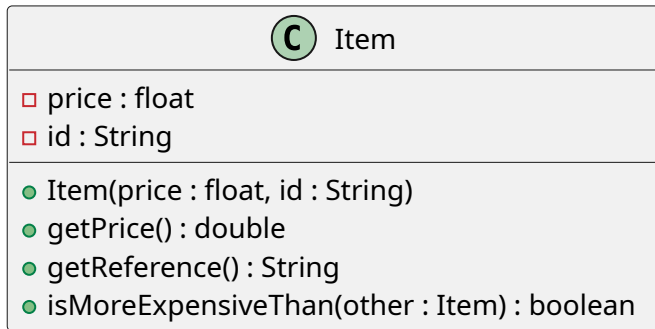
→ préserver l'intégrité des objets

→ le « programmeur créateur » contrôle (et est responsable) de son interface par rapport au « programmeur utilisateur »

- **permettre l'évolutivité**

→ il est possible de modifier tout ce qui n'est pas public sans impact pour le « programmeur utilisateur »

private et public dans les diagrammes de classes



Règle

- = public (autre symbole possible +)
- = private (autre symbole possible -)

Encapsulation et contrôle de l'état

Toutes les valeurs autorisées par le type des attributs ne sont pas forcément correctes pour l'objet :

- attribut `int month` dans `Date` doit être entre 1 et 12
- attribut `double celsiusValue` dans `Temperature` doit être supérieur à -273.15 (zéro absolu)
- attribut `double price` dans `Item` doit être positif

Toutes les valeurs autorisées par le type des paramètres d'une méthode ne sont pas forcément correctes :

- paramètre `double amount` dans `withdraw` doit être supérieur ou égal à 0 (pas de retrait d'argent d'un montant négatif)
- paramètre `int divisor` dans `divide` ne doit pas être égal à 0 (pas de division par zéro)

Contrôle avec mot clé assert

En Java, il est possible d'utiliser le mot-clé assert pour vérifier que certains prédicats sont vrais.

```
public class Temperature {  
    private double valueInCelsius;  
    public Temperature(double valueInCelsius) {  
        assert valueInCelsius >= -273.15 :  
            (valueInCelsius + " temp not allowed");  
        this.valueInCelsius = valueInCelsius;  
    }  
}
```

⇒ vérifie que valueInCelsius est supérieur à -273.15.

Exemple de fonctionnement d'assert

```
public class Temperature {  
    private double valueInCelsius;  
    public Temperature(double valueInCelsius) {  
        assert valueInCelsius >= -273.15 :  
            (valueInCelsius + " temp not allowed");  
        this.valueInCelsius = valueInCelsius;  
    }  
    public static void main(String[] args){  
        Temperature t = new Temperature(-300.);  
    }  
}
```

Exception in thread "main"

java.lang.AssertionError: -300.0 temp not allowed

at Temperature.<init> (Temperature.java:4)

Règles d'assert

Syntaxe d'assert

```
assert condition : message;
```

Fonctionnement d'assert

- Si la condition est fausse, l'exécution s'arrête (fin du programme) et le message est affiché via une Exception de type `AssertionError`.
- Si la condition est vraie, rien ne se passe.

Activation d'assert

De base, les assert ne sont pas activés, il faut utiliser l'option `-ea` dans la commande java pour les activer :

```
java -ea Main
```

Exemple assert

```
public class Date {  
    private int month;  
    private int year;  
    public Date(int month, int year) {  
        this.setMonth(month);  
        this.year = year;  
    }  
    void setMonth(int month){  
        assert 1 <= month && month <= 12 :  
            (month + " month number not allowed");  
        this.month = month;  
    }  
}
```

✓ vérifie que month est compris entre 1 et 12

Section 10

Les types primitifs

Les types primitifs

En java, il existe des types **primitifs** qui ne sont pas des objets :

type	catégorie	taille	valeurs possibles	affichage
byte	entier	8 bits	-128 à 127	0
short	entier	16 bits	-32768 à 32767	0
int	entier	32 bits	-2^{31} à $2^{31} - 1$	0
long	entier	64 bits	-2^{63} à $2^{63} - 1$	0
float	flottant	32 bits		0.0
double	flottant	64 bits		0.0
char	caractère	16 bits	caractère unicode	'\000'
boolean	booléen	non définie	false ou true	false

Types primitifs vs objets

- Les noms des types primitifs commencent par un minuscule (vs majuscule pour les types objets).
- Il n'y a pas de classe associée aux types primitifs et ils ne peuvent pas être utilisés pour appeler une méthode.
- Il n'y a pas de constructeurs pour les types primitifs (pas d'instanciation).
- Les variables de type primitif contiennent directement la valeur et **pas** une référence comme c'est le cas pour les objets.
- On affecte les variables de type primitifs avec :
 - ▶ des littéraux : 'a', 12, 42.0, 42.0f, ...
 - ▶ des résultats d'opérations : 2. + x, 2 / 5, 3 * 4, ...

Comportement types primitifs

Lors d'un appel de méthode les arguments sont passés par valeur : une copie de la valeur de l'argument est créé lors de l'appel.

Cela un impact différent suivant que l'argument soit un objet ou un type primitif :

- Pour les objets, cela signifie passer une copie de la **référence** : il est donc possible de modifier l'état de l'objet.
- Pour les types primitifs, cela signifie que l'argument est une **copie** uniquement créée pour l'appel et toute modification de sa valeur n'aura pas d'impact en dehors de l'appel.

Exemple comportement objet

```
public class Point {  
    private double x, y;  
    public Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
    public double getX() { return x; }  
    public double getY() { return y; }  
    public void setX(double x) { this.x = x; }  
    public void setY(double y) { this.y = y; }  
    public String toString() {  
        return "(" + x + ", " + y + ")";  
    }  
}
```

Exemple comportement objet receveur

```
public class App{  
    public static void main(String[] args){  
        Point p = new Point(0., 0.);  
        p.setX(1.);  
        p.setY(2.);  
        System.out.println(point.toString());  
        // => affiche (1.0, 2.0)  
    }  
}
```

L'objet passé en notation pointée est modifié.

Exemple comportement objet en argument

```
public class Point {  
    public void copyInto(Point p){  
        p.setX(getX());  
        p.setY(getY());  
    }  
  
    public static void main(String[] args){  
        Point p1 = new Point(1, 2);  
        Point p2 = new Point(0, 0);  
        p1.copyInto(p2);  
        System.out.println(p2.toString());  
        // => affiche (1.0, 2.0)  
    }  
}
```

L'objet passé en argument est modifié.

Exemple comportement type primitif en argument

```
public class Point {  
    public double distanceTo(Point p){  
        return Math.hypot(p.x - x, p.y - y);  
    }  
    public void addDistanceTo(double d, Point p){  
        d += this.distanceTo(p);  
    }  
    public static void main(String[] args){  
        double d = 0;  
        Point p1 = new Point(1, 2);  
        Point p2 = new Point(0, 0);  
        p1.addDistanceTo(d, p2);  
        System.out.println(d); // => affiche 0.0  
    }  
}
```


Égalité de type primitif

Pour tester l'égalité de types primitifs *entiers* (byte, short, int, long, char et boolean), il suffit d'utiliser `==`.

Pour tester l'égalité de types primitifs *flottants* (float, double), il faut faire attention aux erreurs d'approximation.

Bonne comparaison entre 2 doubles d1 et d2

Fixer un ϵ petit et vérifier que $|d1 - d2| < \epsilon$.

Exemple :

```
double d1 = 0.1;
double d2 = 0.1 + 0.1 + 0.1;
double epsilon = 0.000001d;
d1 == d2 // false : mauvaise comparaison
Math.abs(d1 - d2) < epsilon; // true : bonne comparaison
```

Section 11

L'égalité d'objets

Deux égalités d'objets ?

Question

Quand peut-on dire que 2 objets sont égaux ?
Égalité d'objets ou de valeur ?

Égalité d'objet

Les deux références désignent le même objet :
Égalité testée par l'opérateur `==`

Égalité de valeurs

Les objets des deux références sont équivalents :
Égalité testée par la méthode `equals`

equals vs ==

```
String str1 = new String("Le Seigneur des Anneaux");  
String str2 = new String("Le Seigneur des Anneaux");
```

- 2 références différentes sur 2 objets différents `str1 == str2` $\Rightarrow$ false
- les deux objets référencés sont équivalents `str1.equals(str2)` $\Rightarrow$ true

La classe String définit une méthode equals :

```
public boolean equals(Object o) { ... }
```

Un equals pour chaque classe

equals doit être (re)définie et adaptée pour chaque classe
(par défaut, elle se comporte comme ==)

Code méthode equals

```
public class Item {  
    public boolean equals(Object o) {  
        if (o instanceof Item) {  
            // teste si o est un Item  
            Item other = (Item) o;  
            // cast o en Item afin d'accéder  
            // aux attributs d'Item  
            return (this.price == other.price) &&  
                (this.id.equals(other.id));  
        }  
        else {  
            return false;  
        }  
    }  
}
```

Explication code equals

- la méthode `equals` compare l'objet courant `this` avec un `Object o` (et pas un `Item`).
- `o` n'est pas forcément une instance d'`Item`. C'est pour cela, qu'on teste si `o` est une instance de `Item` avec `o instanceof Item` qui renvoie `true` si `o` est un `Item` et `false` sinon.
- Si `o` n'est pas un `Item` alors `equals` retourne `false`.
- Si `o` est un `Item`, on a besoin d'accéder à son `id` et son `price` pour pouvoir les comparer avec ceux de `this`. C'est pour cela qu'on fait un transtypage (*cast*) de `o` en `Item`.
- Le transtypage est obligatoire car une variable ou argument de type `Object` n'a pas d'attributs `id` ou `price`.

Section 12

Mot-clé final

Mot-clé final pour les attributs

Mot-clé final dans la déclaration d'un attribut : interdit la modification de la valeur de l'attribut après la construction de l'objet.

Exemple :

```
public class Integer {  
    public final int value;  
    public Integer(int value) {  
        this.value = value;  
    }  
}
```

- Un attribut final doit être initialisé après la construction de l'instance
- La valeur de l'attribut ne peut plus être modifiée ensuite (évite les effets de bord et permet de donner l'accès en lecture de l'attribut).

Utilisation mot-clé final

```
public class Integer {  
    public final int value;  
    public Integer(int value) {  
        this.value = value;  
    }  
}
```

```
Integer i = new Integer(2);  
i.value = 1; // Erreur à la compilation
```

final interdit l'affectation en dehors du constructeur et de l'initialisation par défaut.

Pourquoi final ?

- permet de donner l'accès à l'attribut sans crainte de modification (exemple : `length` des tableaux)
- permet de créer des objets immuable/immutable : objet dont l'état ne peut changer après leur construction :
 - ▶ évite la création d'effets de bord qui sont source d'erreurs en programmation
 - ▶ utile pour les petits objets (peu d'attributs : et donc facile de construire nouvelles instances)
 - ▶ utile pour les objets n'ayant pas vocation à changer d'état (objet contenant des données permettant un transfert d'information : `record`).

Exemple

Les `String` en Java sont immuables.

Section 13

Mot-clé static

Définition de π

```
public class Disc {  
    private double radius;  
    public Disc(double radius) { this.radius = radius; }  
    public double perimeter() {  
        return 2 * 3.14 * this.radius;  
    }  
    public double surface() {  
        return 3.14 * this.radius * this.radius;  
    }  
}
```

Bonne pratique

Il faut nommer les constantes (surtout si elles apparaissent plusieurs fois dans le code)
!

π en attribut ?

```
public class Disc {  
    private double radius; private double pi;  
    public Disc(double radius) {  
        this.radius = radius;  
        this.pi= 3.14;  
    }  
    public double perimeter() {  
        return 2 * this.pi * this.radius;  
    }  
    public double surface() {  
        return this.pi * this.radius * this.radius;  
    }  
}
```

un attribut « `this.pi` » pour chaque instance de `Disc` : est-ce raisonnable ?

Attributs de classe

Solution : définir un attribut de classe (lié à la classe toute entière plutôt qu'à chaque instance).

La définition de chaque classe est unique, donc les attributs de classes **existent en un seul exemplaire**.

Ils sont créés au moment où la classe est chargée en mémoire par la JVM
et ce quel que soit le nombre d'instances (y compris 0).

- Il n'est pas nécessaire de disposer d'une instance pour utiliser une caractéristique statique.
- Ils sont définis à l'aide du mot-clé `static`

Utilisation du mot-clé `static`

La déclaration des attributs de classe se fait à l'aide du mot réservé `static`
accès via le nom de classe (utilisation de la notation `.`)

```
public class Disc {  
    private double radius;  
    private static double pi = 3.14;  
    public Disc(double radius) {  
        this.radius = radius;  
    }  
    public double perimeter() {  
        return 2 * Disc.pi * this.radius;  
    }  
    public double surface() {  
        return Disc.pi * this.radius * this.radius;  
    }  
}
```

static, public et private

```
public class StaticExample {  
    private static int compteur;  
    public static double pi = 3.14159;  
}
```

Sens de public et private

StaticExample.compteur n'est visible qu'à l'intérieur de la classe StaticExample :

- dans des méthodes/constructeurs de la classe
- pour l'initialisation d'autres attributs

⇒ attribut de classe (privé) partagé par les instances de la classe

StaticExample.pi est visible partout

static : attributs

static est un mot-clé à utiliser avec parcimonie et pertinence

avec final : définition de constantes

```
public class ConstantExample {  
    public static final float PI = 3.141592f;  
    public static final String BEST_MOVIE_TITLE = "Matrix";  
}
```

- le qualificatif final signifie qu'une fois initialisée la valeur ne peut plus être modifiée, exemple : Double.MAX_VALUE, Math.PI, Boolean.TRUE, ...
- convention de nommage : les identifiants des constantes sont en majuscules avec _ pour séparer les mots (SNAKE_CASE).

Exemple d'utilisation

```
public class Order {  
    // attributs de classes : static  
    private static final String ORDER_ID_PREFIX="order#";  
    private static int counter = 1;  
    // pour compter les instances créées  
    // attributs d'instance  
    private Client client;  
    private Catalog catalog;  
    private String id;  
    public Order(Client client, Catalogue cata) {  
        this.client = client;  
        this.catalogue = cata;  
        this.id = Order.ORDER_ID_PREFIX + Order.counter++;  
    }  
}
```

Exemple d'utilisation

```
// utilisation :  
Order o1 = new Order(c,k);  
// c,k supposés définis et initialisés  
Order o2 = new Order(c,k);  
System.out.println("o1 -> "+o1.getId());  
System.out.println("o2 -> "+o2.getId());
```

o1 : Order	
client	c
catalog	k
id	"order#1"

o2 : Order	
client	c
catalog	k
id	"order#2"

Exemple de static

Documentation de la classe `java.lang.System`

```
public static final PrintStream out
```

The “standard” output stream. This stream is already open and ready to accept output data. Typically this stream corresponds to display output or another output destination specified by the host environment or user. The encoding used in the conversion from characters to bytes is equivalent to `Console.charset()` if the Console exists, `Charset.defaultCharset()` otherwise.

For simple stand-alone Java applications, a typical way to write a line of output data is:

```
System.out.println(data)
```

See the `println` methods in class `PrintStream`.

Méthode sur les doubles

On souhaite disposer d'une méthode pour calculer le sinus d'un nombre.

Signature ? `public double sin(double x) { ... }`

Une méthode se définit dans une classe : `Math`

Utilisation ? = appel/invocation $\Rightarrow$ il faut un objet

```
Math math1 = new Math();  
math1.sin(45);  
math1.sin(60);  
Math math2 = new Math();  
trigo2.sin(60);
```

Intérêt des objets `math1`, `math2` ?

Méthodes de classe

```
public class StaticExample {  
    public static void staticMethod() {  
        System.out.println("ceci est une méthode statique");  
    }  
}
```

Appel/Invocation : pas besoin d'instance (juste le nom de la classe)

`StaticExample.staticMethod()`

Important

Pas d'instance pour appeler la méthode donc `this` n'a aucun sens dans le corps d'une méthode statique.

Mot-clé `static` pour les méthodes

L'usage de `static` doit être limité et justifié à priori quasiment jamais car « pas objet ».

⇒ pratique réservée pour des méthodes dites « utilitaires » = fonctions

fonctions = méthodes de classe (`static`) dont le traitement ne dépend pas de l'état d'un objet

```
public class Math {  
    public static double sin(double x) { ... }  
    public static double sqrt(double x) { ... }  
    public static int max(int a, int b) { ... }  
}
```

Intérêt : éviter la création d'objet « jetable ».

→ on retrouve la notion de fonction comme en Python

Méthode main

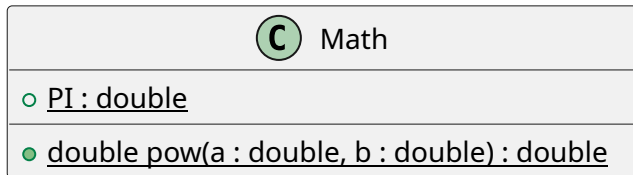
Cas particulier, la méthode `main`, sa signature doit rigoureusement être

```
public class SomeClass {  
  
    public static void main(String[] args) {  
        // ...  
    }  
}
```

Méthode appelée en utilisant la commande `java` avec comme argument `SomeClass` puis potentiellement d'autres arguments qui seront les valeurs de `args[]`.

```
java SomeClass arg0 arg1 ...
```


Méthode static dans les diagrammes



Règles

Membres (attributs ou méthode) de classe soulignés

Exemple : staticMember

Section 14

Surcharge de méthode/constructeurs

Méthodes ayant le même nom

Dans une classe, plusieurs méthodes peuvent avoir le même nom.

C'est ce qu'on appelle la surcharge de méthode.

Il est par contre nécessaire que la séquence dans l'ordre des types des arguments soit différente pour chaque méthode ayant le même nom.

La méthode est choisie par le compilateur de la façon suivante :

- Le nombre de paramètres doit correspondre
- Les affectations des paramètres doivent être valides

Exemple de surcharge de méthode

```
public class DataArtist {  
  
    public void draw(String s) {  
        /* ... */  
    }  
    public void draw(int i) {  
        /* ... */  
    }  
    public void draw(double f) {  
        /* ... */  
    }  
    public void draw(int i, double f) {  
        /* ... */  
    }  
}
```

Exemple de surcharge de méthode

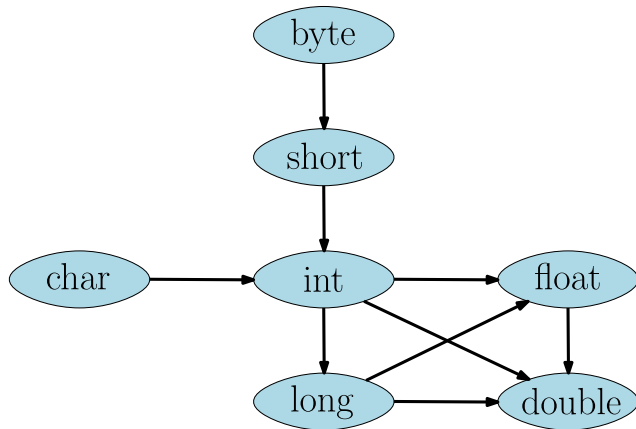
```
class Adder {  
    public static int add(int intVal1, int intVal2) {  
        System.out.println("integer");  
        return intVal1 + intVal2;  
    }  
  
    public static double add(double doubleVal1,  
                             double doubleVal2) {  
        System.out.println("double");  
        return doubleVal1 + doubleVal2;  
    }  
}
```

Exemple de surcharge de méthode

```
int intValue = 1;  
double doubleValue = 2.2;  
double result1 = Adder.add(doubleValue, doubleValue);  
// → double  
  
int result3 = Adder.add(intValue, intValue);  
// → int
```

Promotion pour les types primitifs

Si les types des arguments ne correspondent pas aux types des paramètres, les types des arguments sont promus en suivant les flèches:



Plusieurs constructeurs

C'est exactement les mêmes règles qui s'appliquent pour les constructeurs d'une classe.

```
public class Point{
    public int x, y;
    public Point(int x, int y) {
        this.x = x;
        this.y = y;
    }
    public Point() {
        this(0, 0);
        // this appelle le constructeur de la classe.
    }
    public Point(Point p) {
        this(p.x, p.y);
    }
}
```


Section 15

Commentaires et documentation

Commentaires inutiles

```
String get(String[] source, int index) {  
    // Teste si l'index est dans les limites du tableau.  
    if (index < 0 || index >= source.length)  
        return null;  
    return source[index];  
}
```

Si un commentaire semble nécessaire, le remplacer par une méthode :

```
boolean indexIsInBounds(String[] source, int index) {  
    return index >= 0 && index < source.length;  
}  
  
String get(String[] source, int index) {  
    if (!indexIsInBounds(source, index)) return null;  
    return source[index];  
}
```

Commentaires inutiles

Les commentaires se désynchronisent souvent du code :

```
/* doit toujours retourner true. */  
boolean isAvailable() {  
    return true;  
}
```

risque de devenir un jour :

```
/* doit toujours retourner true. */  
boolean isAvailable() {  
    return false;  
}
```

Commentaires inutiles = répétition

Commentaires inutiles

Des commentaires qui peuvent sembler utiles :

```
/* une méthode qui retourne que les carrés : */
List<Rectangle> get(List<Rectangle> list) {
    /* le résultat sera stocké dans cette liste : */
    List<Rectangle> list2 = new ArrayList<Rectangle>();
    for (Rectangle x : list)
        if (x.w == x.h /* un carré ? */)
            list2.add(x);
    return list2;
}

class Rectangle {
    public int w; /* largeur */
    public int h; /* hauteur */
}
```

Commentaires inutiles

On peut se passer de commentaire en rajoutant une méthode et en nommant correctement les éléments du code.

```
List<Rectangle> findSquares(List<Rectangle> rectangles) {  
    List<Rectangle> squares = new ArrayList<Rectangle>();  
    for (Rectangle rectangle : rectangles)  
        if (rectangle.isSquare())  
            squares.add(rectangle);  
    return squares;  
}  
  
class Rectangle {  
    private int width, height;  
    boolean isSquare() {  
        return width == height;  
    }  
}
```

Commentaires utiles

Des commentaires pour décrire les tâches à réaliser peuvent être utiles

```
void processElement(Stack<Formula> stack,  
                    String element) {  
    // TODO : prendre en compte les signes '-' et '/'  
    switch (element) {  
        case "+": processSum(stack); break;  
        case "*": processProduct(stack); break;  
        default : processInteger(stack, element);  
        break;  
    }  
}
```

Commentaires utiles

Documentation ou spécification du comportement d'une méthode :

```
/**
 * Returns true if this list contains the
 * specified element. More formally, returns
 * true if and only if this list contains
 * at least one element e such that
 * (o==null ? e==null : o.equals(e)).
 *
 * @param o element whose presence in this list is
 * to be tested
 * @return true if this list
 * contains the specified element
 */
```

```
public boolean contains(Object o) {
```

JavaDoc permet de générer automatiquement une documentation du code à partir de commentaires associés aux classes, méthodes, propriétés, ...

La documentation contient :

- Une description des membres : attributs et méthodes (publics et protégés) des classes
- Une description des classes, interfaces, ...
- Des liens permettant de naviguer entre les classes
- Des informations sur les implémentations et extensions

Un bloc de commentaire Java commençant par `/**` deviendra un bloc de commentaire Javadoc qui sera inclus dans la documentation du code source.

Tag	Description
@author	pour préciser l'auteur de la fonctionnalité
@deprecated	indique que l'attribut, la méthode ou la classe est dépréciée
@return	pour décrire la valeur de retour
{@code literal}	Formate <code>literal</code> en code
{@link reference}	permet de créer un lien

Pour générer la javadoc en IntelliJ

Tools → generate Javadoc

```
/**
 * The {@code Byte} class wraps a value of primitive
 * type {@code byte} in an object. An object of type
 * {@code Byte} contains a single field whose type is
 * {@code byte}.
 *
 * <p>In addition, this class provides several methods
 * for converting a {@code byte} to a {@code String}
 * and a {@code String} to a {@code byte}, as well as
 * other constants and methods useful when dealing
 * with a {@code byte}.
 *
 * @author Nakul Saraiya
 * @author Joseph D. Darcy
```

Section 16

Tests

Différents types de de tests

Règle

Un code non testé n'a aucune valeur.

Corollaire

Tout code doit être testé

Différents type de tests

- **Test unitaires** : Tester les différentes parties (méthodes, classes) d'un programme indépendamment les unes des autres.
- **Test d'intégration** : Tester des portions du programme combinant plusieurs classes.
- **Test de non régression** : Vérifier que le nouveau code ajouté ne corrompt pas les codes précédents : les tests précédemment réussis doivent encore l'être.

Tests unitaires

- Tester une unité de code : classe, méthodes, ...
- Vérifier un comportement :
 - ▶ cas normaux
 - ▶ cas limites
 - ▶ cas anormaux

Tests unitaires en java : JUnit avec assertJ

- Un framework de test unitaire pour Java
- S'appuie sur des **assertions**

Utilisation de JUnit (1/3)

Règles

- 1 classe de test = un ensemble de méthodes de test
- 1 classe de test par classe à tester
- 1 méthode de test = 1 cas de test
- 1 cas de test = (description, données d'entrée, résultat attendu)

Structure d'une méthode de test de base

- méthode d'instance publique
- annotée avec `@Test` (à mettre avant la déclaration de la méthode)
- ne prend aucun paramètre
- ne renvoie rien
- lève une `assertionError` en cas de test échoué

Utilisation de JUnit (2/3)

Conventions de nommage

- nom d'une classe de test : *NameTestedClassTest*
- nom d'une méthode de test : *testNameTestedMethod*

Structure d'un projet avec tests

les tests sont séparés du code de production : répertoire `main` pour le code de production et répertoire `test` pour le code de tests.

⇒ nécessaire de séparer les test du code de production car :

- on ne donne pas l'accès au code de test au client par exemple
- les tests ont un rôle spécifique différent du code de production

Utilisation de JUnit (3/3)

```
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.*;

public class NameTestedClassTest {
    @Test
    void testNameTestedMethod(){
        // code containing assertions to test nameTestedMethod
    }
}
```


Assertions assertJ (1/2)

- `assertThat(condition).isTrue()` : vérifie que `condition` est vraie.
- `assertThat(condition).isFalse()` : vérifie que `condition` est faux.
- `assertThat(actual).isEqualTo(expected)` : vérifie que `expected` est égal à `actual` égal : `equals` pour les objets et `==` pour les types primitifs.
- `assertThat(actual).isCloseTo(expected, within(delta))` : vérifie que $|expected - actual| \leq delta$
- `assertThat(object).isNull()` : vérifie que la référence est null

Assertions assertJ (2/2)

- `assertThat(object).isNotNull()` : vérifie que la référence **n'est pas** null
- `assertThat(actual).isSameAs(expected)` : vérifie que les deux objets sont les mêmes (même référence).
- `assertThat(list).containsExactly(e1, e2, e3)` : vérifie que la liste `list` contient uniquement les éléments `e1`, `e2` et `e3` dans cet ordre.
- `assertThat(list1).containsExactlyElementsOf(list2)` : vérifie que les deux listes `list1` et `list2` contiennent les mêmes éléments dans le même ordre.
- `fail(message)` : échoue toujours en affichant `message`

Message

Il est possible de provoquer l'affichage d'un message lors d'un test faux en appelant `as(message)` sur le retour d'un `assertThat`.

Exemple de classe à tester : RationalNumber

```
public class RationalNumber {  
    public final int numerator;  
    public final int denominator;  
    public RationalNumber(int numerator, int denominator) {  
        int gcd = gcd(numerator, denominator);  
        this.numerator = numerator / gcd;  
        this.denominator = denominator / gcd;  
    }  
    public RationalNumber add(RationalNumber val) {  
        int numerator = (this.numerator * val.denominator)  
            + (this.denominator * val.numerator);  
        int denominator = this.denominator * val.denominator;  
        return new RationalNumber(numerator, denominator);  
    }  
}
```

Exemple de classe de test

```
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.*;
public class RationalNumberTest {
    @Test
    void testAdd(){
        RationalNumber one = new RationalNumber(1, 1);
        RationalNumber onePlusOne = one.add(one);
        assertThat(onePlusOne.numerator)
            .as("Numerator of one plus one is two.")
            .isEqualTo(2);
        assertThat(onePlusOne.denominator)
            .as("Denominator of one plus one is one.")
            .isEqualTo(1);
    }
}
```

Exemple de classe à tester : Box

```
public class Box {  
    /**  
     * Create a box with the specified weight  
     * @param weight the weight of the created box  
     */  
    public Box(int weight) {  
        this.weight = weight;  
    }  
    /** weight of the box */  
    private int weight;  
    /** @return this box's weight */  
    public int getWeight() {  
        return this.weight;  
    }  
}
```

Exemple de classe de test : TestBox

```
import static org.assertj.core.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class BoxTest {
    @Test
    public void testCreationIsOk() {
        Box someBox = new Box(10);
        assertThat(someBox.getWeight()).isEqualTo(10);
    }
    // ...
}
```

Exemple de classe à tester (1/2) : Emails

```
public class Emails {  
    private String text;  
    public Emails(String text) { this.text = text; }  
    public List<String> userNames() {  
        int pos = 0;  
        List<String> users = new ArrayList<String>();  
        for(;;) {  
            int atIndex = text.indexOf('@', pos);  
            if (atIndex == -1) break;  
            String userName = userName(atIndex);  
            if (userName.length() > 0) users.add(userName);  
            pos = atIndex + 1;  
        }  
        return users;  
    }  
}
```

Exemple de classe à tester (2/2) : Emails

```
private String userName(int atIndex) {  
    int back = atIndex - 1;  
    while (back >= 0 &&  
           (Character.isLetterOrDigit(text.charAt(back))  
            || text.charAt(back) == '.')) {  
        back--;  
    }  
    return text.substring(back + 1, atIndex);  
}
```


Exemple de classe de test (1/3)

```
import org.junit.jupiter.api.Test;
import static org.assertj.core.api.Assertions.*;

public class EmailsTest{
    @Test
    public void testUsersBasic() {
        Emails emails =
            new Emails("foo bart@cs.edu xyz marge@ms.com baz");
        assertThat(emails.getUserNames())
            .containsExactly("bart", "marge");
    }
}
```

Exemple de classe de test (2/3)

@Test

```
public void testUsersChars() {  
    Emails emails =  
        new Emails("fo f.ast@cs.edu bar&a.2.c@ms.com ");  
    assertThat(emails.getUserNames())  
        .containsExactly("f.ast", "a.2.c");  
}
```

@Test

```
public void testUsersChars() {  
    Emails emails =  
        new Emails("fo f.ast@cs.edu bar&a.2.c@ms.com ");  
    assertThat(emails.getUserNames())  
        .containsExactly("f.ast", "a.2.c");  
}
```

Exemple de classe de test (3/3)

@Test

```
public void testUsersHard() {  
    Emails emails = new Emails("x y@cs 3@ @z@");  
    assertThat(emails.getUserNames())  
        .isNotEmpty();  
    .containsExactly("y", "3", "z");  
    emails = new Emails("no emails here!");  
    assertThat(emails.getUserNames()).isEmpty();  
    emails = new Emails("@@@");  
    assertThat(emails.getUserNames()).isEmpty();  
    emails = new Emails("");  
    assertThat(emails.getUserNames()).isEmpty();  
}
```

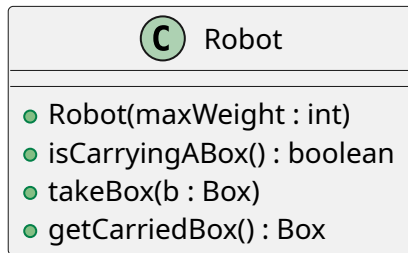
Méthodes de test

- préfixée de l'annotation `@Test`
- signature de la forme `public void testMethod()`
- le corps de la méthode contient des assertions : `assertThat`
le test est réussi si toutes les assertions sont vérifiées
- plusieurs méthodes de tests peuvent être nécessaires pour tester la correction d'une méthode
- principes :
 - 1 créer la situation initiale et vérifier les « préconditions »
 - 2 appeler la méthode testée
 - 3 à l'aide d'assertions, vérifier les « postconditions » = situation attendue après l'exécution de la méthode

Méthodologie pour créer des tests

Un robot peut porter une caisse d'un poids maximal défini à la construction du robot. Initialement un robot ne porte pas de caisse. S'il porte déjà une caisse il ne peut en prendre une autre.

Classe Robot



Méthodologie pour créer des tests

Un robot peut porter une caisse d'un poids maximal défini à la construction du robot. **Initialement un robot ne porte pas de caisse.** S'il porte déjà une caisse il ne peut en prendre une autre.

```
import static org.assertj.core.api.Assertions.*;
import org.junit.jupiter.api.Test;

public class RobotTest {
    public void notCarryingABoxWhenCreated() {
        Robot robbie = new Robot(15);
        // no carried box ?
        assertThat(robbie.isCarryingABox()).assertFalse();
    }
}
```

Méthodologie pour créer des tests

Un robot **peut porter une caisse d'un poids maximal défini à la construction du robot**. Initialement un robot ne porte pas de caisse. S'il porte déjà une caisse il ne peut en prendre une autre.

```
public class RobotTest {
    @Test
    public void robotCanTakeLightBox() {
        // initial configuration : a robot and a box
        Robot robbie = new Robot(15);
        Box b = new Box(10);
        // precondition : robot ne porte rien
        assertThat(robbie.isCarryingABox()).assertFalse();
        // execution of the tested method
        robbie.takeBox(b);
        // postcondition : the carried box is ?
    }
}
```

Méthodologie pour créer des tests

Un robot **peut porter une caisse d'un poids maximal défini à la construction du robot**. Initialement un robot ne porte pas de caisse. S'il porte déjà une caisse il ne peut en prendre une autre.

```
public class RobotTest {  
    @Test  
    public void robotCannotTakeTooHeavyBox() {  
        Robot robbie = new Robot(15);  
        Box b = new Box(20);  
        // precondition : robot does not carry a box  
        assertThat(robbie.isCarryingABox()).assertFalse();  
        // execution of the tested method  
        robbie.takeBox(b);  
        // postcondition : the carried box is b  
        assertThat(robbie.isCarryingABox()).assertFalse();  
    }  
}
```


Méthodologie pour créer des tests

Un robot peut porter une caisse d'un poids maximal défini à la construction du robot. Initialement un robot ne porte pas de caisse. **S'il porte déjà une caisse il ne peut en prendre une autre.**

@Test

```
public void robotCanTakeOnlyOneBox() {  
    Robot robbie = new Robot(15);  
    Box b1 = new Box(10); Box b2 = new Box(4);  
    robbie.takeBox(b1);  
    // precondition : the carried box is b  
    assertThat(robbie.getCarriedBox()).isSameAs(b1);  
    // execution of the tested method  
    robbie.takeBox(b2);  
    // postcondition: the carried box is b1 and not b2  
    assertThat(robbie.getCarriedBox()).isNotSameAs(b2);  
}
```

Méthodologie pour créer des tests

Travailler une méthode à la fois :

- ➊ Définir la signature de la méthode,
- ➋ Écrire la javadoc de la méthode,
- ➌ Écrire les tests qui permettront de contrôler que le code écrit pour la méthode est correct = répond au cahier des charges
- ➍ Coder la méthode,
- ➎ Exécuter les tests définis à l'étape 3, en vérifiant la non régression,
- ➏ Si les tests sont réussis passer à la méthode suivante (étape 1) sinon recommencer à l'étape 4.

Il ne s'agit pas de travailler plus, mais d'être plus efficace.

Test unitaires (à retenir)

- Il est essentiel de tester son code.
- Écrire au moins une méthode de test pour chaque méthode du code de production.
- Il est important de tester tous les types de cas :
 - ▶ cas normaux (utilisation naturelle de la méthode sur une donnée naturelle)
 - ▶ cas limites (utilisation de la méthode sur une donnée "étrange")
 - ▶ cas anormaux (vérification que les erreurs d'utilisation, c'est-à-dire que les cas d'erreurs sont bien pris en compte et gérés)

La suite : le TDD (Test Driven Development)

- Écrire un test qui échoue avant de pouvoir écrire du code
- Écrire une assertion à la fois qui fait échouer un test
- Écrire le minimum de code pour que l'assertion soit satisfaite

Section 17

Les énumérations

Énumérations en Java (1/2)

En programmation, une *énumération* est un type spécial qui permet de définir des variables pouvant prendre des valeurs parmi un ensemble prédéfini de constantes.

Il est possible de définir des énumérations en Java grâce au mot-clé `enum` :

```
enum Suit{  
    SPADES,  
    HEARTS,  
    DIAMONDS,  
    CLUBS;  
}
```

Énumérations en Java (2/2)

```
enum Suit {  
    SPADES, HEARTS, DIAMONDS, CLUBS;  
}
```

Une énumération est une classe avec des éléments prédéfinis et statiques.

On peut donc tester directement l'égalité avec l'opérateur égal car il n'y qu'un objet et donc qu'une référence qui correspond à chaque valeur possible.

```
Suit suit = Suit.SPADES;  
/* ... */  
if (suit == Suit.SPADES)  
{  
    /* ..... */  
}
```

Définition d'attributs, de méthodes et d'un constructeur

```
public enum Suit {  
    SPADES("Pique", "Pi"), HEARTS("Cœur", "Co"),  
    DIAMONDS("Carreau", "Ca"), CLUBS("Trèfle", "Tr");  
  
    private final String frenchName;  
    private final String frenchSymbol;  
  
    Suit(String name, String symbol) {  
        this.frenchName = name;  
        this.frenchSymbol = symbol;  
    }  
  
    public String frenchName() { return frenchName; }  
    public String frenchSymbol() { return frenchSymbol; }  
}
```

Toutes les énumération de java étendent la classe Enum.

Quelques méthodes utiles :

- `String name()` : retourne le nom de la constante tel que déclaré dans l'enum.
- `int ordinal()` : retourne la position de la constante dans la déclaration de l'enum (commençant par 0).

Tout enum définit aussi une méthode statique `values()` renvoyant un tableau contenant les constantes dans l'ordre de leurs déclarations.

Exemple d'utilisation d'énumération (1/2)

```
public static void main(String[] args) {  
    Suit[] values = Suit.values();  
    for (Suit suit : values)  
        System.out.printf("Le symbole de %s est %s.\n",  
                           suit.frenchName(), suit.frenchSymbol());  
}
```

Affichage :

Le symbole de Pique est Pi.

Le symbole de Cœur est Co.

Le symbole de Carreau est Ca.

Le symbole de Trèfle est Tr.

Exemple d'utilisation d'énumération (2/2)

```
public static void main(String[] args) {  
    for (Suit suit : Suit.values())  
        System.out.printf("The position of %s is %d.\n",  
                           suit.name(), suit.ordinal());  
}
```

Affichage :

The position of SPADES is 0

The position of HEARTS is 1

The position of DIAMONDS is 2

The position of CLUBS is 3

Mot-clé switch et enum

Supposons qu'on est une enum pour les jour de la semaine :

```
public enum Day
{
    SUNDAY,
    MONDAY,
    TUESDAY,
    WEDNESDAY,
    THURSDAY,
    FRIDAY,
    SATURDAY;
}
```

Mot-clé switch et enum

On peut utiliser le mot-clé switch pour traiter des cas différents :

```
public enum Day{  
    public void dayIsLike() {  
        switch (this) {  
            case MONDAY:  
                System.out.println("Mondays are bad."); break;  
            case FRIDAY:  
                System.out.println("Fridays are better."); break;  
            case SATURDAY:  
            case SUNDAY:  
                System.out.println("Weekends are best."); break;  
            default:  
                System.out.println("Midweek days are so-so.");  
                break;  
        }  
    }  
}
```

Mot-clé switch et enum (nouvelle version)

Depuis java 14, on peut écrire de façon équivalente :

```
public enum Day{  
    public void dayIsLike() {  
        switch (this) {  
            case MONDAY  
                -> System.out.println("Mondays are bad.");  
            case FRIDAY  
                -> System.out.println("Fridays are better.");  
            case SATURDAY, SUNDAY  
                -> System.out.println("Weekends are best.");  
            default  
                -> System.out.println("Midweek days are so-so.");  
        }  
    }  
}
```

Section 18

Interfaces : type commun à plusieurs classes

Problèmes de recyclage

- Papiers, bouteilles, piles électriques, cageots, ... sont des objets **différents** :
⇒ *déchirer* du papier, *remplir* un bouteille
- Mais ces objets partagent tous la propriété d'être **recyclable**
⇒ tous peuvent être *recyclés* (même si le processus peut varier)
On peut *recycler* tous les objets d'une poubelle.

En programmation objet

- Paper, Bottle, Battery, Crate, ... sont des classes d'objets **différentes**
- Toutes ont une méthode `recycle()` avec une implémentation **adaptée** à chacune des classes.

Comment recycler tous les objets d'une poubelles ?

Avec le code suivant ?

```
T[] trashCan = // ...  
for(T trashItem : trashCan){  
    trashItem.recycle();  
}
```

Problème

Comment définir le tableau trashCan ?

Quel est le type T de ces éléments ?

Remarques

- les objets de type T implémentent la méthode recycle
- Trashcan doit pouvoir contenir des objets provenant de classes différentes

Homogénéité des collections en Java

Les collections (`List`, tableaux, ...) sont **homogènes** : tous les éléments contenus dans la collection sont du même type.

Raisons :

- Parce que cela aide à la lisibilité du code (on comprend ce que contient la collection) comme par exemple lorsqu'on définit `List<Rectangle> rectangles` qui est une liste de rectangles.
- Cela évite des erreurs à l'exécution arrivant avec des langages non-statiquement typés comme Python lorsqu'on appelle une méthode sur un objet sans savoir si l'objet a une implémentation pour cette méthode.

Question

Les éléments de `TrashCan` doivent avoir un type, lequel ?

La solution objet

Conserver les classes différentes et créer un type commun.

- On doit conserver les classes différenciées : Paper, Bottle, ...
- On doit traiter les objets sans les différencier par leur classe.
- Il faut pouvoir considérer les instances des classes comme des objets de type implémente la méthode `recycle`.

Projection

On va projeter les objets sur un type commun qui ne gardera que la partie commune des fonctionnalités. On ne considère qu'une facette de l'objet.

Dans notre cas, la seule méthode qu'on pourra appeler sur les éléments de `trashCan` sera `recycle` même si les objets disposait à la base d'autres méthodes (par exemple `burn()` pour les Paper).

Solution java : interface

- En java, une interface est un ensemble de déclaration de **signatures** de méthodes (type de retour, nom de la méthode et type des arguments) et définit un type.
- Une classe peut **implémenter** une interface et doit dans ce cas définir le comportement (code) pour **chacune** des méthodes de l'interface.
- les instances d'une classe pourront être vues comme étant du **type de l'interface**, manipulées comme telles et avoir leur référence stockée dans une variable du type de l'interface.
- Un référence du type d'une interface accepte uniquement les appels de méthodes définies dans l'interface.

Exemple d'utilisation (1/2)

```
public interface Recyclable{
    void recycle(); // public par défaut
}

public class Paper implements Recyclable{
    // ...
    public void recycle(){
        System.out.println("Recycling paper");
    }
}

public class Bottle implements Recyclable{
    // ...
    public void recycle(){
        System.out.println("Recycling bottle");
    }
}
```

Exemple d'utilisation (2/2)

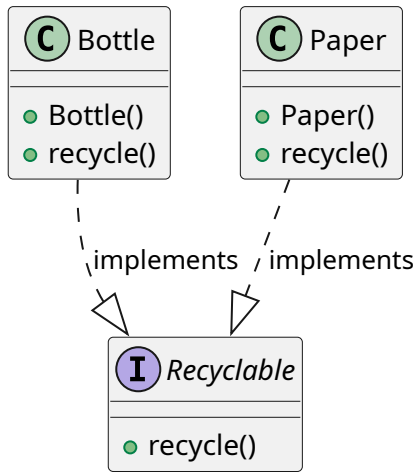
```
Recyclable[] trashcan = new Recyclable[2];
```

```
trashcan[0] = new Paper();
```

```
trashcan[1] = new Bottle();
```

```
for(Recyclable trashItem : trashCan){  
    trashItem.recycle();  
}
```

La solution en diagramme



Implémentation d'une interface

Implémenter une interface

Une classe *implémente* une interface :

- si toute méthode déclarée dans l'interface est déclarée et programmée (codée) dans la classe.
 - si la classe indique à sa déclaration qu'elle implémente l'interface.
-
- Une classe peut implémenter des méthodes en plus de l'interface
 - Une classe peut implémenter plusieurs interfaces (séparées par des virgules à la déclaration)
 - Pour implémenter une méthode d'une interface, la méthode de la classe doit avoir :
 - ▶ Le même nom
 - ▶ Les mêmes paramètres (les mêmes types) dans le même ordre
 - ▶ Le même type de retour

Interface en Java

```
public interface IntegerIterator {  
    boolean hasNext();  
    int next();  
}
```

- Déclaration dans un fichier de même nom comme pour les classes.
- Liste de signatures de méthodes
- Pas d'instructions
- Pas d'attributs d'instances
- Pas de visibilité spécifiée pour les méthodes car tout est public par défaut

Section 19

Interfaces : passer d'une implémentation à une autre

Interface

On vient de voir qu'une interface permet de définir un type commun correspondant à des instances de classes différentes.

⇒ cela permet de regrouper des objets partageant un même contrat (ensemble de méthode) dans une même collection (liste, tableau, ...) et d'utiliser ce service.

Une autre utilisation importante des interfaces est de séparer un contrat (déclaration des méthodes) de son implémentation (codes des méthodes) et donc de permettre de changer très facilement d'implémentation.

Inversion des dépendances (Dependency inversion principle)

Il faut dépendre des abstractions (les interfaces) et pas des implémentations (les classes).

Exemple : interface pour l'affichage

Supposons que des classes implémentent un service de façons différentes :

```
class SimplePrinter {  
    public void print(String document){  
        System.out.println(document);  
    }  
}  
  
class BracePrinter {  
    public void print(String document){  
        System.out.println("{ " + document + " }");  
    }  
}
```

Les instances de ces deux classes possèdent une méthode print avec la même signature (types des arguments et du retour)

Code facilement modifiable

Nous souhaiterions pouvoir facilement passer du code suivant :

```
Printer printer = new SimplePrinter();  
printer.print("truc"); // → truc
```

au code suivant :

```
Printer printer = new BracePrinter();  
printer.print("truc"); // → {truc}
```

Il nous faudrait définir un type `Printer` qui oblige la variable à contenir des références vers des objets qui implémentent la méthode `print`.

⇒ définition d'une interface `Printer`.

Abstraction

On peut vouloir traiter les objets en utilisant les services qu'ils partagent :

```
for (int index = 0; index < printers.length index++)  
    printers[index].print(document);
```

On peut aussi vouloir écrire un programme en supposant que les objets manipulés implémentent certains services (comme le fait de pouvoir les comparer) :

```
boolean isSorted(Comparable[] array) {  
    for (int i = 0; i < array.length - 1; i++)  
        if (array[i].compareTo(array[i+1]) > 0)  
            return false;  
    return true  
}
```

Description d'une interface

Description d'une interface en Java :

```
public interface Printer{  
    /**  
     * Print the {@code String} {@code document}  
     * @param document the string to be printed  
     */  
    void print(String document);  
}
```

Une interface :

- définit la **signature** (types des arguments et du retour) d'une ou plusieurs méthodes
- est un contrat
- définit un type : référence vers un objet implémentant les méthodes de l'interface

Implémentation d'une interface

Le mot-clé `implements` permet d'indiquer qu'une classe implémente un interface :

```
class SimplePrinter implements Printer {  
    void print(String document){  
        System.out.println(document);  
    }  
}  
  
class BracePrinter implements Printer {  
    void print(String document){  
        System.out.println("{ " + document + " }");  
    }  
}
```

Java vérifie à la compilation que toutes les méthodes de l'interface sont implémentées.

Références et interfaces

Déclaration d'une variable de type référence vers une instance d'une classe qui implémente l'interface `Printer` :

```
Printer printer;
```

Important

Une interface ne définit pas de constructeurs.

Interdit : ~~`printer = new Printer();`~~

Compatibilité classe et instance

Pour affecter une référence à une variable d'un type défini par une interface, on doit instancier une classe implémentant l'interface.

Références et interfaces

Il est donc possible d'instancier une classe implémentant l'interface,

```
SimplePrinter simplePrinter = new SimplePrinter();
```

puis de stocker la référence de l'objet dans une variable de type de l'interface :

```
Printer printer1 = simplePrinter;
```

On parle alors d'**upcasting** (transtypage vers le haut). On peut aussi directement mettre un tel objet sans passer par une variable intermédiaire :

```
Printer printer2 = new BracePrinter();
```

Par contre, cela ne fonctionne pas dans le cas où la classe n'implémente pas l'interface :

```
Printer printer3 = new String("Hello!"); //interdit
```

Références et interfaces

```
class Utils {  
    static void printString(Printer[] printers,  
                            String doc) {  
        for (int i = 0; i < printers.length; i++)  
            printers[i].print(doc);  
    }  
  
    static void printArray(String[] array,  
                           Printer printer) {  
        for (int i = 0; i < array.length; i++)  
            printer.print(array[i]);  
    }  
}
```

Polymorphisme

L'existence des méthodes est vérifiée à la compilation.

Le code suivant ne compilera pas car l'interface `Printer` n'a pas de méthode `println` :

```
Printer printer = new SimplePrinter();  
printer.println("Hello!"); // Impossible
```

Définition polymorphisme

Du grec ancien *polús* (plusieurs) et *morphê* (forme)

En programmation orientée objet, le polymorphisme permet à une classe d'être vue et traitée comme étant du type d'une des interfaces qu'elle implémente ou bien son propre type.

Polymorphisme

Le choix de la méthode à exécuter ne peut être fait qu'à l'exécution :

```
Printer[] printers = new Printer[2];  
printers[0] = new SimplePrinter();  
printers[1] = new BracePrinter();  
Random random = new Random(); // générateur aléatoire  
int index = random.nextInt(2); // 0 et 1  
printers[index].print("mon message");
```

L'affichage dépend du tirage aléatoire pour index :

- index=0 → méthode de la classe SimplePrinter → mon message
- index=1 → méthode de la classe BracePrinter → {mon message}

Implémentations multiples

Il peut être utile d'avoir une classe implémentant plusieurs interfaces.

Par exemple, une classe `Modem` pourrait implémenter les deux interfaces suivantes :

```
public interface ConnexionManager{
    void dial(string phoneNumber);
    void hangUp();
}

public interface TransmissionManager{
    void send(char c);
    char receive();
}

public class Modem
    implements ConnexionManager, TransmissionManager{
}
```

Implémentations multiples

Une classe Printable avec une méthode print qui permet d'afficher l'objet.

```
public interface Printable {  
    void print();  
}
```

Une classe Stack avec deux méthodes :

- push qui permet d'empiler un entier.
- pop dépile et retourne l'entier en haut de la pile.

```
public interface Stack {  
    void push(int value);  
    int pop();  
}
```

Implémentations multiples

Implémentation des deux interfaces précédentes :

```
public class PrintableArrayStack
    implements Stack, Printable {
    private int[] array; private int size;
    public PrintableArrayStack(int capacity) {
        array = new int[capacity]; size = 0;
    }
    public void push(int v) { array[size] = v; size++; }
    public int pop() { size--; return array[size]; }
    public void print() {
        for (int i = 0; i < size; i++)
            System.out.print(array[i]+" ");
        System.out.println();
    }
}
```

Implémentations multiples

Implémentation d'une des deux interfaces :

```
public class PrintableString implements Printable {  
    private String string;  
    public PrintableString(String string) {  
        this.string = string;  
    }  
    public void print() {  
        System.out.println(string);  
    }  
}
```


Implémentations multiples

Exemple d'utilisation des classes précédentes :

```
Printable[] printables = new Printable[3];
printables[0] = new PrintableString("bonjour");
PrintableArrayStack stack = new PrintableArrayStack(10);
printables[1] = stack;
printables[2] = new PrintableString("salut");
stack.push(10);
stack.push(30); System.out.println(stack.pop());
stack.push(12);
for (int i = 0; i < printables.length; i++)
    printables[i].print();
```

Qu'écrit ce programme sur la sortie standard ?

Vérification des types

Vérification des types à la compilation :

```
Stack[] arrayStack = new Stack[2];  
arrayStack[0] = new PrintableStack();  
arrayStack[1] = new PrintableString("t"); // Erreur !
```

PrintableString n'implémente pas Stack.

```
Stack stack = new PrintableStack();  
Printable printable = stack; // Erreur !
```

Le type Stack n'est pas compatible avec le type Printable.

Résumé des interfaces

- Une interface est un ensemble de signatures de méthodes.
- Une classe peut implémenter une ou plusieurs interfaces : elle doit préciser le comportement de chacune des méthodes de chaque interface qu'elle implémente.
- Il est possible de déclarer une variable pouvant contenir des références vers des instances de classes qui implémentent l'interface.
- Java vérifie à la compilation que toutes les affectations et les appels de méthodes sont corrects.
- Le choix du code qui va être exécuté est décidé à l'exécution (en fonction de l'instance pointée par la référence).

Section 20

Gestion de version

Principe

- Le code d'un projet est stocké dans un serveur.
- Les développeurs soumettent des modifications avec des commentaires à chaque fois.
- Le serveur conserve l'historique des mises à jour

Pourquoi la gestion de version ?

- Pour travailler de manière harmonieuse en équipe sans se marcher dessus
- Pour revenir en arrière en cas de problèmes (chaque modification correspond à une version du code sur laquelle il est possible de revenir)
- Possibilité de faire valider le code (via des tests) par le serveur et de rendre le déploiement automatique

- Logiciel de gestion de version le plus populaire
- Serveur gratuit : github
- Version libre de logiciel serveur : gitlab
- Gestion de version décentralisée : la gestion de version se fait aussi en local

Utilisation de git

- Via l'IDE : VCS (Version Control Systems) dans le menu d'IntelliJ
- En ligne de commande : commande git

Git (source: Randall Munroe (xkcd))



Fonctionnement basique de git

- À la première utilisation, on crée une copie locale du dépôt git (`clone` ou `init`)
- À chaque commit :
 - ▶ on récupère la version courante du dépôt sur le serveur (`pull`)
 - ▶ On ajoute les fichiers à modifier (`add`)
 - ▶ On finalise le commit en donnant un message résumant les modifications (`commit`)
- Après un ou plusieurs commits, on met à jour la version distante avec nos modifications (`push`)

Exemples de commandes git

```
git clone adresse_projet
```

⇒ Clone un projet en local depuis un serveur

```
git add nom_de_fichier
```

⇒ Ajoute un fichier à la prochaine mise à jour.

```
git commit -m"commentaire"
```

⇒ Fait une mise à jour en local

```
git pull
```

⇒ Récupère les mises à jour du serveur en local

```
git push
```

→ Pousse les mises à jour locales sur le serveur

- Importance d'avoir une gestion de version de son code
- Commandes de base : `clone`, `add`, `commit`, `push`, `pull`, ...

Git avancé (dans vos prochains cours)

- Notion de branche : on “fork” le dépôt pour créer une branche séparée sur laquelle on va travailler et on demande ensuite de l'intégrer dans la branche principale une fois les modifications effectuées.
- CI/CD (Intégration Continue/Déploiement (ou livraison) continue) : automatisation des tests et du déploiement (par exemple sur les serveurs qui utilise le code) ou de la livraison (publication du code compilé sur un serveur) à chaque push sur la branche principale.

Section 21

Structure d'un projet et paquetages

Structure d'un projet

En Java, un projet peut être découpé en paquetages (package).

Les paquetages permettent de :

- associer des classes afin de mieux organiser le code
- de créer des parties indépendantes réutilisables
- d'avoir plusieurs classes qui possèdent le même nom (du moment qu'elles ne sont pas dans le même paquetage)

Un paquetage (package) :

- est une collection de classes
- peut contenir des sous-paquetages

Lors de l'exécution...

Java utilise l'arborescence de fichier pour retrouver les fichiers `.class`

- Une classe (ou une interface) correspond à un fichier `.class`
- Un dossier correspond à un paquetage

Les `.class` du paquetage `com.univ_amu` doivent :

- être dans le sous-dossier `com/univ_amu`
- le dossier `com` doit être à la racine d'un des dossiers du `ClassPath`

Le `ClassPath` inclut :

- le répertoire courant
- les dossiers de la variable d'environnement `CLASSPATH`
- des archives `JAR`
- des dossiers précisés sur la ligne de commande de java (`-classpath path`)

Lors de la compilation... (1/2)

Le mot-clé `package` permet de préciser le packaging des classes ou interfaces définies dans le fichier :

```
package com.univ_amu;  
public class MyClass { /* ... */ }
```

Java utilise l'arborescence pour retrouver le code des classes ou interfaces :

- Une classe (ou une interface) `MyClass` est cherchée dans le fichier `MyClass.java`
- Le fichier `MyClass.java` est cherché dans le dossier associé au packaging de `MyClass`

Lors de la compilation... (2/2)

```
package com.univ_amu;  
public class MyClass { /* ... */ }
```

Dans l'exemple précédent, il est donc conseillé que le fichier :

- se nomme `MyClass.java`
- se trouve dans le dossier `com/univ_amu` (Par défaut, la compilation crée `MyClass.class` dans `com/univ_amu`)

Utilisation d'une classe à partir d'un autre paquetage

Accessibilité :

- Seules les classes publiques sont utilisable à partir d'un autre paquetage
- Un fichier ne peut contenir qu'une seule classe publique

On peut désigner une classe qui se trouve dans un autre paquetage :

```
package com.my_project;  
    public class Main {  
        public static void main(String[] args) {  
            com.univ_amu.MyClass myClass =  
                new com.univ_amu.MyClass();  
        }  
    }  
}
```


Importer une classe

Vous pouvez également importer une classe :

```
package com.my_project;
import com.univ_amu.MyClass;

public class Main {
    public static void main(String[] args) {
        MyClass myClass = new MyClass();
    }
}
```

Deux classes de deux paquetages différents peuvent avoir le même nom :

- Exemple : `java.util.List` et `java.awt.List`
- Attention de choisir le bon import

Utiliser une classe d'un autre paquetage sans import

En fait, il est possible d'utiliser des classes d'un autre paquetage sans import.

Il suffit de mettre le chemin complet à chaque utilisation de la classe.

```
public class AppPackage {  
    static public void main(String[] args){  
        java.util.List<Integer> list1 =  
            new java.util.ArrayList<>();  
        java.awt.List list2 =  
            new java.awt.List();  
    }  
}
```

⇒ On peut même utiliser dans ce cas deux classes ayant le même nom mais des paquetages différents.

Importer un paquetage

Vous pouvez également importer toutes les classes d'un paquetage :

```
package com.my_project;  
import com.univ_amu.*;  
  
public class Main {  
    public static void main(String[] args) {  
        MyClass myClass = new MyClass();  
    }  
}
```

Remarques :

- Les classes des sous-paquetages ne sont pas importées
- Le paquetage `java.lang` est importé par défaut

Importer des membres statiques

Depuis Java 5, il est possible d'importer directement des méthodes ou attributs de classes (static).

La syntaxe est la suivante :

```
import static my_package.my_class.myStaticMember;
```

Exemple :

```
import static java.lang.Math.PI;  
import static java.lang.Math.pow;
```

Exemple sans import statique

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello World!");  
        System.out.println(  
            "A circle with a diameter of 5 cm has");  
        System.out.println("a circumference of "  
            + (Math.PI * 5) + " cm");  
        System.out.println("and an area of "  
            + (Math.PI * Math.pow(2.5, 2))  
            + " sq. cm");  
    }  
}
```

Exemple d'import statique

```
import static java.lang.Math.PI;
import static java.lang.Math.pow;
import static java.lang.System.out;

public class HelloWorld {
    public static void main(String[] args) {
        out.println("Hello World!");
        out.println(
            "A circle with a diameter of 5 cm has");
        out.println("a circumference of "
            + (PI * 5) + " cm");
        out.println("and an area of "
            + (PI * pow(2.5, 2)) + " sq. cm");
    }
}
```

Un exemple

Le fichier `com/univ_amu/HelloWorld.java` :

```
package com.univ_amu;
    public class HelloWorld {
        public static void main(String[] arg) {
            System.out.println("Hello world ! ");
        }
    }
```

```
$ javac com/univ_amu/HelloWorld.java
```

```
$ ls com/univ_amu
```

```
HelloWorld.java HelloWorld.class
```

```
$ java com.univ_amu.HelloWorld
```

```
Hello world !
```

Quelques remarques

Nommage des paquetages :

- Les noms de paquetages sont écrits en minuscules
- Pour éviter les collisions, on utilise le nom du domaine à l'envers $\Rightarrow$ `com.google.gson`, `com.oracle.jdbc`
- Si le nom n'est pas valide, on utilise des *underscores* : $\Rightarrow$ `com.univ_amu`

Fichier JAR (Java Archive) :

- est une archive ZIP pour distribuer un ensemble de classes Java
- contient un *manifest* (qui peut préciser la classe qui contient le main)
- peut également faire partie du ClassPath
- peut être généré en ligne de commande (`jar`) ou avec un IDE

Section 22

Bonnes pratiques de programmation

Pièges à éviter pour le nommage de variables/attributs

Vous ne devez pas avoir des variables avec des :

- noms en une lettre (même pour les indices) : `i`, `j`, ...
- noms numérotés : `a1`, `a2`, `a3`, ...
- abréviations ayant plusieurs interprétations : `rec`, `res`, ...
- noms ne donnant pas le sens précis : `temporary`, `result`, ...
- des noms trompeurs : par exemple un `accountList` doit être une `List` (et pas un `array` ou un autre type)
- types de l'objet au singulier pour une collection d'objet : une liste de personnes doit s'appeler `persons` et non `person`.
- noms imprononçables : `genymdhms`, ...

Règles de nommage (1/2)

En anglais

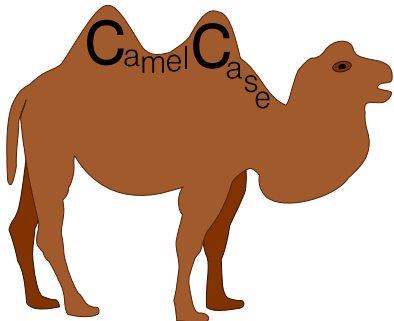
- Le **Java** et la quasi-totalité des langages de programmation utilise l'anglais (dans les mot-clés et les bibliothèques standards).
⇒ On doit programmer en anglais pour avoir la cohérence du code
- Utiliser l'anglais permet aussi d'augmenter le nombre de personnes pouvant lire le code et d'avoir de nombreux exemples existants pour s'inspirer.

En “camel case”

Un nom composé de plusieurs mots n'utilise ni espace ni ponctuation, et sépare les mots en mettant en capitale la première lettre de chaque mot.

Exemples: `getElementsByTagName`, `ListArray`, ...

Règles de nommage (2/2)



CamelCase vs camelCase

- Les noms qui définissent un type (classes, interfaces, enum, record, ...) commence par un majuscule.
- Les autres noms (méthodes, variables, attributs, ...) commencent par une minuscule.

Nommage des méthodes : cas 1

Méthodes procédurales

Méthodes modifiant l'état de l'objet
⇒ groupe verbal à l'infinitif.

Exemples

- `boolean add(E element)`
- `E set(int index, E element)`
- `boolean removeAll(Collection<?> c)`

Nommage des méthodes : cas 2

Expressions non booléennes

Méthodes renvoyant une partie de l'état de l'objet $\Rightarrow$ groupe nominal ou getter.

Exemples

- `int size()`
- `List<E> subList(int fromIndex, int toIndex)`
- `int hashCode()`
- `ListIterator<E> listIterator()`
- `E get(int index)`
- `Color getBackground()`
- `float getOpacity()`

Nommage des méthodes : cas 3

Expressions booléennes

Méthodes testant un prédicat sur l'objet $\Rightarrow$ groupe verbal au présent.

Exemples

- `boolean isEmpty()`
- `boolean contains(Object o)`
- `boolean equals(Object o)`

Nommage des méthodes : cas 4

Méthodes de conversion $\Rightarrow$ utilisation du `to`

Exemples :

- `String toString()`
- `Object[] toArray()`

Les règles ne sont pas absolues mais juste des conventions qui peuvent avoir des exceptions.

En général le plus simple est de s'inspirer de l'existant : par exemple la **JDK** et de bien réfléchir lorsqu'on souhaite déroger aux règles.

Comment rendre le nommage des méthodes facile ?

En écrivant des méthodes courtes

De préférence une dizaine de ligne maximum.

Comment écrire des méthodes courtes

En extrayant le plus possible les parties du code d'une méthode à d'autres méthodes.

Conseils

- Réfléchir avant de coder au rôle de la méthode
- Se demander ce qui peut être confié à d'autres méthodes

Ne pas mentir

```
class User {  
    private boolean authenticated;  
    private String password;  
  
    public boolean checkPassword(String password) {  
        if (password.equals(this.password)) {  
            authenticated = true;  
            return true;  
        }  
        return false;  
    }  
}
```

La méthode authentifie l'utilisateur alors qu'elle ne devrait que vérifier la validité du mot de passe d'après ses données.

Section 23

Des principes pour bien programmer

Comment rendre le nommage des méthodes facile ?

En écrivant des méthodes courtes

De préférence une dizaine de ligne maximum.

Comment écrire des méthodes courtes

En extrayant le plus possible les parties du code d'une méthode à d'autres méthodes.

Conseils

- Réfléchir avant de coder au rôle de la méthode
- Se demander ce qui peut être confié à d'autres méthodes

Pourquoi découper et nommer ?

Quelle est la sémantique du texte suivant ?

L'animal vertébré vivipare caractérisé par la présence de mamelles, qui se nourrit en grignotant à l'aide de leurs deux paires d'incisives, et qui se multiplie avec une rapidité d'intensité forte, aux organes d'audition et d'équilibration à l'étendue supérieure à la moyenne dans le sens de la longueur, fait descendre par le gosier, postérieurement à une réduction en petites parcelles avec les dents, un être vivant appartenant au règne végétal cultivé pour l'usage culinaire, muni de pédicelles partant en faisceau depuis son pédoncule, et dont la partie souterraine permettant la fixation au sol est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

L'animal vertébré vivipare caractérisé par la présence de mamelles,
qui se nourrit en grignotant à l'aide de leurs deux paires d'incisives,
et qui se multiplie avec une rapidité d'intensité forte,
aux organes d'audition et d'équilibration
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère

qui se nourrit en grignotant à l'aide de leurs deux paires d'incisives,
et qui se multiplie avec une rapidité d'intensité forte,
aux organes d'audition et d'équilibration
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère lagomorphe

et qui se multiplie avec une rapidité d'intensité forte,
aux organes d'audition et d'équilibration
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

**Le mammifère
lagomorphe
très prolifique**

aux organes d'audition et d'équilibration
à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

**Le mammifère
lagomorphe
très prolifique
aux oreilles**

à l'étendue supérieure à la moyenne dans le sens de la longueur,
fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

**Le mammifère
lagomorphe
très prolifique
aux oreilles
longues**

fait descendre par le gosier,
postérieurement à
une réduction en petites parcelles avec les dents,
un être vivant appartenant au règne végétal
cultivé pour l'usage culinaire,
muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère

lagomorphe

très prolifique

aux oreilles

longues

avale

postérieurement à

une réduction en petites parcelles avec les dents,

un être vivant appartenant au règne végétal

cultivé pour l'usage culinaire,

muni de pédicelles partant en faisceau depuis son pédoncule,

et dont la partie souterraine permettant la fixation au sol

est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère

lagomorphe

très prolifique

aux oreilles

longues

avale

après

une réduction en petites parcelles avec les dents,

un être vivant appartenant au règne végétal

cultivé pour l'usage culinaire,

muni de pédicelles partant en faisceau depuis son pédoncule,

et dont la partie souterraine permettant la fixation au sol

est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère

lagomorphe

très prolifique

aux oreilles

longues

avale

après

mâchage

un être vivant appartenant au règne végétal

cultivé pour l'usage culinaire,

muni de pédicelles partant en faisceau depuis son pédoncule,

et dont la partie souterraine permettant la fixation au sol

est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère

lagomorphe

très prolifique

aux oreilles

longues

avale

après

mâchage

une plante

cultivé pour l'usage culinaire,

muni de pédicelles partant en faisceau depuis son pédoncule,

et dont la partie souterraine permettant la fixation au sol

est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère

lagomorphe

très prolifique

aux oreilles

longues

avale

après

mâchage

une plante

potagère

muni de pédicelles partant en faisceau depuis son pédoncule,
et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère

lagomorphe

très prolifique

aux oreilles

longues

avale

après

mâchage

une plante

potagère

ombellifère

et dont la partie souterraine permettant la fixation au sol
est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

Le mammifère

lagomorphe

très prolifique

aux oreilles

longues

avale

après

mâchage

une plante

potagère

ombellifère

à racine

est d'une couleur située à l'extrémité du spectre rappelant celle du sang.

Pourquoi découper et nommer ?

**Le mammifère
lagomorphe
très prolifique
aux oreilles
longues
avale
après
mâchage
une plante
potagère
ombellifère
à racine
rouge.**

Pourquoi découper et nommer ?

Quelle est la sémantique du texte suivant ?

Le mammifère lagomorphe très prolifique aux longues oreilles avale après mâchage une plante potagère ombellifère à racine rouge.

Pourquoi découper et nommer ?

Le mammifère lagomorphe très prolifique aux longues oreilles
avale après mâchage
une plante potagère ombellifère à racine rouge.

Pourquoi découper et nommer ?

Le lapin

avale après mâchage

une plante potagère ombellifère à racine rouge.

Pourquoi découper et nommer ?

Le lapin

mange

une plante potagère ombellifère à racine rouge.

Pourquoi découper et nommer ?

**Le lapin
mange
une carotte.**

Pourquoi découper et nommer ?

Quelle est la sémantique du texte suivant ?

Le lapin mange une carotte.

Principes de programmation

Il existe de nombreux principes de programmation permettant de guider l'écriture de code :

- DOT : Do One Thing
- DRY : Don't Repeat Yourself
- KISS : Keep It Simple Stupid
- SRP : Single Responsibility Principle
- ...

Autres principes

Les 5 principes SOLID (principes de conception objet) que vous verrez en détail en L3 dans le cours de **Programmation et conception**

Do One Thing

Une fonction ne doit faire qu'**une seule chose**.

Pour cela, elle ne doit réaliser que des étapes de même niveau d'abstraction.

On décompose la fonction :

Pour faire la cuisine je dois (premier niveau d'abstraction) :

- choisir une recette;
- réunir les ingrédients;
- suivre la recette.

Pour choisir une recette, je dois (deuxième niveau d'abstraction):

- réfléchir à ce que j'ai envie de manger;
- chercher sur marmiton.

Mauvaise approche

```
void cook(){  
    // On choisit la recette  
    Food wantToEat = thinkAboutFood();  
    Recipe recipe = lookOnMarmiton(wantToEat);  
    // On réunit les ingrédients  
    openFridge();  
    for(Ingredient ingredient :  
        recipe.getFreshIngredients()){  
        takeInFrige(ingredient);  
    }  
    closeFridge();  
    openCupboard();  
    ...  
    // On suit la recette
```

Bonne approche

```
void cook(){  
    Recipe recipe = chooseRecipe();  
    gatherIngredients(recipe);  
    followRecipe(recipe);  
}
```

```
Recipe chooseRecipe(){  
    Food wantToEat = thinkAboutFood();  
    Recipe recipe = lookOnMarmiton(wantToEat);  
    return recipe;  
}
```

...

Do One Thing

Do One Thing

Une variable/méthode/classe doit n'avoir qu'une seule signification.

Une variable/méthode/classe **ne** doit **pas** avoir :

- un sens dans une circonstance et autre sens dans un domaine différent,
- ou bien avoir deux sens en même temps.

Le principe **Do One thing** est fortement lié aux principes suivants :

- **Don't Repeat Yourself** : il ne faut pas se répéter car la répétition produit de l'inconsistance et des erreurs.
- **Once and Only Once** : chaque comportement ne doit être défini qu'une fois.

Do one Thing pour les méthodes

Une méthode ne doit avoir qu'une **responsabilité = raison de changer**.

Comment déceler une méthode ne respectant pas DOT

- Une méthode ayant un “and” dans le nom $\Rightarrow$ `doOneThingAndAnotherThing`
- Une méthode peut réalisant une action complexe sans appeler d'autres méthodes.
- Une méthode trop longue (plusieurs dizaines de lignes de code)
- un niveau d'indentation élevé : par exemple un `if` dans un `for` dans un `for`

Exemple de méthode ayant trop de responsabilités

```
int countAndPrintSquares(List<Rectangle> rectangles) {  
    int countSquare = 0;  
    for (Rectangle rectangle : rectangles) {  
        if (rectangle.width == rectangle.height) {  
            System.out.println(rectangle);  
            countSquare++;  
        }  
    }  
    return countSquare;  
}
```

La méthode doit :

- tester si un rectangle est un carré
- afficher les carrés

Chaque méthode a une responsabilité

```
boolean isSquare(){ return this.width == this.height; }
static int printSquares(List<Rectangle> rectangles) {
    for (Rectangle rectangle : rectangles) {
        if (rectangle.isSquare()) {
            System.out.println(rectangle);
        }
    }
}
static void countSquares(List<Rectangle> rectangles) {
    for (Rectangle rectangle : rectangles) {
        if (rectangle.isSquare()) squareCount++;
    }
    return squareCount;
}
```

Autre exemple

```
boolean removeFirstMinimumValue(List<Integer> list) {  
    if (list.isEmpty()) { return false; }  
    int position = 0;  
    for (int i = 0; i < list.size(); i++) {  
        if (list.get(position) > list.get(i)) {  
            position = i;  
        }  
    }  
    list.remove(position);  
    return true;  
}
```

Correction (1/2)

```
int findMinimumValuePosition(List<Integer> list) {  
    if (list.isEmpty()) {  
        throw new IllegalArgumentException();  
    }  
    int position = 0;  
    for (int i = 0; i < list.size(); i++) {  
        if (list.get(position) > list.get(i)) {  
            position = i;  
        }  
    }  
    return position;  
}
```

Correction (2/2)

```
boolean removeFirstMinimumValue(List<Integer> list) {  
    if (list.isEmpty()) {  
        return false;  
    }  
    int position = findMinimumValuePosition(list);  
    list.remove(position);  
    return true;  
}
```

Single Responsibility Principle (SRP)

Principe SRP

Une classe ne doit avoir qu'une **responsabilité = raison de changer**

Les effets néfastes des responsabilités multiples

- Difficulté à nommer car la classe est trop complexe
- Difficulté à réutiliser le code car seulement une des responsabilités est réutilisable
- Difficulté à mettre à jour le code car changer l'implémentation d'une partie impacte les autres parties

Pourquoi SRP ?

- Facilite le nommage et documentation
- Facilite l'écriture des tests
- Facilite la réutilisation des classes

Section 24

Types paramétrés

Stack d'Object

Supposons que nous ayons la classe suivante :

```
public class Stack {  
    private Object[] stack = new Object[100];  
    private int size = 0;  
    public void push(Object object) {  
        stack[size] = object; size++;  
    }  
  
    public Object pop() {  
        size--;  
        Object object = stack[size];  
        stack[size]=null; // Pour le Garbage Collector.  
        return object;  
    }  
}
```

Problème de Stack d'Object

Nous rencontrons le problème suivant :

```
Stack stack = new Stack();  
String string = "truc";  
stack.push(string);  
string = (String)stack.pop();  
// Transtypage (cast) obligatoire !
```

Nous avons également le problème suivant :

```
Stack stack = new Stack();  
Integer intValue = new Integer(2);  
stack.push(intValue);  
String string = (String)stack.pop();  
// Erreur à l'exécution
```


La solution : types paramétrés

Par conséquent, on souhaiterait pouvoir préciser le type des éléments :

```
Stack<String> stack = new Stack<String>();  
String string = "truc";  
stack.push(string); // Le paramètre doit être un String.  
String string = stack.pop(); // retourne un String.
```

Java nous permet de définir une classe Stack qui prend en paramètre un type. Ce type paramétré va pouvoir être utilisé dans les signatures des méthodes et lors de la définition des champs de la classe.

Lors de la compilation, Java va utiliser le type paramétré pour effectuer :

- des vérifications de type ;
- des transtypes automatiques ;
- des opérations d'emballage ou de déballage de valeurs.

Définition de classes paramétrées

La nouvelle version de la classe Stack :

```
public class Stack<T> {  
    private Object[] stack = new Object[100];  
    private int size = 0;  
    public void push(T element) {  
        stack[size] = element;  
        size++;  
    }  
    public T pop() {  
        size--;  
        T element = (T)stack[size];  
        stack[size] = null;  
        return element;  
    }  
}
```

Emballage et déballage

Les types primitifs ne sont pas des classes :

Dans le cas d'un `int`, on doit utiliser la classe d'emballage (wrapper class) `Integer` :

Interdit : ~~`Stack<int> stack = new Stack<int>();`~~

Autorisé :

```
Stack<Integer> stack = new Stack<Integer>();
int intValue = 2;
Integer integer = Integer.valueOf(intValue);
// → emballage du int dans un Integer.
stack.push(integer);
Integer otherInteger = stack.pop();
int otherIntValue = otherInteger.intValue();
// → déballage du int présent dans le Integer.
```

Types primitifs

type	classe d'emballage	taille	valeurs possibles
byte	Byte	8 bits	-128 à 127
short	Short	16 bits	-32768 à 32767
int	Integer	32 bits	-2^{31} à $2^{31} - 1$
long	Long	64 bits	-2^{63} à $2^{63} - 1$
float	Float	32 bits	
double	Double	64 bits	
char	Character	16 bits	caractère unicode
boolean	Boolean	non définie	false ou true

Classes d'emballage

La classe `Number` sert de base pour toutes les classes d'emballage.

Elle contient les méthodes suivantes :

- `public int intValue()`
- `public long longValue()`
- `public float floatValue()`
- `public double doubleValue()`
- `public byte byteValue()`
- `public short shortValue()`

Classes d'emballage

Les classes d'emballage étendent `Number` :

- `Byte` → `public static Byte valueOf(byte b)`
- `Short` → `public static Short valueOf(short s)`
- `Integer` → `public static Integer valueOf(int i)`
- `Long` → `public static Long valueOf(long l)`
- `Byte` → `public static Byte valueOf(byte b)`

Ils existent des constructeurs mais ils sont dépréciés (et donc pas à utiliser).

La classe Character

Les classes d'emballage ne contiennent pas que des méthodes liées aux instances :

- `public static Byte valueOf(byte b)`
- `public static char charValue()`
- `public static boolean isLowerCase(char ch)`
- `public static boolean isUpperCase(char ch)`
- `public static boolean isDigit(char ch)`
- `public static boolean isLetter(char ch)`
- `public static boolean isLetterOrDigit(char ch)`
- `public static char toLowerCase(char ch)`
- `public static char toUpperCase(char ch)`
- `public static char toTitleCase(char ch)`

Emballage et déballage automatique

Depuis Java 5, il existe l'emballage et le déballage automatique :

```
Stack<Integer> stack = new Stack<Integer>();  
int intValue = 2;  
stack.push(intValue);  
// → emballage automatique du int dans un Integer.  
int otherIntValue = stack.pop();  
// → déballage automatique du int.
```

Attention

Il est important de noter que des allocations sont effectuées lors des emballages sans que des new soient présents dans le code.

Plusieurs paramètres de types

```
public class Pair<A, B> {  
    public A first;  
    public B second;  
    public Pair(A first, B second) {  
        this.first = first;  
        this.second = second;  
    }  
    public static <A, B> Pair<A,B>  
        makePair(A first, B second) {  
        return new Pair<A,B>(first, second);  
    }  
}
```

Utilisation d'une classe avec plusieurs paramètres de types

```
public class Main {  
  
    public static void main(String[] args) {  
        Pair<String,Integer> pair =  
            Pair.makePair("tot",12);  
        System.out.println(pair);  
    }  
}
```

Section 25

Interfaces génériques

Exemple d'interfaces génériques en Java

- Comparable<T> : objets qu'on peut comparer à des objets de type T.
- List<T> : liste d'objets de type T.
- Stack<E> : pile d'objet de type T.
- Iterable<T> : collection d'objet de type T qu'on peut parcourir avec un boucle.

Rappel : types paramétrés/génériques

Types dont la définition contient un autre type.

Définition de Comparable<T>

```
public interface Comparable<T>{  
    /**  
     * Compares this object with the specified object for  
     * order. Returns a negative integer, zero, or a  
     * positive integer as this object is less than,  
     * equal to, or greater than the specified object.  
     * @param other the objet 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.  
     */  
    int compareTo(T other);  
}
```

Utilisation de Comparable<T>

// Utilisation typique

```
public class MyOrderedClass
    implements Comparable<MyOrderedClass>{

    int compareTo(MyOrderedClass other){
        // ...
    }
}
```

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

Utilisation de Comparable<T> pour la classe Student

```
public class Student
    implements Comparable<Student>{

    public final String lastName;
    public final int idNumber;
    //...
    public int compareTo(Student other){
        return lastname.compareTo(other.lastName);
    }
}

List<Student> students; //...
Collections.sort(students);
// Tri par nom de famille
```

Utilisation de Comparable<T> pour la classe Student

```
public class Student
    implements Comparable<Student>{

    public final String lastName;
    public final int idNumber;
    //...
    public int compareTo(Student other){
        return idNumber - other.idNumber;
    }
}
```

```
Student[] students; //...
Arrays.sort(students);
// Tri par numéro d'étudiant
```


Interface Iterable

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

- Définition complexe
- Utilisation facile et utile

Utilisation d'Iterable

Utilisation : si une classe implémente `Iterable<T>`, ces instances contiennent une collection d'objets de type `T`.

On peut parcourir les objets de la collection à l'aide d'une boucle `for`.

```
public class Order{
    Iterable<Items> items;
    //...
    public void printAllItems(){
        for (Item item : this.items){
            System.out.println(item.toString());
        }
    }
}
```

Interface Collection

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

Mot-clé extends

Quand une interface “fille” étend une interface “mère”, elle hérite de toutes les méthodes de sa “mère”.

Une classe implémentant la classe “fille” doit donc définir les méthodes des deux interfaces.

Section 26

Collections

Pourquoi des collections d'objets ?

- pour manipuler des ensembles d'objets simultanément,
- pour ranger et accéder à des objets selon les besoins,
- avoir une manière de grouper des objets plus souple que les tableaux.

Une variété de collections

On ne manipule pas toutes les collections de la même façon :

- grouper des valeurs pour les traiter ensemble,
- ordonner des valeurs,
- tester l'appartenance d'éléments,
- créer une file d'attente,
- associer des valeurs entre elles,

Différentes collections existent, plus ou moins efficaces selon les opérations.

Exemples de collection

Exemples d'utilisation de collections dans la vie courante.

- Réfrigérateur : **liste** (*list*) d'aliments mis au frais.
- File d'attente au guichet : **file** (*queue*) de personnes attendant dans l'ordre d'arrivée.
- Pile de T-shirts : **pile** (*stack*) de vêtements avec un ordre et un accès uniquement possible avec le dernier objet empilé.
- Annuaire : **association** (*map*) de noms avec des numéros de téléphone.

Section 27

Liste : l'interface List

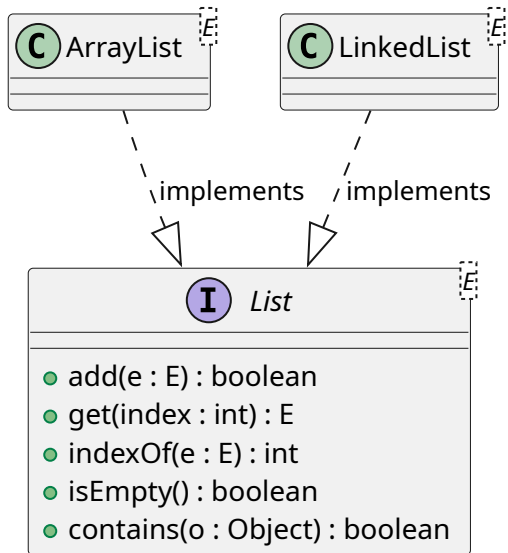
L'interface List

Vous avez déjà utiliser List : séquence ordonnée indexable par des entiers.

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

Implémentations : LinkedList, ArrayList, ...

L'interface List version diagramme



Section 28

Ensembles : l'interface Set

Exemple d'utilisation d'ensemble

Exemples d'opérations :

- ajouter un ami,
- enlever un ami,
- déterminer si une personne est votre ami,
- déterminer si vous avez au moins un ami

Structure d'ensemble (*Set*)

L'interface Set<E>

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

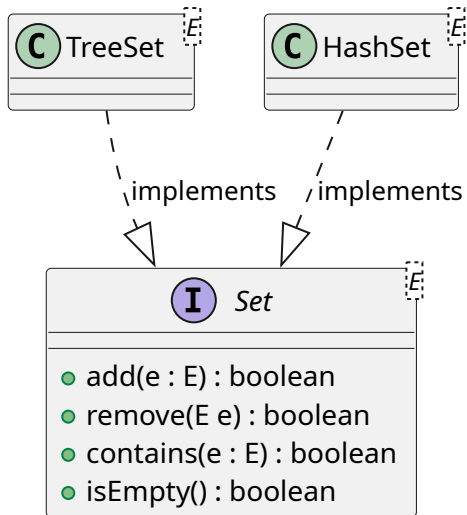
E peut être remplacé par le type d'objets de votre choix

Exemple:

Set<Integer>, Set<String>, Set<List<Double>>, ...

Implémentations disponibles : TreeSet, HashSet

L'interface Set



Type de retour

Pourquoi `add`, `remove` retournent des valeurs `true` ou `false` (de type `boolean`) ?

Règle

Par convention, les méthodes modifiant une collection retourne

- `true` si la modification a été réalisée,
- `false` si aucune modification n'a eu lieu.

Exemple

`false` est retournée par `set.remove(elt)` si `elt` n'était pas dans l'ensemble.

Exemple d'utilisation de Set

```
public Set<String> dictionary(List<String> text) {  
    Set<String> words = new HashSet<>();  
    for (String word : text) {  
        words.add(word);  
    }  
    return words;  
}
```

Remarques

- l'initialisation d'un ensemble vide,
- l'opération d'ajout d'un élément.

Différences entre List et Set

- un objet Set contient chaque élément *au plus une fois*, un objet List peut avoir des répétitions,
- un objet Set peut **rapidement** accomplir les opérations add, contains, remove. Un objet List accomplit contains et remove **très lentement**.
- les éléments d'un objet List sont classés en séquence (0,1,2, ...) et indicés, les éléments d'un objet Set ne sont pas classés. On ne peut pas trier un Set.
- Set utilise la méthode equals de ses éléments, et HashSet utilise la méthode hashCode, il faut donc les implémenter ou utiliser les méthodes par défaut.

Section 29

Files et piles

File d'attente

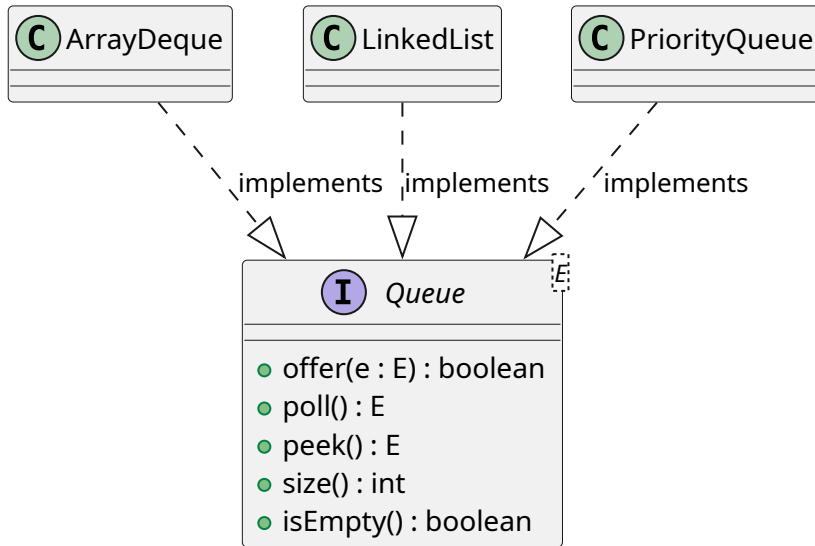
Exemples d'opérations :

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

Structure de file (*Queue*).

Les éléments sortent dans l'ordre où ils sont rentrés (ordre FIFO, *First-in First-out*).

L'interface Queue



Pile de T-shirts

Exemples d'opérations :

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

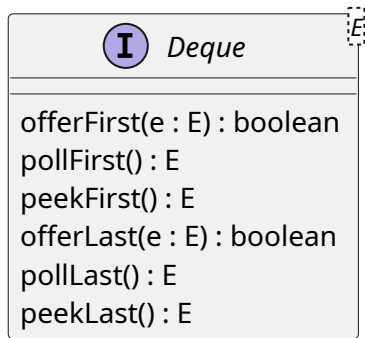
Structure de pile (*Stack*).

Les éléments sortent dans l'ordre **inverse** où ils sont rentrés (LIFO, *Last-in First-out*).

- **File** : premier entré, premier sorti, une file a deux extrémités actives, une extrémité où l'on rentre et l'autre dont l'on sort.
- **Pile** : dernier entré, premier sorti, une pile a une seule extrémité active (le haut de la pile), à laquelle on ajoute ou retire des éléments.

En Java, **une seule interface commune aux deux : file à double extrémité** (*deque* : abréviation de *double-ended queue*).

L'interface Deque



- `poll` : retirer et retourner l'élément à l'extrémité,
- `peek` : retourner l'élément à l'extrémité, sans le retirer de la deque.

Implémentations : `ArrayDeque`, `LinkedList`

Patron de méthode des files

Traiter tous les éléments de la file, dans l'ordre :

```
public void process(Deque<E> queue) {  
    while (!queue.isEmpty()) {  
        E first = queue.pollFirst();  
        process(first);  
    }  
}
```

Le traitement d'un élément peut provoquer l'ajout de nouveaux éléments dans la file.

Attention à tester que la file est non-vide avant d'extraire un élément !
(sinon vous obtiendrez null).

Exemple d'utilisation d'une Deque

Calcul des scores possibles au rugby (< 100) :

- Au rugby on peut marquer soit 3 points, soit 5 points, soit 7 points d'un coup.
- Le score d'une équipe ne peut donc pas être un seul point, ou 2 points, ou 4 points, mais peut être 6 points ($3 + 3$), 8 points ($3 + 5$), 9 points ($3 + 3 + 3$), ...
- Quels sont les scores réalisables ?
- Pour chaque score n réalisable, $n + 3$, $n + 5$ et $n + 7$ sont aussi réalisables. 0 est réalisable.

Exemple d'utilisation d'une Deque

```
Set<Integer> scores = new HashSet<>();  
public void computeScores() {  
    Deque<Integer> feasibles = new ArrayDeque<>();  
    feasibles.offerLast(0);  
    while (!feasibles.isEmpty()) {  
        addScore(feasibles.pollFirst(), feasibles);  
    }  
}  
  
public void addScore(int n, Deque<Integer> feasibles) {  
    if (n >= 100 || scores.contains(n)) { return; }  
    scores.add(n);  
    feasibles.offerLast(n+3);  
    feasibles.offerLast(n+5);  
    feasibles.offerLast(n+7);  
}
```

Exemple d'utilisation d'une Deque

- `offerLast + pollFirst` : utilisation de la deque comme une file.
- Instanciation avec `new ArrayDeque<>()`.
- La file contient les scores réalisables.
- Si un score est déjà connu, on le saute.

Section 30

Tables d'associations

Exemples d'opérations :

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

Structure de table d'associations (Map), en Python : dict.

L'interface Map

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

De nouveau, on peut remplacer K et V par n'importe quels types.

Implémentations : HashMap, TreeMap

Exemple d'utilisation de Map

On veut compter le nombre d'appels téléphoniques effectués vers chaque numéro.

- Associer à chaque numéro le nombre d'appel.
- Lorsqu'un appel est effectué, récupérer le nombre d'appels associé et l'augmenter de 1. Cas spécial : premier appel.
- Notez bien dans le prochain transparent : l'initialisation, l'ajout d'une association, le test d'une clé, la récupération d'une valeur.

Exemple d'utilisation de Map

```
public class PhoneCallStats {  
    Map<PhoneNumber,Integer> callCounts = new HashMap<>();  
  
    public void call(PhoneNumber number) {  
        int oldCount = this.getCallCount(number);  
        callCounts.put(number,oldCount+1);  
    }  
  
    public int getCallCount(PhoneNumber number) {  
        if(callCounts.containsKey(number))  
            return callCounts.getNumber();  
        return 0;  
    }  
}
```


Section 31

Collections

L'interface Collection

Set, Queue, Deque, List **étendent** (*extends*) l'interface Collection (**mais pas Map !**).

Ce qui signifie que toutes ces interfaces possèdent aussi les services définies par l'interface Collection.

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

Extension

Une interface A étends une interface B :

- interface A extends B à la déclaration de l'interface.
- l'interface A définit toutes les méthodes de l'interface B.

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>();
```

Itérer sur une collection

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

La boucle `for` permet de faire les mêmes instructions pour tous les éléments d'une collection.

```
Collection<Item> itemList = ...;  
for (Item item : itemList) {  
    ... // instructions utilisant item  
}
```

Les éléments sont traités un par un.

Ordre des éléments : dépend de l'implémentation de la collection.

Syntaxe du for (foreach)

```
for (Item item : collection) { ... }
```

- for : mot-clé de répétition,
- les paramètres de la répétition :
 - ▶ Item : le type des éléments de la collection,
 - ▶ item : un identifiant pour une **nouvelle variable** désignant l'élément de la collection, qui change à chaque itération,
 - ▶ : se lit *in* (contenu dans),
 - ▶ collection : la collection sur laquelle itérer.
- ... : les instructions à répéter pour chaque élément de la collection.

Interrompre une itération

Il peut être nécessaire d'interrompre l'itération au milieu de la collection.

- `return` fait sortir de la méthode, donc de l'itération.
- `break` fait sortir de l'itération. La prochaine instruction évaluée est celle suivant la fin de la boucle `for`.

Exemple d'interruption

- Choisir un objet avec une probabilité proportionnelle à son poids.
- `totalWeight` est la somme des poids des objets.

```
int s = randomGen.nextInt(totalWeight);
Item selected = null;
for (Item item : itemList) {
    s = s - item.weight;
    if (s <= 0) {
        selected = item;
        break;
    }
}
selected.display();
// prochaine instruction après break
```

Interrompre une itération

Il peut être nécessaire de sauter certains éléments d'une collection.

- `continue` arrête le traitement de l'élément en cours et passe à l'élément suivant de l'itération. La prochaine instruction évaluée est la première instruction du bloc de la boucle avec l'élément suivant (s'il reste au moins un élément).

```
int sum = 0;
int product = 1;
for (Item item : items) {
    if (item.isBad()) { continue; }
    sum = sum + item.value();
    product = product * item.value();
}
```


Patron de méthode : itérer une opération

Modèle pour faire une opération sur chaque élément.

```
public void iterateProcess(Collection<Elt> elements) {  
    initialise();  
    for (Elt element : elements) {  
        process(element);  
    }  
    return;  
}
```

Résumé des interfaces des collections et map

nom	interface	instanciation
liste	List	<code>new ArrayList<>()</code>
pile/file	Deque	<code>new ArrayDeque<>()</code>
ensemble	Set	<code>new HashSet<>()</code>
table d'associations	Map	<code>new HashMap<>()</code>

Les noms des interfaces sont accompagnés du type des éléments : `List<Integer>`, `Set<Node>`, `Map<Node, Point2D>`, ...

Entre `<...>`, on utilise `Integer`, `Double`, `Boolean` plutôt que `int`, `double`, `boolean`.