TD
Programmation orientée objets

Yannick Loiseau yannick.loiseau@uca.fr

Licence 2me année
version du 23 février 2022

Les solutions données dans cette correction sont beaucoup plus détaillées que ce qui est exigé en séance de TD (ou en examen), ceci dans le but de donner le maximum d'éléments à celles et ceux qui désireraient aller plus loin.

Attention : la rédaction de cette correction n'est pas finalisée. Certaines réponses sont insuffisamment détaillées.

TD 1

Objets et rappels d'algorithmique

Exercice 1 (Instants) Un instant est défini par un nombre d'heures (entre 0 et 23), de minutes (entre 0 et 59) et de secondes (entre 0 et 59).

Question 1 *:* Est-ce qu'un instant peut changer ? Doit on définir des méthodes pour lire les valeurs d'un instant ? Doit on définir des méthodes pour modifier les valeurs d'un instant ?

Réponse : La modifiabilité de l'état d'un objet (on parle aussi de mutabilité) est un élément à considérer. Est-il perminent et souhaitable pour un objet donné de changer d'état, c'est-à-dire de modifier la valeur de ses attributs, ou est-il préférable que l'objet ne change jamais d'état. En effet, le fait qu'un objet puisse changer d'état rend certains aspects plus complexe, comme le raisonement sur la validité du programme, l'utilisation de l'objet dans un contexte de programmation concurente, les erreurs d'*aliasing*, etc. Dans le cas où l'objet est modifiable, les méthodes permettant de changer son état doivent être élaborée. Ici par exemple, doterait-on l'objet de setters pour chaque attribut, ou d'une seule méthode permettant de « décaler » un instant (en donnant directement la durée de décalage). Cette dernière approche est souvent préférable (voir *Tell don't ask*) Il est donc souvent préférable, si cela est possible, de créer des objets non modifiable (voir notamment Bloch, 2018, Item 17).

Ici, *a priori*, un instant ne doit pas changer. En effet, les instants seront comparés par leurs valeurs plutôt que par leurs adresses (cf. question 4 et suivantes). De ce point de vue, il est plus cohérent et prudent de rendre l'objet non modifiable, afin de pouvoir l'utiliser sans risque dans des ensembles ou des tableaux associatifs (après avoir redéfini les bonnes méthodes). On appelle ce type d'objet un *objet valeur*, ou *datatypes*, notamment en UML (*Unified Modeling Language*) avec le stéréotype correspondant, et correspond au patron de conception *Value Object* décris notamment dans Fowler, 2002, ou aux *records* de certains langages.

Les instances étant non modifiables, il ne faut pas créer de mutateur. Les attributs devront donc être initialisés dans le constructeur, qui aurra un paramètre pour chacun

des attributs. Il peut par contre être utile de définir des accesseurs en lecture, pour *encapsuler* la manière dont sont stockés les paramètres de l'instant. Dans d'autres langage que JAVA (PYTHON, C#, ECMASCRIPT), pourrait utiliser des *properties* pour contrôler l'accès en écriture et représenter les attributs calculés. Notez cependant que ces accesseurs ne sont pas nécessaires. De manière générale, il vaut mieux éviter de définir des accesseurs si ceux-ci ne sont pas requis.

> Les *properties*, que l'on trouve par exemple en PYTHON, C# ou ECMASCRIPT, permettent de définir des accesseurs qui seront appelés de manière implicite. On accède donc aux valeur avec la même syntaxe que s'ils s'agissaient d'attributs publics, mais ce sont les accesseurs qui sont appelés. Ainsi, `print(obj.foo)` est « transformé » en `print(obj.get_foo())` et `obj.foo = 42` est « transformé » en `obj.set_foo(42)`.

Comme les valeurs des attributs sont ici des types primitifs, et donc aussi des types valeur, on aurait également pu utiliser des attributs publics finaux (`final`), mais les accesseurs permettent d'avoir une interface homogène quelle que soit l'implémentation choisie, et donc de bien encapsuler notre classe. Passer par une méthode ajoute en effet un niveau d'indirection. En n'accédant pas directement à l'attribut, mais en passant par une méthode, on masque l'existence même de l'attribut, ce qui laisse l'opportunité de changer l'implémentation sans que le code client utilisant la classe n'ait besoin de changer, tant que les méthodes en question sont présentes.

Question 2 (Mise en œuvre) *:* Écrivez une classe Java permettant de créer des instants. Envisagez différentes solution pour celle-ci, notamment en rapport avec les attributs et le (ou les) constructeur. Vous veillerez à bien respecter l'encapsulation de votre classe. Implémentez également la méthode `toString` qui retourne une représentation textuelle de l'instant.

Réponse : D'un point de vue conceptuel, notre objet a trois attributs publics (en lecture seule, voir la réponse à la question 1). Pour faciliter la suite (notamment les questions 3 et 4), nous allons également introduire un attribut dérivé représentant le nombre total de secondes.

Le diagramme UML de niveau conceptuel correspondant est donc celui de la figure 1.1 page suivante. Ce diagramme ne représente pas l'implémentation de la classe. Par exemple, les attributs sont représentés comme public, car conceptuellement l'objet possède ces propriétés. On verra dans la suite que l'implémentation peut être très différente, notamment pour des considérations d'encapsulation. Le stéréotype UML «datatype» implique la comparaison par valeur comme discuté précédemment.

D'un point de vue mise en œuvre, on peut donc choisir :

— de stocker ces trois attributs, et de fournir un accesseur qui calculera le nombre total de secondes ;

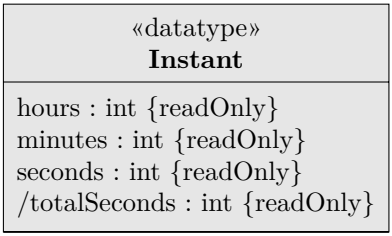

FIGURE 1.1 – Diagramme de classe d'un instant au niveau conceptuel

— de stocker le nombre total de secondes et de fournir des accesseurs pour les autres attributs qui calculerons le nombre d'heures, minutes et secondes ;
— de stocker les quatre attributs, en faisant les calculs nécessaires une fois pour toute, dans le constructeur par exemple, puisque notre objet ne changera pas.

Pour le constructeur, la première approche stockera telles quelles les valeurs passées, après avoir vérifié leur validité ; la seconde calculera le nombre total de secondes et stockera cette valeur, après validation.

Pour des raisons de facilité de maintenance et d'évolutivité du code, il est bon d'encapsuler notre classe, c'est-à-dire de ne définir que des attributs privé et de fournir des accesseurs (en lecture ou écriture). L'encapsulation permet aux classes clientes de rester indépendant de l'implémentation réelle de notre classe, tant que les méthodes publiques définies dans l'interface sont présentes, et donc ici de choisir indifféremment une des approches, selon des considérations de performance en terme de temps d'exécution ou d'espace mémoire.

Cependant, la deuxième solution est plus intéressante à plusieurs égards. En effet, il est plus simple de vérifier les préconditions dans ce cas là, et il devient alors possible de construire un instant avec des valeurs initiales différentes du domaine voulu, et même négatives, tant que les valeurs finales pour les heures, minutes, et secondes sont valides (c'est évidemment possible aussi dans la première approche, mais moins facile). L'inconvénient majeur est que les valeurs pour les heures, minutes et secondes sont calculée à chaque fois que l'on cherche à y accéder. Si cela s'avérait être critique en terme de performance, il serait facile de stocker les valeurs calculées dans un attribut privé, sans pour autant changer l'interface de notre classe, et donc sans apporter de modifications au code l'utilisant. Il faut cependant privilégier dans un premier temps une conception facile à maintenir, les optimisations ne venant qu'*a posteriori*, après avoir mesuré les performances et identifier les points critiques. Nous utiliserons donc ici la deuxième approche.

Notre classe dispose donc d'un attribut privé pour stocker le nombre total de secondes ainsi qu'un accesseur permettant d'y accéder en lecture. On implémente aussi trois méthodes retournant les nombres d'heures, minutes et secondes. Ce ne sont pas des accesseurs à proprement parler, puisqu'elles ne retournent pas un attribut,

```java
public class Instant {
  public static final int SECONDS_IN_MINUTE = 60;
  public static final int MINUTES_IN_HOUR = 60;
  public static final int SECONDS_IN_HOUR = MINUTES_IN_HOUR *
    SECONDS_IN_MINUTE;
  public static final int MAX_HOURS = 24;
  public static final int MAX_TOTALSECONDS = MAX_HOURS *
    SECONDS_IN_HOUR;
  public static final int MIN_TOTALSECONDS = 0;

  private final int totalSeconds;

  public int hours() {
    return this.totalSeconds() / SECONDS_IN_HOUR;
  }

  public int minutes() {
    return (this.totalSeconds() / SECONDS_IN_MINUTE)
           % MINUTES_IN_HOUR;
  }

  public int seconds() {
    return this.totalSeconds() % SECONDS_IN_MINUTE;
  }

  public int totalSeconds() { return this.totalSeconds; }

  @Override
  public String toString() {
    return String.format("[%dh %02dm %02ds]",
                         this.hours(),
                         this.minutes(),
                         this.seconds());
  }
}
```

LISTING 1.1 – Code Java de la classe Instant

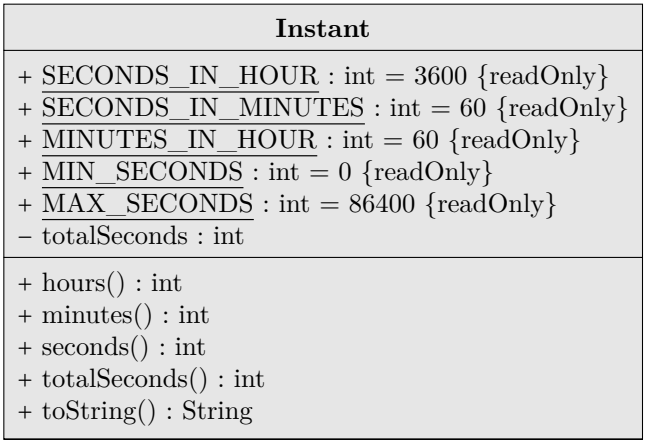

FIGURE 1.2 – Diagramme de classe d'un instant au niveau implémentation

mais peuvent être considérer comme tels puisque donnant accès à des propriétés (état) de notre instant. Le fait que ces valeurs soient sockées ou non est un détail de mise en œuvre. Pour faciliter la lecture de la sortie, on redéfini la méthode `toString` par défaut afin de fournir un affichage plus lisible. C'est une méthode que possèdent tous les objets JAVA, puisqu'elle est définie dans la classe `Object`. Elle peut cependant être redéfinie dans une classe (c'est-à-dire personnalisée). Elle retourne une représentation de l'objet sous forme de chaine de caractères (`String`), généralement compatible avec un affichage dans le terminal par exemple. Elle est notamment appelée implicitement par la méthode `println` pour afficher l'objet. On retrouve ce type de méthode dans beaucoup de langages (`toString` en ECMASCRIPT, `__str__` en PYTHON, etc.)

Nous alons également définir des constantes, c'est-à-dire des attributs statiques finaux, pour représenter différentes valeurs de calcul, telles que le nombre de secondes dans une heure par exemple. Il est recommandé de définir des constantes lorsque l'on doit utiliser des valeurs numériques non évidentes. Il est ainsi plus facile de les faire évoluer, et le code est beaucoup plus lisible.

Le diagramme de classe au niveau implémentation est représenté figure 1.2. Notre classe prend ainsi la forme du listing 1.1 page précédente.

Même au sein de la classe elle-même, on accède au nombre total de secondes via la méthode `totalSeconds` plutôt que par l'attribut privé lui-même. Cela s'appelle de l'*auto-encapsulation*, et permet de simplifier la redéfinition dans une sous-classe par la suite et simplifie l'évolution du code.

Constructeurs Considérons à présent les constructeurs de cette classe. Comme nos instances ne seront pas modifiables, il faut faire en sorte de passer les valeurs requises au constructeur de la classe.

```java
public Instant(int h, int m, int s) {
  this(validateTotalSeconds(secondsFrom(h, m, s)));
}

private Instant(int total) {
  this.totalSeconds = total;
}

private static int validateTotalSeconds(int s) {
  if (s < MIN_TOTALSECONDS || s >= MAX_TOTALSECONDS) {
    throw new IllegalArgumentException(String.format(
        "Total seconds must be between %s and %d and got %d",
        MIN_TOTALSECONDS, MAX_TOTALSECONDS, s));
  }
  return s;
}

private static int secondsFrom(int hours, int minutes, int seconds) {
 return (hours * SECONDS_IN_HOUR)
      + (minutes * SECONDS_IN_MINUTE)
      + seconds;
}
```

LISTING 1.2 – Constructeurs d'Instant

On crée ici deux constructeurs s'initialisant avec les heures, minutes, secondes pour le premier et avec le nombre total de secondes pour le deuxième. Ce dernier constructeur facilitera l'implémentation d'autres méthodes. Le constructeur utilisé lors de l'instanciation sera déterminé par le type et le nombre d'arguments fournis. On utilise donc ici de la *surcharge* de constructeur. Ces deux constructeurs sont donnés en listing 1.2 page précédente.

Pour faciliter la lecture du code, on crée deux méthodes statiques : la première, `secondsFrom` permet de convertir des heures, minutes et secondes en nombre total de secondes, et la deuxième, `validateTotalSeconds`, valide qu'un nombre total de seconde est bien dans l'intervalle requis. Ces méthodes sont statique, car elle ne travaillent que sur leurs arguments sans impliquer d'instance d'`Instant`, et privée car elle ne servent qu'au fonctionnement interne de la classe, et ne sont donc pas exposées au code utilisateur. Il est souvent intéressant de définir des méthodes de validation pour vérifier les préconditions de l'état des objets. Ces méthodes lèvent une exception si la valeur donnée est invalide, et retourne cette dernière sinon. Il est ainsi facile de les utiliser pour l'initialisation des attributs de l'objet. De cette manière, il est impossible de construire une instance d'`Instant` qui soit invalide : la validité de l'état des objets est garantie.

Classiquement, les *datatype* sont des types finaux, c'est-à-dire qui ne peuvent pas être étendus. Une des raisons principale est de faciliter l'implémentation de la comparaison par valeur, qui est assez complexe en présence de sous-typage, comme nous le verrons à la question 4. Pour l'intérêt de la suite, nous garderons cette classe ouverte à l'extension ; elle ne sera donc pas marquée `final`. Il est cependant possible de limiter l'extensibiliter d'une classe en réduisant la visibilité du constructeur. Le constructeur de la sous-classe devant appeller celui ce sa super-classe, un constructeur privé limite de fait l'extensibilité de la classe : les seules sous-classes possibles seront des classes internes. C'est l'approche que nous allons utiliser ici (après avoir supprimé le constructeur public précédent). Le constructeur étant privé, il faut fournir une méthode statique alternative pour l'instanciation d'un `Instant`. De manière générale, ce type de méthode (des *factories*) est souvent plus intéressant qu'un constructeur. Outre controler l'extensibilité de la classe, ils permettent de définir plusieurs façon de construire un objet, avec des noms explicites pour les méthodes. Un autre intérêt est de permetre de valider les arguments *avant* l'instanciation de l'objet, ce que ne permet pas le constructeur. Le constructeur étant privé, la validation peut désormais se faire en amont. Voir notamment Bloch, 2018, Items 1 et 4 sur ce type de méthodes et leurs intérêts.

Question 3 (Différence de deux instants) *:* Écrivez une méthode Java `minus` qui retourne le temps entre deux instants.

Réponse : Le temps entre deux instants sera représenté par un instant. Il y deux manières de créer cette méthode :

```java
public static Instant of(int total) {
  return new Instant(validateTotalSeconds(total));
}

public static Instant of(int hours, int minutes, int seconds) {
  return Instant.of(secondsFrom(hours, minutes, seconds));
}
```

LISTING 1.3 – Méthodes statique d'instanciation d'`Instant` (*factories*)

```java
public Instant minus(Instant other) {
  if (other == null) {
    return this;
  }
  return Instant.of(this.totalSeconds() - other.totalSeconds());
}
```

LISTING 1.4 – Différence d'`Instant`

1. une méthode statique prenant deux instants en paramètres, et s'utilisant donc ainsi : `Instant t3 = Instant.minus(t1, t2);`
2. une méthode d'instance : `Instant t3 = t1.minus(t2);`

La deuxième solution est bien évidemment plus fidèle aux concepts de l'approche objet. On pourrait dans l'absolu utiliser une méthode qui modifie l'instant `t1` plutôt que retourner un nouvel instant. Ayant considéré `Instant` comme un objet valeur, et donc non modifiable, ceci n'est pas envisageable. Par ailleurs, retourner une nouvelle valeur correspond mieux à la sémantique de la soustraction.

Étant donnés les paramètres qu'accepte le constructeur et la méthode `totalSeconds`, l'implémentation est directe (listing 1.4).

On peut cependant discuter de ce qui se passe lorsque le deuxième instant est plus grand que le premier. L'implémentation présentée ici renvoie une erreur (`IllegalArgumentException`), puisque on crée un instant avec un nombre négatif de secondes. On pourrait changer les spécifications et autoriser des instants négatifs, ou utiliser la valeur absolue (`java.lang.Math.abs`). Cette dernière approche est intéressante, mais peut dans certains cas être déconcertante, puisqu'alors $t_1 - t_2 = t_2 - t_1$, ce qui est inhabituel pour une différence, qui n'est généralement pas un opérateur commutatif. De plus, une différence est généralement monotone ($t_1 < t_2 \wedge t_3 > 0 \Rightarrow t_1 - t_3 < t_2 - t_3$, la comparaison sera mise en œuvre question 4) ce qui n'est pas le cas si l'on utilise la valeur absolue. En fait, cela dépend de la sémantique exacte de cette méthode. On peut la considérer comme une soustraction, comme c'est le cas précédemment ; elle

retourne un instant (loi de composition interne), mais n'est pas commutative et est monotone. Cependant, vues les restrictions sur les valeurs possibles pour un instant, elle est partielle (non définie sur tout son domaine). On peut aussi considérer que la méthode retourne la durée entre deux instant. Dans ce cas, l'utilisation de la valeur absolue est justifiée, de même que le non respect des propriétés de la soustraction. Cependant, dans ce cas là, retourner un instant peut préter à confusion, puisque c'est en fait une durée, même si les deux types sont isomorphes.

On notera aussi le test à `null`. En effet, lorsqu'une méthode prend en paramètre une instance d'un objet, c'est en fait une référence vers l'instance en mémoire. Contraitement au C (et C++), il n'est pas possible de spécifier en JAVA si les arguments sont passé par valeur (copie) ou par référence ; cela est déterminé par le type de la valeur : par valeur pour les types primitifs, et par référence pour les types objets. C'est le cas dans la majorité des langages objets. Il faut donc envisager la possibilité que la valeur reçue en argument soit la référence nulle (dans les langages où celle-ci existe).

Ici, la méthode retourne l'objet lui-même. On considère en effet ici que `null` est un élément neutre pour la soustraction. Sans ce test, `t1.minus(null)` lèverait une `NullPointerException` lors de l'appel de la méthode `totalSeconds` sur `other`. Ce comportement pourrait être considéré comme valide ; c'est d'ailleur le comportement de beaucoup de méthodes. Cependant, cela laisse la possibilité de nombreuses erreurs qui ne seront découvertes qu'à l'exécution.

Dans tous les cas, il convient de documenter le comportement de la méthodes dans ces deux cas limites. On peut pour cela utiliser `javadoc`, qui permet de générer automatiquement la documentation d'une API (*Application Programming Interface*) à partir de commentaires spécialement formaté.

Question 4 (Comparaison) *:* Écrivez une méthode `compareTo` qui retourne un entier négatif si l'instant considéré est plus petit qu'un instant `t` passé en paramètre, 0 si ces deux instants sont égaux, et un entier positif sinon (cette méthode est définie dans l'interface `Comparable<T>` [1]).

Réponse : Il est classique dans les langages de programmation de disposer d'une fonction ou d'une méthode de comparaison qui retourne un nombre négatif si le premier est inférieur, un nombre positif si le second est supérieur, et 0 si les valeurs sont égales. L'utilisation classique est (ici en JAVA) :

- pour tester $a < b$: `a.compareTo(b) < 0` ;
- pour tester $a \leq b$: `a.compareTo(b) <= 0` ;
- pour tester $a = b$: `a.compareTo(b) == 0` ;
- pour tester $a > b$: `a.compareTo(b) > 0` ;
- pour tester $a \geq b$: `a.compareTo(b) >= 0` ;

La méthode `compareTo` étant définie dans l'interface `Comparable<T>`, on spécifie donc que notre classe l'implémente.

1. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Comparable.html#

D'autres langages, comme PYTHON ou C++, redéfinissent individuellement les six opérateurs de comparaison. Par exemple, en PYTHON, on défini les méthodes `__lt__`, `__le__`, `__eq__`, `__ne__`, `__gt__` et `__ge__` pour représenter respectivement les opérateurs $<$, $\leq$, $=$, $\neq$, $>$ et $\geq$. Bien sûr, pour faciliter l'implémentation, il suffit de définir `__lt__` et `__eq__` et de définir les autres en utilisant celles-ci (p. ex. dans un *mixin*).

Égalité Pour finaliser le fait que notre objet est un objet *valeur*, et pour être cohérent avec la comparaison de deux instant égaux, on redéfini la méthode `equals`[2] pour faire une comparaison par valeur. En effet, la méthode `equals` par défaut (héritée de `Object`) fait une comparaison d'instance (identité), et ne retourne donc vrai que si deux variables représentent la même instance (comparaison par adresse). On veut éviter ici d'avoir `t1.compareTo(t2) == 0` mais `t1.equals(t2) == false`. Il faut donc que le test d'égalité se fasse également sur les valeurs en utilisant `compareTo`[3]. Cet aspect n'est pas propre à JAVA, mais doit être pris en compte dans tous les langages gérant l'égalité et la comparaison de façon similaire. Dans un langage où la comparaison se fait par redéfinition des opérateurs (comme PYTHON décrit plus haut par exemple), ce problème ne se pose évidemment pas, puisqu'il n'existe alors qu'une seule manière de tester l'égalité. De manière générale, il est recommandé lorsque l'on crée des objets valeurs de redéfinir l'égalité pour effectuer une égalité par valeur.

Fonction de hachage Par ailleurs, pour être cohérent et maintenir le bon comportement de notre objet, il faut également redéfinir `hashCode`[4]. Pour rappel, cette méthode retourne un entier « représentatif » de notre objet, qui sera par exemple utilisé comme clé dans une table de hachage. Deux objets considérés égaux par `equals` doivent donc *toujours* avoir la même valeur de hachage. Par défaut, il s'agit de l'adresse mémoire, ce qui est cohérent avec le comportement par défaut de `equals`. Il faut donc toujours redéfinir `hashCode` lorsque l'on redéfini `equals`, afin de calculer la valeur de hachage en fonction des mêmes attributs que ceux utilisés dans la comparaison. Là aussi, cet aspect s'applique à de nombreux langages (par exemple en PYTHON avec la méthode `__hash__`).

Une instance de `Instant` peut désormais être utilisée sans problème dans un `Set` ou comme clé d'une `Map`.

2. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#equals(java.lang.Object)
3. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Comparable.html#compareTo(T)
4. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Object.html#hashCode()

Les *datatypes* tels que l'on vient de mettre en œuvre (non modifiables et comparés par valeur) sont des types particuliers qui sont souvent utilisé. Ils sont à rapprocher des type produits (*tuple* ou *record*) dans les types algébriques. Un certain nombre de langages fournissent donc des constructions spécifiques pour implémenter de tels types sans avoir à mettre en place « à la main » toutes les méthodes précédentes (voir par exemple les `namedtuple`[a] de PYTHON). C'est en fait le cas également de JAVA avec les `records`[b], introduits dans la version 16 du langage.

a. https://docs.python.org/fr/3/library/collections.html#collections.namedtuple
b. https://openjdk.java.net/jeps/395

Le diagramme de classe de ces aspects est présenté figure 1.3.

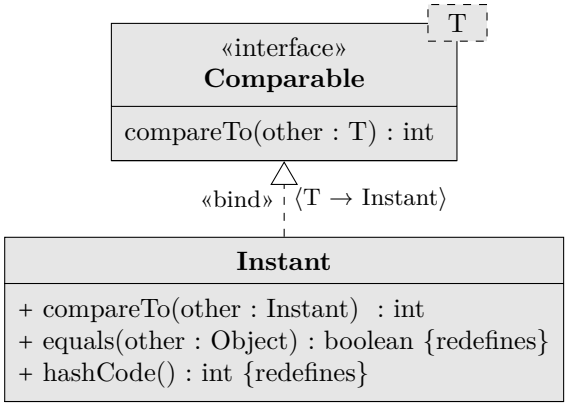

FIGURE 1.3 – Diagramme de classe des aspects de comparaison des instants

Le code Java correspondant est présenté listing 1.5. On exploite ici la méthode statique `Integer.compare`, qui fait exactement ce que l'on veut (retourne −1, 0, 1).

Le test sur la classe réelle de l'instance `other` dans la méthode `compareTo` est là pour gérer l'extensibilité de notre classe `Instant`. En effet, d'un point de vue mathématique, on veut que, pour deux instants a et b, $a < b \Leftrightarrow b > a$ et que $a = b \Leftrightarrow b = a$, et que donc `a.compareTo(b) == -1 * b.compareTo(a)`. On désire également que les opérateurs de comparaison définis via `compareTo` gardent les propriétés habituelles, comme la transitivité ($a > b \wedge b > c \Rightarrow a > c$, etc.) Le problème est que si l'on crée une sous-classe de `Instant` qui redéfini `compareTo`, comme cela sera le cas pour `NULL` et `MAX` dans la suite, `a.compareTo(b)` et `b.compareTo(a)` n'exécuteront pas les mêmes méthodes, et on ne peut donc garantir l'équivalence.

Une première solution est d'empêcher cette configuration, en déclarant la méthode (ou même la classe) finale (mot clé `final`), mais cela limite fortement l'extensibilité.

```
@Override
public int compareTo(Instant other) {
  if (other.getClass() == this.getClass()) {
    return Integer.compare(this.totalSeconds(), other.totalSeconds());
  }
  if (this.getClass().isAssignableFrom(other.getClass())) {
    return -1 * other.compareTo(this);
  }
  throw new ClassCastException(String.format(
        "Can't compare a %s and a %s",
        this.getClass().getName(),
        other.getClass().getName()));
}

@Override
public boolean equals(Object other) {
  if (other == null) { return false; }
  if (other == this) { return true; }
  if (!(other instanceof Instant)) { return false; }
  return this.compareTo((Instant) other) == 0;
}

@Override
public int hashCode() {
  return Integer.hashCode(this.totalSeconds());
}
```

LISTING 1.5 – Comparaison d'Instant

Pour rappel, une méthode finale ne peut pas être redéfinie, et une classe finale ne peut pas être étendue, et ne peut donc pas avoir de sous-classe. C'est normalement le cas pour les classes *datatype*, pour éviter justement les difficultés que l'on va présenter dans la suite. À fin d'illustration, nous ne considérerons pas cette solution ici.

L'autre solution est de forcer l'utilisation de la méthode redéfinie, ce que fait le code présenté ici. Si les deux instances ont la même classe réelle, retournée par `getClass`, on compare leurs valeurs. Si `other` est une instance d'une sous-classe de la classe de `this`, ce que teste `isAssignableFrom`, on utilise la méthode redéfinie en inversant l'appel. Dans tous les autres cas, par exemple si les deux instants sont instances de deux sous-classes d'`Instant` différentes, on considère qu'elle sont incomparables en levant une exception.

Cet exemple illustre la difficulté d'implémenter correctement une classe prévue pour être étendue par héritage et redéfinition, notamment lorsque l'on veut être en mesure de garantir certains invariants, même entre un type et son sous-type pour garantir la substituabilité. C'est une des raisons qui fait souvent préférer la composition et délégation à l'héritage.

Pour plus de détails, voir notamment Bloch, 2018, Items 10, 11 et 14.

Objet Null Dans le cadre d'objets valeur, il est souvent utile de définir un objet *nul*, sous-type de l'objet en question, n'ayant qu'une seule instance (*singleton*). On peut définir une méthode `isNull`, qui retourne `false` pour les instances normales et `true` pour l'instance nulle.

Ce type d'objet redéfini souvent les comparaisons, notamment dans le cas particulier d'instances maximales et minimales (ici p. ex. pour être inférieur à tout autre `Instant`), utiles dans différentes manipulations des listes de valeurs. On peut aussi définir des comportements particulier, comme être le *zéro* ou l'*élément neutre* d'une opération algébrique.

Même dans le cas d'objets « normaux », et même non comparables, il est intéressant de définir un objet *nul*. En effet, il peut être utilisé comme valeur de retour dans le cas de méthodes ou fonctions partielles, c'est-à-dire pouvant échouer (p. ex. méthodes de recherche dans une structure) à la place d'une exception ou de la valeur `null`, qui cause une erreur si elle n'est pas convenablement gérée. On s'approche ici du type `Option` (ou `Maybe`) que l'on trouve dans certain langage, surtout fonctionnels, comme Caml, Scala, Haskell, etc. Pour les curieux, ce type est une *monade*.

Le plus simple est de passer par une classe anonyme, qui permet de n'avoir qu'une instance de notre objet nul facilement, stockée dans un attribut statique final, plutôt que de créer une sous-classe séparée, comme montré en listing 1.6 page ci-contre

```
public static final Instant NULL = new Instant() {
  @Override
  public boolean isNull() { return true; }
  @Override
  public String toString() { return "Instant.NULL"; }
  @Override
  public int compareTo(Instant other) {
     return this == other ? 0 : -1;
  }
};
```

Listing 1.6 – Valeur nulle d'instant.

Les classes anonymes permettent de définir des classes « à la volée » au moment de leur instanciation : on n'aura donc qu'une seule instance de la classe en question. Ici, le code pour NULL serait globalement équivalent à :

```
class Instant {
  private static final class NullInstant extends Instant {
    @Override
    public boolean isNull() { return true; }

    @Override
    public String toString() { return "Instant.NULL"; }
    @Override
    public int compareTo(final Instant other) {
       return this == other ? 0 : -1;
    }
  }
  public static final Instant NULL = new NullInstant();
  //...
}
```

tout en interdisant la création de toute autre instance de NullInstant. La classe anonyme sera compilée sous Instant$1.class.

Valeurs extrêmes Dans notre cas précis, les bornes sont définies (entre 0 et 24h), mais il peut quand même être intéressant de créer des objets limites, sur le même schéma que l'objet nul, ne serait-ce que pour en faciliter l'utilisation, augmenter l'encapsulation de notre classe, et distinguer une valeur nulle (non trouvée) d'une

```java
public static final Instant MAX = new Instant() {
  @Override
  public int totalSeconds() {
    return MAX_TOTALSECONDS;
  }
  @Override
  public int compareTo(Instant other) {
     return this == other ? 0 : 1;
  }
  @Override
  public String toString() { return "Instant.MAX"; }
};

public static final Instant MIN = new Instant() {
  @Override
  public int totalSeconds() {
    return MIN_TOTALSECONDS;
  }
  @Override
  public int compareTo(Instant other) {
     return this == other ? 0 : -1;
  }
  @Override
  public String toString() { return "Instant.MIN"; }
};
```

Listing 1.7 – Valeurs extrèmes d'instants.

valeur minimale ou maximale. Ceci sera particulièrement utile pour les questions 6 et 7. Ces deux valeurs sont définies en listing 1.7.

Pour le `MAX`, on passe aussi par un attribut statique. Dans l'état actuel des choses, on ne peut pas simplement faire `public static final Instant MAX = new Instant(24,0);` car alors on passe par un constructeur qui peut lever une exception, difficile à gérer à ce niveau de définition de la classe (on n'est pas dans une méthode...). On crée ici une classe anonyme qui redéfini `compareTo` pour être supérieure à tout autre `Instant`.

Pour le `MIN`, on aurait put créer directement une instance dans une variable statique publique de la classe `Instant`, initialisée à 0 (valeur par défaut), exploitant ainsi le fait que nos `Instant` sont naturellement bornés. Cependant, la même approche que pour `MAX` permet d'être indépendant des bornes choisies. La classe sera donc plus facile à faire évoluer (intervalles négatifs, ou de plus de 24 heures) ; il suffit de

```java
package td.instants;

public final class MainInstant {
  private MainInstant() { }
  public static void main(final String[] args) {
    Instant[] instants = {
      Instant.NULL,
      Instant.MAX,
      Instant.of(23, 59, 59),
      Instant.of(0, 10, 5),
      Instant.of(0, 0, 0),
      Instant.of(2, 0, 10),
      Instant.of(0, 5, 0)
    };

    for (Instant e : instants) {
      for (Instant f : instants) {
        System.out.format("%s compare %s = %d%n",
                          e, f, e.compareTo(f));
      }
    }
  }
}
```

LISTING 1.8 – Méthode principale.

changer les variables statiques MAX_SECONDS ou MIN_SECONDS, les instances spéciales MIN et MAX garderont leurs propriétés.

De manière générale, l'objet nul ou les valeurs extrêmes sont des application du patron de conception *special case*, décrit notamment dans FOWLER, 2002. Ce type de valeurs spéciales est similaire à ce que l'on retrouve pour les nombres réels avec *NaN* et *Infinity*.

Question 5 *:* Ajoutez une méthode `main` à votre programme. Cette méthode `main` crée un tableau d'objets de la classe `Instant` et affiche le résultat de la comparaison de tous les couples possibles.

Réponse : La méthode `main` est assez directe, et est donnée en listing 1.8.

On constate que l'on a bien le comportement attendu pour NULL et MAX : le premier est inférieur à tout instant (sauf lui-même) et le deuxième est supérieur à tout instant (sauf lui-même).

```java
public static Instant getNext(Instant searched, Iterable<Instant> instants) {
  Instant result = null;
  for (Instant current  : instants) {
    if (current == null) { continue; }
    if ((searched == null || current.compareTo(searched) > 0)
        && (result == null || current.compareTo(result) < 0)) {
      result = current;
    }
  }
  return result;
}
```

LISTING 1.9 – Implémentation de la recherche du suivant.

Question 6 *:* Soit $u = (u_0, \ldots, u_n)$ une liste de $n+1$ instants. Écrivez un algorithme qui retourne le plus petit instant u_i de la liste qui est strictement supérieur à un instant t donné en paramètre.

Réponse : Pour cela, on parcour la liste en stockant la valeur si elle répond aux critères. L'algorithme est donné algorithme 1.1, et son implémentation en listing 1.9.

Données : un instant t, une liste d'instants l
Résultat : le plus petit instant de l supérieur à t
début
  $val \leftarrow \varnothing$
  pour chaque $i \in l$ **faire**
    si $i > t \wedge (val = \varnothing \vee i < val)$ **alors**
      $val \leftarrow i$
  retourner val

ALGORITHME 1.1 – Recherche du suivant.

On considère que la méthode `getNext` peut prendre un instant `null` en paramètre. Dans ce cas, elle retourne le minimum de la liste. Ceci facilitera la question suivante.

On aurait pu envisager de modifier `compareTo` pour accepter un paramètre nul, mais dans ce cas, on ne respecterait plus le contrat de l'interface, qui spécifie que la méthode doit dans cas lever l'exception `NullPointerException`, qui est utilisée lorsque l'on essai d'appeler une méthode sur un objet `null`. On remarque d'ailleurs que comme les opérateurs booléens sont « paresseux », on ne risque pas d'appeler `compareTo` sur `null` car si `searched` ou `result` sont `null`, le « ou » ne teste pas la deuxième possibilité, et n'appelle donc pas la méthode.

```java
public static void sort(List<Instant> values) {
  Instant next = null;
  for (int i = 0; i < values.size() - 1; i++) {
    next = getNext(next, values.subList(i, values.size()));
    values.remove(next);
    values.add(i, next);
  }
}
```

LISTING 1.10 – Implémentation du tri.

Question 7 : Soit $u = (u_0, \ldots, u_n)$ une liste de $n+1$ instants. Écrivez un algorithme qui trie la liste u, en utilisant l'algorithme précédent. On supposera que la liste ne contient pas deux fois le même élément.

Réponse :

Un algorithme de tri simple utilisant la fonction précédente est donné en algorithme 1.2, et sa mise en œuvre en JAVA est présentée listing 1.10.

Données : une liste d'instants l
Résultat : la liste l triée
début
  $t \leftarrow \varnothing$
  pour $i \leftarrow 0$ **à** *taille(l)* **faire**
    $t \leftarrow$ suivant(t,l)
    supprimer(l, t)
    ajouter(l,i,t)

ALGORITHME 1.2 – Tri d'une liste d'instants.

Il est possible d'optimiser cet algorithme en ne recherchant le suivant que dans la sous-liste suivant la position courante (voir `subList` [5]).

On note que les deux méthodes précédentes pourraient être utilisées sur n'importe quel objet ayant une méthode `compareTo`. En effet, dans `getNext`, c'est la seule méthode appelée sur les éléments de la liste ou celui recherché. Il n'est donc pas nécessaire que ce soient des `Instant` puisque n'importe quel objet implémentant `Comparable` pourrait être utilisé.

On pourrait donc réécrire ces méthodes avec une signature plus générique, ce qui permettrait de les réutiliser avec des listes d'entiers ou de chaînes de caractères par exemple.

5. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/List.html#subList(int,int)

```java
public static <T extends Comparable<T>> T getNext(T searched, ↵
    Iterable<T> iter) {
  T result = null;
  for (T current  : iter) {
    if (current == null) { continue; }
    if ((searched == null || current.compareTo(searched) > 0)
        && (result == null || current.compareTo(result) < 0)) {
      result = current;
    }
  }
  return result;
}

public static <T extends Comparable<T>> void sort(List<T> lst) {
  T next = null;
  for (int i = 0; i < lst.size() - 1; i++) {
    next = getNext(next, lst);
    lst.remove(next);
    lst.add(i, next);
  }
}
```

Listing 1.11 – Version générique du tri.

Ici, le type des éléments comparés est `T` où `T` est n'importe quel type comparable avec lui-même, c'est-à-dire implémentant `Comparable<T>`

Glossaire

abstraction généralisation d'une idée ou d'un concept. C'est une simplification ne gardant que les caractéristiques essentielles des éléments considérés, et s'appliquant à différents exemples concrets. Elle permet de manipuler un seul concept plutôt que différents cas particuliers, et masque les détails sous un concept commun. C'est une forme de généralisation des concepts.

accesseur (aussi *getter* et *setter*) méthode donnant accès à l'état interne d'un objet, c'est-à-dire retournant (ou modifiant) la valeur d'un attribut, éventuellement en vérifiant des pré conditions ou en maintenant des invariants sur l'état. Les méthodes de modification d'état sont aussi appelées mutateurs, l'accesseur ne désignant alors que l'accès en lecture. Cela reste un détail d'implémentation, opaque à utilisation de la classe : du point de vue de l'interface, on ne peut différencier un accesseur d'une méthode calculant une valeur. Leur utilisation doit cependant être limitée et prudente pour éviter de casser l'encapsulation de l'objet.

analyseur syntaxique (parser) TODO

API Interface de programmation d'application. C'est un ensemble de types de données (abstrait) et d'opérations associées, ou d'interfaces dans le monde objet. Elle définit la manière d'interagir avec un système ou une bibliothèque logicielle.

arbre syntaxique abstrait structure de donnée représentant un document ayant une grammaire formelle, par exemple le code source d'un programme, sous forme d'un arbre manipulable. Il est produit par l'analyseur syntaxique.

argument valeur passée à une fonction, procédure ou méthode lors de l'appel (généralement via la pile). Elle sera liée à la variable du paramètre correspondant dans le corps de la fonction. Cette correspondance peut être positionnelle ou par mot-clé. Selon le langage et la nature de l'argument, celui-ci peut être passé par valeur (copie) ou par référence (adresse).

assembleur langage de programmation très proche de la machine, et donc dépendant de l'architecture matérielle utilisée. Les instructions disponibles sont celles implé-

mentées directement dans le processeur. Le code assembleur n'est généralement pas écrit par une personne, mais généré par un compilateur à partir de code source dans un langage plus haut niveau.

attribut (donnée membre, propriété, variable d'instance, champ) donnée (valeur) associée à chaque instance d'une classe ou d'un objet. Comme une variable, un attribut a un nom, et généralement un type associé. L'ensemble des attributs d'un objet constituent sa structure, leurs valeurs définissent sont état.

attribut dérivé attribut dont la valeur est calculée en fonction d'un ou plusieurs autres attributs du même objet. Il est souvent matérialisé par un accesseur qui calcule la valeur à chaque appel. Il peut aussi être stocké comme un attribut classique, mais doit être maintenu à jour si les attributs dont il dépend sont modifiables (dans le mutateur p. ex.)

auto-encapsulation fait d'appliquer l'encapsulation, en particulier le masquage d'information, à la classe elle-même. Concrètement, la classe elle-même n'accède pas directement à son état, mais passe par des méthodes dédiées. Ainsi, les seules méthodes accédant à l'état d'une classe sont les accesseurs. Ceci facilite l'évolution de la classe et permet de garantir la validité de son état.

bibliothèque ensemble de définitions de types, de structures de données et de traitements, généralement concernant un domaine ou une tâche précise, et conçue pour être réutilisable. On parle de bibliothèques standard pour celles fournies avec le langage, et de bibliothèques tierces pour celles développées indépendamment. Elles évitent de développer des fonctionnalités à partir de zéro dans chaque programme, mais créent une dépendance pour les bibliothèques tierces. Une bibliothèque définie souvent un espace de noms pour les éléments qu'elle contient. Contrairement au framework, une bibliothèque ne définie pas l'architecture de l'application, mais plutôt des « briques » élémentaires. *Voir aussi* paquet & gestionnaire de dépendances.

bytecode (ou opcode) ensemble d'instructions de bas niveau, sorte d'assembleur, pour une machine virtuelle. Le bytecode est le résultat de la compilation d'un langage de plus haut niveau d'abstraction. Il est interprété par la machine virtuelle lors de l'exécution du programme.

classe définition abstraite un ensemble d'objets ayant des propriétés communes. Une classe peut être vue comme un modèle d'objet, servant à en dériver des instances. C'est elle qui en spécifie les propriétés caractéristiques. Elle défini donc aussi un type, à la fois son interface et son implémentation.

classe abstraite classe ne pouvant pas être instanciée directement, et devant donc être étendue. Elle contient des méthodes abstraites qui seront redéfinie dans les sous-classes, différant ainsi la définition d'une partie de son implémentation. Elle permet de définir un type abstrait. *Voir aussi* extension.

closure (ou fermeture) fonction capturant son contexte, c'est-à-dire les variables libres (non locales) dans le corps de la fonction qui existaient dans son environnement au moment de sa définition. Les closures sont souvent mises en œeuvre avec les lambda définie dans une fonction génératrice. Une closure peut être vue comme le dual d'un objet.

code natif (ou binaire) instructions directement compréhensibles par le processeur, et donc dépendantes de l'architecture matérielle de l'ordinateur. On parle aussi de langage machine. Elles sont représentées sous forme de codes numériques, et stockées sous forme binaire (par opposition à une forme textuelle). C'est l'équivalent binaire de l'assembleur. Il n'est pas écrit directement, mais généralement produit par un compilateur à partir de code source.

code source description d'un programme informatique écrit dans un langage de programmation. C'est du simple texte et n'importe quel éditeur peut servir à l'écrire, mais un IDE (*Integrated Development Environment*) est souvent utilisé. Le code source sera ensuite convertit par le compilateur en programme exécutable par l'ordinateur (code natif) ou en en *bytecode*, ou directement exécuté par un interpréteur.

coercition fait de forcer le type d'une valeur vers un type différent (cible), sans en changer la représentation interne. Ce changement de type peut être explicite (*cast*) ou implicite lors de l'affectation ou utilisation comme argument où le type cible est requis. Un cas est l'utilisation de type incomparables mais ayant la même représentation interne (`int` et `char` en C par exemple). Ici, le changement concerne la manière d'interpréter la représentation en mémoire à l'exécution. L'autre cas courant est l'interprétation comme une instance d'un super-type (coercition ascendante ou *upcast*) ou d'un sous-type (coercition descendante ou *downcast*). Dans ce cas, le changement ne concerne que le système de vérification de type au moment de la compilation. *Voir aussi* transtypage.

cohésion degré de liaison des composants internes d'un système unique (membres d'une classe, classes d'un paquet, etc.) On cherche en général à la maximiser. S'oppose au couplage.

compilateur programme effectuant la compilation du code source.

compilation transformation d'un artefact écrit dans un langage (généralement du code source) en un autre utilisant un langage ayant un plus bas niveau d'abstraction (généralement du code natif). *Voir aussi* interprétation.

complexité (algorithmique) analyse des ressources nécessaires à un algorithme proportionnellement à la taille des données à traiter. Cela peut être le temps, la quantité de mémoire, le nombre d'opérations, etc. Ne pas confondre avec la complexité d'un système.

complexité (système) fait d'être constitué d'un grand nombre d'éléments en interaction dont les comportements s'influencent mutuellement. Un système complexe

a donc un comportement difficile à prévoir. S'oppose à simple, différent de compliqué ou difficile. Ne pas confondre avec la complexité algorithmique.

comportement façon dont un objet agit et réagit, du point de vue de ses changements d'état et de l'envoie de message. Généralement mis en œuvre par des méthodes.

composition TODO

constructeur méthode initialisant les nouvelles instances d'une classe. Généralement appelé automatiquement par le processus d'instanciation. Selon le langage, il se nomme comme la classe (en JAVA, C++) ou avec un nom prédéfini (`__init__` en PYTHON, `constructor` en ECMASCRIPT. *Voir aussi* destructeur.

couplage degré de liaison ou de dépendance entre éléments de composants différents. Représente donc la dépendance d'un élément vis-à-vis de l'extérieur, de son contexte. C'est l'inter dépendance des composants, leur degré d'interaction. On cherche en général à le minimiser, car c'est l'une des sources de la complexité des systèmes. S'oppose à la cohésion. On parle de couplage *abstrait* lorsque celui-ci se fait sur un élément abstrait, comme une interface.

délégation mécanisme par lequel un objet fait suivre, ou délègue, un traitement à un autre objet. Ce peut être une alternative à l'héritage, ou même à l'organisation en classes dans les langages à prototype (comme SELF ou ECMASCRIPT). La délégation permet de partager le comportement défini dans un objet sans introduire de sous-type ni d'héritage. On parle de réutilisation « boite noire », car le fonctionnement interne de l'objet n'est jamais exposé (voir aussi composition) Elle est plus flexible et plus dynamique que l'héritage. Elle peut être simulée manuellement, ou gérée nativement par le langage. Voir LIEBERMAN, 1986

dynamique se dit d'une propriété ou valeur qui n'est connue, fixée ou définie qu'au moment de l'exécution, ou qui peut changer au cours de l'exécution du programme (*run time*). S'oppose à statique

effet de bord modification de l'environnement par l'exécution d'une opération. Une opération agit par effet de bord notamment lorsqu'elle modifie une variable non locale, une donnée extérieure passée par référence, ou effectue une entrée/sortie. *Voir aussi* transparence référentielle & procédure.

encapsulation regroupement dans une même entité des données sémantiquement liées, et des traitements qui s'y rapportent. Liée au masquage d'information.

espace de noms contexte pour les éléments ou identifiants qu'il contient. Il permet notamment d'éviter les collisions dans les noms de ceux-ci et les ambiguïtés qui pourrait en découler. Les espaces de noms peuvent parfois être utilisés de manière récursive, c'est-à-dire qu'un espace peut lui-même être contenu dans un espace de noms.

exception erreur imprévue qui surgit dans un programme. Une exception est *levée* au point d'erreur et se propage dans la pile d'appel pour être traitée à un

autre endroit du programme, interrompant le flot normal d'exécution. C'est généralement un objet à part entière.

expression élément du langage (ensemble d'opération, valeurs, appels de fonctions) ayant une valeur. Une expression peut donc être affectée à une variable, passée en argument d'une fonction, etc. S'oppose à instruction.

extension principe permettant de définir un nouveau type en le faisant dériver d'un super-type. Le nouveau type possède donc implicitement toutes les propriétés de son super-type, ainsi que certaines propriétés propres. L'extension peut-être facilitée par l'héritage de, ou la délégation vers, des types (partiellement) concrets.

fermeture voir *closure*.

fonction opération retournant une valeur comme résultat. C'est donc une expression. Idéalement, elle ne doit pas avoir d'effet de bord, afin d'être référentiellement transparent (cf. transparence référentielle). On parle alors de *fonction pure*. *Voir aussi* procédure.

fonction anonyme fonction n'étant pas associée à un identifiant. Elle est la plupart du temps utilisée comme argument d'une fonction d'ordre supérieur. Elle représente l'équivalent pour les type fonctionnel des notations littérales pour les types de données. *Voir aussi* objet fonction.

fonction d'ordre supérieur fonction manipulant des fonctions. Cela peut être une fonction dont le comportement est spécifié par un traitement passé en paramètre sous la forme d'un objet fonction ou une fonction créant des fonctions (souvent des *closure*s).

framework ensemble de classes, structures de données, et fonctions, définissant une architecture réutilisable pour une famille de logiciel. Contrairement aux bibliothèques, il définie le comportement général, abstrait, de l'architecture. Seul le comportement spécifique à une application particulière devra être défini au cas par cas. Cette spécialisation peut se faire par extension de classes abstraites du framework, par composition et injection de dépendances à la configuration, etc.

généralisation TODO S'oppose à la spécialisation.

getter voir accesseur.

héritage accès par les instance de la sous-classe aux propriétés définies dans sa super-classe. Permet donc de définir une nouvelle classe en dérivant une ou plusieur super-classe. L'héritage facilite l'extension et le sous-typage en faisant de la réutilisation. On parle de réutilisation « boite blanche », car la sous-classe est liée au fonctionnement interne de la super-classe. La spécialisation de la sous-classe peut nécessiter de redéfinir certaines méthodes héritées (voir redéfinition). L'héritage est transitif. Alternative à la délégation.

IDE (*Integrated Development Environment*) environnement de développement, intégrant divers outils d'aide à la programmation et au génie logiciel.

identifiant nom symbolique associé à une entité du programme (fonction, variable, type) pour le désigner et le manipulé. Il est généralement associé à une valeur (voir liaison).

implémentation code effectif d'une structure ou d'une opération. L'implémentation doit au moins contenir le code associé aux méthodes spécifiées dans l'interface associée au type concret. S'oppose à l'interface.

instance l'instance d'une classe est une valeur dont le type est la classe. C'est un des éléments de l'ensemble décrit par la classe. Elle contient les valeurs associées aux attributs définis dans la classe correspondante, qui représentent son état, et en expose les méthodes.

instanciation processus de création de nouveaux objets (généralement instances d'une classe). Elle se décompose en deux parties : l'allocation mémoire et l'initialisation. L'allocation se fait via l'opérateur `new` (Java, ECMAScript) ou par appel d'une méthode de classe (`___new___` en Python, `new` en Smalltalk. L'initialisation se fait via le constructeur.

instruction élément du langage (opérations, structures de contrôle, appels de fonctions) ayant un effet sur l'état du système. S'oppose à expression.

interface (concept) L'interface d'une classe, d'un type ou d'un module est l'ensemble des propriétés misent à disposition. Elle définie la manière dont on peut interagir avec les instances de ce type. S'oppose à l'implémentation.

interface (type) élément du langage définissant un type purement abstrait, sous la forme d'un ensemble de signatures de méthodes. Une interface ne peut pas contenir d'attributs, ceux-ci étant du domaine de l'implémentation.

interface fonctionnelle interface ne spécifiant qu'une seule méthode, et représentant donc un objet fonction. Dans les langages disposant de lambda, celles-ci peuvent être utilisées comme instances d'interfaces fonctionnelles. Équivalent des type fonctionnels. *Voir aussi* SAM.

interprétation exécution directe d'un langage plus haut niveau que du code natif (code source, arbre syntaxique abstrait (AST pour *abstract syntax tree*) ou représentation intermédiaire, *bytecode*) sans passer par une phase de compilation. On parle aussi d'évaluation dynamique. *Voir aussi* interpréteur.

interprète voir interpréteur

interpréteur (parfois interprète) programme effectuant l'interprétation d'un langage de haut niveau, par opposition à un compilateur.

javadoc outil de génération automatique de documentation à partir du code source et de commentaires spécialement formatés pour le langage Java.

lambda voir fonction anonyme

liaison (*binding*) association d'une valeur (ou d'une expression) à un identifiant, comme une variable et sa valeur, ou un appel de méthode au code effectif de l'opération. Les liaisons statiques (ou *immédiates*) sont résolues à la compilation ; les liaisons dynamiques (ou *différées*, *tardives*) sont effectuées à l'exécution (*late binding*).

machine virtuelle environnement d'exécution de *bytecode* jouant le rôle d'abstraction de l'infrastructure matérielle réelle.

masquage d'information le fait de ne rendre accessible que le nécessaire aux utilisateurs d'un composant, en masquant les détails d'implémentation ou les caractéristiques non essentielles (état, implémentation des méthodes, ...). Les éléments masqués ne sont pas accessibles à l'extérieur de l'objet. La représentation interne de l'objet (son état) ne peut donc être modifiée que par l'intermédiaire des méthodes exposées, et donc de manière contrôlée. Cela permet de diminuer le couplage entre composant. Lié à l'encapsulation. Voir notamment PARNAS, 1972.

membre voir propriété

méta-classe classe d'un objet classe. Dans certains langages, les classes sont elles-mêmes des objets, et peuvent être manipulées comme des valeurs (passage en argument, envoie de message, etc.). Elles sont donc les instances d'une classe particulière : la méta-classe. En JAVA, c'est la classe `Class` qui fixée et finale. En PYTHON, c'est `type` qui peut être étendue et changée pour chaque classe. En SMALLTALK, chaque classe a sa propre méta-classe (elle-même sous-classe de la classe `Class`). Étendre et redéfinir la méta-classe permet de spécialiser le comportement des classes elles-mêmes. Ainsi, les propriétés statiques ou de classes peuvent être vue comme des membres d'instances de la méta-classe (c'est exactement le cas en SMALLTALK). *Voir aussi* méta-programmation.

méthode procédure ou fonction associée à un objet (généralement via sa classe) et invoquée par envoie de message. La résolution est dynamique en fonction de l'objet receveur du message. Celui-ci sera le premier argument (souvent implicite) de l'appel de la méthode.

méthode abstraite méthode n'ayant pas de code associé, pas d'implémentation. Seule sa signature est spécifiée. Une méthode abstraite ne peut exister que dans une classe abstraite. Les sous-classes de la classe abstraite devront définir une implémentation pour cette méthode, ou être elles-mêmes abstraites. On parle aussi de méthode virtuelle pure, ou différée (voir redéfinition).

méthode de classe méthode dont le receveur est la classe elle-même, vue comme une instance de la méta-classe, et non une instance de la classe. Cette notion n'existe pas en JAVA. Ne pas confondre avec les méthodes statiques.

méthode statique méthode qui peut être appelée hors d'un contexte objet ; elle n'a donc pas de receveur, et la liaison est par conséquent statique. Elle est cependant définie dans l'espace de noms de la classe. Pour définir une méthode statique beaucoup de langages utilisent le mot-clé `static`.

mutabilité capacité pour une valeur à être modifiée, ou de changer d'état dans le cas d'une structure de donnée plus complexe. On parle d'immutabilité dans le cas contraire. La contrainte peut être associée au type lui-même (p. ex. type valeur), à une instance particulière (on la qualifie parfois de « gelée » ou *frozen*), ou à une référence vers une instance (comme c'est le cas en RUST par exemple). Il est souvent plus simple de gérer des valeurs ou références non modifiables.

mutateur (*setter*) méthode permettant de modifier l'état interne d'un objet. *Voir aussi* accesseur.

objet entité autonome regroupant un état (des données) et les traitements permettant de manipuler cet état. Peut être vu comme l'encapsulation d'une structure de donnée et des opérations qui s'y rapportent.

objet fonction valeur de premier ordre représentant la réification d'un traitement. Dans les langages fonctionnel, les fonctions elles-mêmes sont des valeurs de premier ordre. Les fonctions anonymes (lambda) sont aussi des objets fonction. Dans les langages objet, on utilise généralement des instances d'interfaces fonctionnelles ou de *single abstract method* (SAM).

OMG (Object Management Group) organisme international de standardisation des technologies objet et d'intégration en entreprise.

opaque se dit d'un élément ou d'un composant que l'on utilise sans en connaitre le fonctionnement ou la sémantique interne. On parle d'utilisation en « boite noire ». Cet aspect dépend du point de vue, un élément opaque pour un acteur (utilisateur) peut ne pas l'être pour un autre (celui qui le définie). Différent d'obscur, incompréhensible. S'oppose à transparent.

opération abstraction représentant un traitement réutilisable. *Voir aussi* fonction, procédure & méthode.

paquet (package) regroupement de plusieurs classes et interfaces, ou structures de données et opérations associées. Ce regroupement a un but sémantique et organisationnel. Il définie également un espace de noms. En JAVA, il est déclaré avec le mot-clé `package`.

paramètre Variable libre dans le corps d'une fonction, procédure ou méthode, dont la valeur sera spécifiée à l'appel par passage d'argument. De manière plus générale, c'est un élément d'information utilisé dans la description d'une entité (type, structure, traitement) qui sera spécifié au moment de l'évaluation.

pointeur valeur représentant l'adresse mémoire d'une valeur et donc directement manipulable.

polymorphisme (lit. plusieurs formes) possibilité de voir un objet de plusieurs façons, comme appartenant à plusieurs types à la fois, ou pour une opération de s'appliquer à plusieurs types. Il existe différentes formes de polymorphisme (voir notamment transtypage, redéfinition, surcharge, type générique, sous-typage).

procédure opération fonctionnant par effet de bord, et ne retournant pas de valeur. Un appel de procédure est donc une instruction et non une expression. *Voir aussi* fonction.

propriété donnée (attribut) ou comportement (méthodes) associée à un objet. Les propriétés sont généralement définie dans la classe correspondante.

receveur (cible) objet recevant le message. C'est celui sur lequel on appelle la méthode, dont il sera le premier argument (souvent implicite). Dans le corps de la méthode, il est généralement désigné par `this` ou `self`.

redéfinition (*overriding*) implémentation alternative d'une méthode déjà définie dans une super-classe, et donc héritée, spécifiée dans la sous-classe. La redéfinition masque la méthode héritée. Cela permet de fournir une implémentation spécifique au sous-type, et permet donc une forme de polymorphisme *ad hoc*. La résolution se fait en fonction du receveur et est dynamique.

référence adresse de l'espace mémoire où est stockée une valeur (généralement dans le tas). Contrairement à un pointeur, une référence n'est pas manipulable directement. Une variable contenant la référence devient un alias pour la valeur, cette dernière n'est pas copiée. En JAVA, comme dans beaucoup de langages haut niveau où la mémoire est gérée, les types complexes ne sont accessibles que par leur références.

résolution (de méthode) processus permettant d'associer un appel symbolique particulier (envoie de message) à la méthode concrète à exécuter. La recherche de l'implémentation dépend de la signature de la méthode, de ses arguments et de l'objet receveur. Elle peut être statique ou dynamique. La recherche peut suivre l'arbre d'héritage ou le graphe de délégation si la méthode n'est pas définie dans le receveur. C'est un des éléments permettant le polymorphisme *ad hoc*, notamment la surcharge et la redéfinition. Le terme s'utilise aussi hors du contexte du paradigme objet, lors de la surcharge notamment. *Voir aussi* liaison.

SAM (*single abstract method*) classe abstraite ayant une seule méthode abstraite. Ces classes sont la version objet des fonctions comme valeur de premier ordre de la programmation fonctionnelle, la méthode abstraite représentant l'opération à effectuer. Équivalent aux interfaces fonctionnelles.

setter voir mutateur.

signature (de fonction ou méthode) ensemble des types des paramètres et de retour d'une opération (y compris les erreurs et exceptions) définissant ainsi son propre

type. On parle aussi de type fonctionnel. C'est notamment ce qui est utilisé dans la résolution. Le nom de l'opération est parfois inclus dans la signature.

simple (système) S'oppose à complexité, différent de facile.

sous-classe classe définissant un sous-type pour autre classe, sa super-classe, généralement par le biais de l'héritage de ses propriétés. On parle aussi de classe fille, classe enfant, ou classe dérivée. La sous-classe peut ajouter ses propres propriétés à celles de sa super-classe ; on parle ainsi d'extension.

sous-typage relation (*is a*) définissant un sous-type. Le sous-typage peut être structurel ou nominal. Il permet le polymorphisme d'inclusion. *Voir aussi* substituabilité & spécialisation.

sous-type type défini tel que les instances d'un sous-type sont des formes spécialisées du super-type (relation de sous-typage). D'un point de vue ensembliste, l'ensemble des instances du sous-type est inclus dans l'ensemble des instances du super-type. Elles peuvent donc être utilisées à la place d'instances du super-type.

spécialisation TODO S'oppose à la généralisation

statique se dit d'une propriété ou valeur qui est connue, définie ou fixée au moment de la compilation (*compile time*), et donc inchangée au cours de l'exécution du programme. S'oppose à dynamique.

substituabilité (ou remplaçabilité). fait de pouvoir remplacer une instance d'un type par une instance d'un autre type sans changer le comportement général du programme. Définie de fait une relation de sous-typage entre les deux types concernés. Voir Liskov, 1987.

super-classe classe servant de super-type à une ou plusieurs autres classe, ses sous-classes. On parle aussi de classe mère ou de classe parente.

super-type voir sous-type.

surcharge (*overloading*) fait d'associer plusieurs opérations de type fonctionnel différents à un même identifiant. La surcharge d'une fonction, d'un opérateur, ou d'une méthode consiste à déclarer une fonction plusieurs fois, avec des paramètres différents, dans le même espace de noms (module ou classe), ou de les importer dans un espace de nom commun si la résolution est dynamique. C'est un des éléments du polymorphisme *ad hoc*. La redéfinition peut être considérée comme un type de surcharge (au sens large), mais le terme désigne plutôt la surcharge *paramétrée*, où la résolution se base uniquement sur la signature de la fonction. Dans beaucoup de langages, la résolution de la surcharge est statique.

transparence référentielle fait pour une expression de pouvoir être remplacée par sa valeur sans altérer le comportement du programme. Cette propriété permet au compilateur ou à l'interpréteur de faire certaines optimisations. Elle facilite aussi le raisonnement sur l'exactitude du programme. La transparence référentielle est garantie dans les langages fonctionnels purs. Elle est en effet incompatible avec les effets de bord visibles.

transparent se dit d'un composant invisible, dont la présence n'est pas perceptible à l'utilisation. Différent de clair, évident. S'oppose à opaque.

transtypage conversion d'une valeur d'un certain type en une valeur équivalente d'un autre type (cible). Cette conversion est généralement automatique et implicite, et implique un changement dans la représentation interne de la valeur. Par exemple, la conversion d'une valeur de type entier `int` en `long`, ou même en `float`, lors de son affectation à une variable déclarée du type cible, ou lors d'une utilisation comme argument dans un appel de fonction acceptant le type cible. Il permet une forme de polymorphisme.La coercition est parfois considérée comme du transtypage, bien qu'elle n'en soit pas au sens strict.

type ensemble de valeurs partageant une structure et un comportement commun.

type abstrait formalisation théorique d'un type de donnée, décrit par les valeurs possibles qu'il peut prendre et par son comportement, et donc les opérations pouvant être y appliquées, ainsi que les pré et post conditions et invariants qui doivent être respectés (son interface). C'est une description abstraite, indépendante d'une quelconque implémentation. Voir notamment Liskov et Zilles, 1974. Dans les langages objets, mis en œuvre par une classe abstraite ou une interface.

type fonctionnel type d'un objet fonction, représenté par la signature des fonctions instances, dans les langages fonctionnels, où les fonctions sont des valeurs de premier ordre. Souvent présent dans la signature des fonction d'ordre supérieur.

type générique type complexe dont le type d'un ou plusieurs constituant n'est pas fixé *a priori*, mais défini à l'instanciation ou à l'extension. Le type est donc un paramètre. Permet un type de polymorphisme universel.

UML (*Unified Modeling Language*) langage graphique de modélisation de systèmes, standardisé par l'OMG (Object Management Group).

Bibliographie

Bloch, J. (jan. 2018). *Effective Java*. 3e éd. Addison-Wesley, p. 412. isbn : 978-0134685991 (cf. p. 2, 8, 14).

Fowler, M. (nov. 2002). *Patterns of Enterprise Application Architecture*. Signature Series. Addison-Wesley, p. 560. isbn : 978-0321127426 (cf. p. 2, 17).

Lieberman, H. (1986). « Using Prototypical Objects to Implement Shared Behavior in Object-Oriented Systems ». In : *Conference Proceedings on Object-Oriented Programming Systems, Languages and Applications*. OOPSLA '86. Portland, Oregon, USA : Association for Computing Machinery, p. 214-223. isbn : 0897912047. doi : 10.1145/28697.28718 (cf. p. 25).

Liskov, B. (1987). « Data abstraction and hierarchy ». In : *Proc. on Object-oriented programming systems, languages and applications (Addendum)*. Orlando, Florida, USA : ACM, p. 17-34. isbn : 0-89791-266-7. doi : 10.1145/62138.62141 (cf. p. 31).

Liskov, B. et S. Zilles (1974). « Programming with Abstract Data Types ». In : *Proceedings of the ACM SIGPLAN Symposium on Very High Level Languages*. Association for Computing Machinery, p. 50-59. isbn : 9781450378840. doi : 10.1145/800233.807045 (cf. p. 32).

Parnas, D. L. (1972). « On the Criteria to Be Used in Decomposing Systems into Modules ». In : *Communication of the ACM* 15.12, p. 1053-1058. issn : 0001-0782. doi : 10.1145/361598.361623 (cf. p. 28).