TP
Programmation orientée objets

Yannick Loiseau yannick.loiseau@uca.fr

Licence 2me année
version du 10 mars 2022

TP 1

Échauffement

1.1 Introduction

Exercice 1 (Premiers pas en Java)

Question 1 (La base) *:* Créez un fichier Hello.java contenant un programme Java qui affiche Hello World ! à l'écran.

> Pour afficher, utilisez la méthode println [a] de System.out [b].
>
> a. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/PrintStream.html#println(java.lang.Object)
>
> b. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/System.html#out

Réponse : Le programme principal en Java est représenté par une classe publique ayant une méthode statique `public static void main(String[] args)`. Le paramètre `args`, qui est un tableau de chaînes de caratères, contient les arguments passés en ligne de commande lors de l'exécution du programme. Comme toute classe publique, elle doit se trouver dans un fichier ayant le même nom que celle-ci. Notre classe `Hello` est donc très simple, le code est présenté listing 1.1 page suivante.

Les entrées–sorties passent par les trois attributs statiques de la classe `System` :

- `in` pour l'entrée standard (par défaut le clavier),
- `out` pour la sortie standard (par défaut la console), où l'on affiche les messages,
- `err` pour l'erreur standard (par défaut la console également), où l'on affiche les messages d'erreur.

Lorsque l'on crée des classe purement statiques, comme c'est le cas ici, il est cependant recommendé de les rendre finales et de créer un constructeur privé, afin de garantir leur utilisation comme module plutôt que comme classe. En effet, ce genre de classe ne sert que de conteneur pour des méthodes (en fait des fonctions puisqu'elles

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

LISTING 1.1 – Code minimal pour la classe Hello

```java
public final class Hello2 {

  private Hello2() {
    throw new UnsupportedOperationException("Don't instantiate utility ↩
      classes");
  }

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

LISTING 1.2 – Code étendu pour la classe Hello

ne s'appliquent pas directement à un objet) en fournissant un espace de nom, et donc sont conceptuellement des modules. Le code étendu est donné listing 1.2.

Question 2 (Compilation) *:* Compilez à l'aide de la commande `javac Hello.java`. Quel est le nom du fichier généré ? Que contient-il ?

Réponse : La compilation de la classe crée un fichier `Hello.class`. Vous pouvez regarder son contenu avec la commande `javap -c Hello.class`. Il s'agit de *bytecode*, c'est-à-dire d'instructions bas niveau pour la JVM (Java Virtual Machine).

Question 3 (Exécution) *:* Exécutez ce programme à l'aide de la commande `java Hello`.

Question 4 (Organisation) *:* Créez un répertoire `src` et déplacez-y votre fichier source. Créez un repertoire `build` et essayez à nouveau de compiler, avec `javac -d build src/*.java`. Quel est l'intérêt de l'option `-d`.

Ajoutez la ligne `package poo.tp.premierspas;` au début de votre programme et recommencez. Que ce passe-t-il ?

Essayer de l'exécuter à nouveau. Vous devrez peut-être utiliser l'option `-cp` de `java`. Regardez son utilité.

Réponse : L'instruction `package` place votre classe dans un paquet. L'option `-d` de `javac` place le fichier généré dans le bon répertoire, correspondant au sous-module (ici `build/poo/tp/premierspas/Hello.class`).

```java
package poo.tp.premierspas;

public final class Hello3 {

  private Hello3() {
    throw new UnsupportedOperationException("Don't instantiate utility ↩
      classes");
  }

  public static void main(String[] args) {
    if (args.length < 1) {
      System.err.println("Give me your name!");
      System.exit(1);
    } else {
      System.out.printf("Hello %s!%n", args[0]);
    }
  }
}
```

Listing 1.3 – Code pour la classe `Hello` avec paramètres

Pour exécuter le programme, il faut désormais donner le nom complet de la classe à la commande `java`, avec le paquet auquel elle appartient. L'option `-cp` permet de donner un ensemble de répertoires ou d'archives dans lesquels `java` doit chercher les classes utilisées (le *class path*). C'est similaire au `-I` de `gcc`.

Une autre solution est d'exporter la variable d'environnement `$CLASSPATH` contenant la même liste de répertoires.

Question 5 (Paramètres) *:* Modifiez le programme pour qu'il accepte le nom de la personne à saluer en argument de la ligne de commande. Si aucun nom n'est fourni, le programme doit échouer avec un message d'erreur adapté.

Réponse : Les paramètres passé en ligne de commande au programme sont automatiquement insérés dans le tableau de chaînes de caratères passé en paramètre à la méthode `main`.

En cas d'erreur, la bonne pratique est d'afficher un message sur l'erreur standard (`System.err` [1]) et de terminer le programme avec un code d'erreur non nul (`System.exit(int)` [2]). Notez cependant que la méthode `exit` ne doit être utilisée que dans des cas très limités, comme ici.

1. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/System.html#err
2. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/System.html#exit(int)

```
$ javac -d build src/Hello3.java
$ java -cp build poo.tp.premierspas.Hello3 && echo ok || echo ko
Give me your name!
ko

$ java -cp build poo.tp.premierspas.Hello3 Zaphod && echo ok || echo ko
Hello Zaphod!
ok
```

LISTING 1.4 – Exécution du programme

On peut tester le comportement de notre classe du listing 1.3 page précédente comme présenté listing 1.4.

Question 6 (Erreurs) *:* Introduisez délibérément des erreurs dans votre programme, et tentez de l'exécuter, afin de vous familiariser avec les messages d'erreur du compilateur Java.

Exercice 2 (Nombre mystère)

On se propose ici de réaliser un jeu simple de question–réponse. Le but du programme est de faire deviner à l'utilisateur un nombre dans un intervalle donné. Le programme devra donc lire dans ses options de ligne de commande les deux valeurs minimales et maximales des nombres à deviner (et déclencher une erreur si le 2^{e} est inférieur au 1er).

Il devra ensuite s'exécuter en mode interactif, demandant successivement à l'utilisateur de donner un nombre, et lui indiquant si le nombre à deviner est plus grand ou plus petit que la proposition.

On peut lire une chaine saisie au clavier avec `System.console().readLine()` [a]. Une chaîne de caractère peut être convertie en entier avec `Integer.parseInt(String)` [b]. On peut générer des nombres aléatoires avec la classe `java.util.Random`, notamment sa méthode `nextInt(int)` [c]. Pour afficher un message d'erreur et terminer le programme, on peut utiliser respectivement la méthode `printf` [d] sur `System.err` et `System.exit(int)` avec une valeur positive décrivant l'erreur.

a. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/Console.html#readLine()

b. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/Integer.html#parseInt(java.lang.String)

c. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Random.html#nextInt(int)

d. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/io/PrintStream.html#printf(java.lang.String,java.lang.Object...)

Question 1 *:* Dans un premier temps, réalisez ce programme comme une unique méthode `public static void main(String[] args)`. Pensez à signaler si la proposition est à l'extérieur de l'intervalle demandé.

Question 2 *:* Le programme est maintenant une instance de la classe. Déplacer les variables locale (min, max, et nombre à deviner) dans des attributs. Découpez la méthode `main` en sous méthodes d'instances. La méthode `main` se contente de créer une instance, de l'initialiser et d'appeler sa méthode `run`.

1.2 Familles

Exercice 3 (Dates) Une date est constituée d'une année (supérieure ou égale à 0), d'un mois (compris entre 1 et 12) et d'un jour (compris entre 1 et 31).

Question 1 *:* Créez la classe `Date` permettant de créer des dates et de les afficher. On ne vérifiera pas que le nombre de jours est correct par rapport au mois. Interdisez aux utilisateurs de modifier une date créée. La classe `Date` doit obligatoirement se trouver dans un fichier nommé `Date.java`

Si l'un des paramètres passé au constructeur est erroné, vous pouvez utiliser `throw new IllegalArgumentException();` pour indiquer une erreur.

Réponse :

Comme on veut rendre cette classe non modifiable, il faut rendre tous les attributs privés et ne fournir que des accesseurs (*getter*). À la rigueur, des attributs public finaux pourraient faire l'affaire, mais cette approche est à éviter car elle casse l'encapsulation de la classe en exposant directement ses attributs, ce qui rend l'évolution du code beaucoup plus difficile. Dans un langage disposant de propriétés (Python, C#, EcmaScript) on passerait par celles-ci. Ici en Java, on passe par des accesseurs. On peut quand même rendre les attributs finaux, afin de s'assurer qu'ils ne seront pas modifiés par ailleur.

Comme les valeurs des attributs ne pourront pas être modifiées après création (privés sans mutateur (*setter*) et de toute façon finaux), il faut définir un constructeur de trois paramètres permettant d'initialiser notre date à l'instanciation.

Le code final de la classe `Date` est donné listing 1.5 page suivante.

Question 2 *:* Compilez votre programme au moyen de la commande `javac -Xlint :all Date.java`. Pouvez-vous l'exécuter ?

Réponse : Seules les classe disposant d'une méthode `public static void main(String[] args)` peuvent être directement exécutées (avec la commande `java NomDeClasse`).

```java
package poo.tp.familles;

public final class Date implements Comparable<Date> {

  private final int year;
  private final int month;
  private final int day;

  public Date(int y, int m, int d) {
    if (y < 0) {
      throw new IllegalArgumentException("the year must be positive");
    }
    if (m < 1 || m > 12) {
      throw new IllegalArgumentException("the month must be between 1 and 12");
    }
    if (d < 1 || d > 31) {
      throw new IllegalArgumentException("the day must be between 1 and 31");
    }
    this.year = y;
    this.month = m;
    this.day = d;
  }

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

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

  public int day() {
    return this.day;
  }
```

LISTING 1.5 – Code initial des dates

```java
@Override
public String toString() {
  return String.format("%04d-%02d-%02d", year(), month(), day());
}
```

LISTING 1.6 – Code pour la méthode `toString` des dates

Question 3 *:* La classe `Object` (dont hérite la classe `Date`) possède une méthode `public String toString()` permettant de retourner une chaîne de caractères associée à un objet. Redéfinissez cette méthode.

L'opérateur permettant de concaténer des chaînes de caractère est `+`. Ainsi, `"abc" + 3 + "def" + "ghi"` génère la chaîne de caractère `"abc3defghi"`.

On peut aussi utiliser la méthode statique `String.format` [a].

a. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html#format(java.lang.String,java.lang.Object...)

Réponse :

La version par défaut de cette méthode n'affichant que la classe d'une instance et son adresse, il est souvent utile de redéfinir celle-ci afin d'afficher une représentation plus compréhensible, ne serait-ce que pour faciliter la recherche de bugs. C'est particulièrement vrai pour les objets valeurs comme ici, puisqu'alors ce sont les valeurs des attributs qui comptent et non l'adresse de l'objet.

Notez dans le listing 1.6 que même si les attributs sont accessibles à la classe, les accesseurs sont utilisés. Cela permet de facilement pourvoir changer l'implémentation en limitant les changement aux accesseurs. C'est ce que l'on appelle l'*auto-encapsulation*.

Question 4 *:* Écrivez une méthode de comparaison permettant de savoir si la date actuelle est antérieure, égale ou postérieure à une date passée en paramètre (voir l'interface `Comparable<T>` [3]).

Réponse :

Pour faire cela, on va implémenter l'interface `Comparable<Date>`, et donc créer une méthode `public int compareTo(Date other)`, qui retournera 0 si les deux dates sont égales, un nombre négatif si `other` est supérieure et un nombre positif si `other` est inférieure.

Pour faciliter l'implémentation, on utilisera `Integer.compare` qui a le même comportement.

Il est intéressant aussi de redéfinir la méthode `public boolean equals(Object other)` qui permet de comparer deux objets quelconque pour l'égalité, en utilisant la

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

```java
@Override
public boolean equals(Object other) {
  if (other == this) { return true; }
  if (other == null) { return false; }
  if (other.getClass() != this.getClass()) { return false; }
  Date otherDate = (Date) other;
  return this.year() == otherDate.year()
        && this.month() == otherDate.month()
        && this.day() == otherDate.day();
}

@Override
public int hashCode() {
  return java.util.Objects.hash(year(), month(), day());
}

@Override
public int compareTo(Date other) {
  if (this.year() != other.year()) {
    return Integer.compare(this.year(), other.year());
  }
  if (this.month() != other.month()) {
    return Integer.compare(this.month(), other.month());
  }
  if (this.day() != other.day()) {
    return Integer.compare(this.day(), other.day());
  }
  return 0;
}
```

LISTING 1.7 – Code de comparaison de dates

valeurs des attributs. Par défaut, cette méthode compare l'identité, c'est-à-dire si les deux instances sont les mêmes.

De manière générale, un objet comparable doit également redéfinir `equals` afin d'être cohérent sur la sémantique de l'égalité : `a.compareTo(b) == 0` ⇔ `a.equals(b)`.

Un point difficile de l'implémentation de `equals` et de `compareTo` est de maintenir les propriétés mathématique de ces opérateurs dans le cas de sous-typage (notamment la commutativité et la transitivité). Pour faciliter le choses ici, on rend la classe `Date` finale (`final`), c'est-à-dire qu'on ne peut pas l'étendre.

Ce type d'objet, immuable et égal par valeur, est appelé un *datatype* en UML (*Unified Modeling Language*), et correspond au patron de conception *Value Object* Fowler, 2002. Les classes de ce types sont la plupart du temps finale.

Enfin, chaque fois que l'on redéfinit la méthode `equals`, il est *impératif* de redéfinir également la méthode `hashCode` Bloch, 2008. En effet, cette méthode, utilisée notamment dans les `HashMap`, retourne un entier (idéalement) unique pour chaque objet identique. Pour maintenir la sémantique de l'égalité, il faut donc que deux objets égaux (au sens de `equals`) aient le même `hashCode`. Il faut donc également redéfinir cette méthode pour que le hash retourné soit calculé à partir des même attributs que ceux utilisés dans `equals`.

L'implémentation des trois méthodes correspondantes est détaillée listing 1.7 page précédente.

Exercice 4 (Personnes) Une personne possède un nom, un prénom et une date de naissance.

Question 1 *:* Créez une classe `Personne` permettant de manipuler les informations concernant une personne. Notamment, une personne doit pouvoir changer son nom.

Réponse :

Approche classique, les attributs sont privés, et on fournit des accesseurs et mutateurs si nécessaire. On va considérer ici que seuls le nom et le prénom peuvent être changés. La date de naissance sera par contre en lecture seule. On crée aussi un contructeur prennant en paramètre ces attributs, et une methode permettant d'afficher une personne.

Question 2 *:* Une personne peut être soit célibataire, soit mariée. Lorsqu'elle est mariée, il est important de connaître la date du mariage. Programmez une méthode permettant à deux personne de se marier. On vérifiera que la date du mariage est postérieure à la date de naissance de chaque personne.

Programmez aussi une méthode permettant de savoir si une personne est mariée ou non.

Réponse : Chaque époux doit connaître son conjoint. On est donc dans un cas classique de double navigabilité. Le mariage d'une personne doit donc mettre à jour le statut d'au moins une autre personne, son nouveau conjoint, et éventuellement d'une autre, son ancien si celle-ci était déjà mariée.

```
public final class Person {

  private String familyName;
  private String givenName;
  private final Date birthDate;
  public Person(String fn, String gn, Date bd) {
    this.familyName = fn;
    this.givenName = gn;
    this.birthDate = bd;
    this.gender = Gender.MALE;
  }
  public Date birthDate() { return this.birthDate; }

  public String getFamilyName() { return this.familyName; }

  public void setFamilyName(String n) {
    if (gender == Gender.MALE) {
      throw new IllegalStateException("Males can't change name");
    } else {
      this.familyName = n;
    }
  }

  public String getGivenName() { return this.givenName; }

  public void setGivenName(String n) { this.givenName = n; }

  public String getFullName() {
    return getGivenName() + " " + getFamilyName();
  }
  @Override
  public String toString() {
    return "Person<" + getFullName() + ">";
  }
```

Listing 1.8 – Attributs et constructeur de Personne

Pour simplifier le modèle, on va considérer que l'on ne peut se marier que si on ne l'est pas déjà, et donc fournir une méthode permettant de divorcer. Cette méthode devra être appelée explicitement avant un mariage.

La méthode `divorce` doit donc réinitialiser les attributs correspondants (date de mariage et époux) à `null` et s'appeller sur l'ancien conjoint, puisqu'il doit divorcer également. Cette méthode doit être sans effet dans le cas d'une personne qui n'est déjà pas (ou plus) mariée, pour éviter une récursion mutuelle infinie.

La méthode `wed` est plus complexe. En effet, elle doit s'appliquer aussi à l'autre conjoint, avec `this` en paramètre, mais uniquement si l'autre personne n'est pas déjà marié avec soi, pour éviter ici aussi une récursion mutuelle infinie. Cependant, comme une exception sera levée dans le cas où la personne est déjà mariée à une autre, les changement sur l'objet lui-même (`this`) ne doivent pas être appliqués *a priori*, mais uniquement si l'on sait que le mariage réciproque réussira. Faute de quoi, seul un des deux conjoints sera marié à l'autre.

Il serait tout aussi simple d'appeler `divorce` dans `wed` plutôt que de lever une exception, mais ce divorce implicite pourrait être perturbant.

Comme le dit le *Zen of Python*[a], « Explicit is better than implicit ».

a. https://www.python.org/dev/peps/pep-0020/

Le code de cet aspect est fournit ci-dessous.

```
private Date weddingDate;
private Person spouse;
public boolean isMarried() {
  return this.weddingDate != null && this.spouse != null;
}

public Person spouse() { return this.spouse; }

public Date weddingDate() { return this.weddingDate; }
public boolean divorce() {
  if (!isMarried()) { return false; }
  this.weddingDate = null;
  Person other = this.spouse;
  this.spouse = null;
  other.divorce();
  return true;
}
public boolean wed(Person spouse, Date weddingDate) {
  if (this.spouse == spouse) { return false; }
```

```
    this.spouse = validateWedding(spouse, weddingDate);
    this.weddingDate = weddingDate;
    try {
      spouse.wed(this, weddingDate);
      return true;
    } catch (IllegalStateException e) {
      this.spouse = null;
      this.weddingDate = null;
      throw e;
    }
  }

  private Person validateWedding(Person spouse, Date weddingDate) {
    if (this.isMarried() || (spouse.isMarried() && spouse.spouse() != this)) {
      throw new IllegalStateException("Only single persons can be wed. Call ↩
      divorce to make a person single.");
    }
    if (weddingDate.compareTo(birthDate()) <= 0) {
      throw new IllegalArgumentException("The wedding date must be after ↩
      birth date");
    }
    if (spouse.isSibling(this) || spouse.isChild(this) || spouse.isParent(this)) {
      throw new IllegalArgumentException("Can't wed a family member");
    }
    return spouse;
  }
```

Exercice 5 (Liens de parenté)

Question 1 *:* À présent, on ajoute à chaque personne deux parents. Ces parents sont optionnels, dans le sens où ils peuvent être inconnus.

Réponse : On va utiliser un `Set<Person>` pour stocker les deux parents. Cependant, cette structure sera totalement encapsulée. Il faudra donc fournir des méthodes permettant d'ajouter un parent (qui lèvera une exception si la personne en a déjà deux), d'en supprimer un, et éventuellement de récupérer la collection des parents. Afin d'éviter un problème d'*aliasing*, où une autre méthode pourait ajouter ou supprimer des parents directement à la structure sans passer par la personne en question, on retourne un ensemble non modifiable (`unmodifiableSet`), ou une copie de l'attribut.

Pour faciliter la création d'une personne, on ajoute aussi deux constructeurs prennant respectivement en paramètre une et deux personnes (en plus des paramètres précédents), qui les ajoutera directement comme parents de la personne créée.

```
private final Set<Person> parents = new HashSet<>(2);
public Person(String fn, String gn, Date bd, Person p1) {
  this(fn, gn, bd);
  this.addParent(p1);
}

public Person(String fn, String gn, Date bd, Person p1, Person p2) {
  this(fn, gn, bd, p1);
  this.addParent(p2);
}
public Set<Person> parents() {
  return unmodifiableSet(parents);
}

public boolean addParent(Person parent) {
  if (parents.size() == 2 && !parents.contains(parent)) {
    throw new IllegalStateException("This person already has two parents");
  }
  if (parent.birthDate().compareTo(this.birthDate()) >= 0) {
    throw new IllegalArgumentException("A parent can't be younger than its ↩
    child");
  }
  return parents.add(parent);
}

public boolean removeParent(Person parent) {
  return this.parents.remove(parent);
}
```

Listing 1.9 – Gestion des parents

```java
public boolean isSibling(Person other) {
  for (Person p : other.parents()) {
    if (this.parents.contains(p)) { return true; }
  }
  return false;
}

public boolean isParent(Person other) {
  return other.isChild(this);
}

public boolean isChild(Person other) {
  return parents.contains(other);
}
```

LISTING 1.10 – Test des fratries

Question 2 : Programmez une méthode permettant de savoir si la personne en question est le frère ou la sœur d'une autre personne passée en paramètre (un seul parent en commun est considéré suffisant).

Réponse : La méthode `isSibling` est directe. Si l'on avait voulu considérer l'égalité des deux parents, on aurait pû utiliser directement la méthode `containsAll`.

Question 3 : On considère maintenant que les personnes sont soit des hommes, soit des femmes et que les hommes ne peuvent pas changer leur nom de famille (le système est assez ancien). Faites les modifications appropriées.

Réponse : Selon comment on souhaite gérer les mariages et parents, les modifications à appliquer varient. Si on autorise les mariages entre personnes du même genre et que les deux parents aient le même genre (adoption), alors les méthodes correspondantes ne changent pas. Les seuls changements à appliquer sont alors :

- passer la classe `Person` en abstraite (`abstract`) ;
- créer une classe `Woman` héritant de `Person` :
  `public class Woman extends Person`
  Cette classe ne redéfinis aucune méthode ;
- créer une classe `Man` héritant de `Person` :
  `public class Man extends Person`
  Cette classe devra redéfinir la méthode `setFamilyName` pour lever une exception de type `UnsupportedOperationException`.

Pour les mariages, on peut modifier la méthode `wed` de `Person` pour lever une exception (`IllegalArgumentException`) si `spouse` est de la même classe que `this` (`this.getClass().equals(spouse.getClass())`), dans le cas où l'on voudrait interdire les mariages entre personnes de même genre.

```java
public enum Gender {
  MALE,
  FEMALE,
}

private final Gender gender;
```

Listing 1.11 – Énumération pour la gestion du genre

Si l'on considère qu'une personne peut changer de genre, il est peut être plus intéressant d'utiliser un attribut de type `enum` pour le représenter (voir listing 1.11). Dans ce cas, le bloquage du changement de nom ne se fera pas par polymorphisme mais par un test sur cet attribut (comme en listing 1.8 page 10). Si c'est la seule différence entre un homme et une femme, cette approche est tout aussi valide. Dans le cas ou de nombreuses méthode ferait un test sur l'attribut de genre pour déterminer leur comportement, il serait alors plus intéressant de passer par du polymorphisme.

TP 2

Exercice 1 (Animaux)

Question 1 *:* Écrivez une classe abstraite `Animal`. On supposera qu'un animal possède toujours un nom, mais ne peux pas en changer. Redéfinissez la méthode `toString`.

Écrivez aussi l'accesseur permettant d'obtenir le nom de l'animal.

Réponse : Cette classe est très simple :

Question 2 *:* Écrivez deux sous-classes `Dog` et `Cat`. Redéfinissez la méthode `toString` pour chacune.

Réponse : Comme ces classes étendent la classe `Animal`, il faut penser à définir un constructeur ayant les même paramètres que celui d'`Animal`, qui se contentera d'appeler `super` (voir listings 2.2 et 2.3 page suivante et page 19).

Question 3 *:* Dans la fonction main, créez les objets `medor` et `felix` de la manière suivante :

```
Dog medor = new Dog("Medor");
Animal felix = new Cat("Felix");
```

Affichez les objets `medor` et `felix`. Quelle est la méthode `toString` appellée ?

Réponse : Dans les deux cas, la méthode redéfinie est appelée. En effet, la résolution de l'appel de méthode se fait dynamiquement à l'exécution. Même si `felix` est déclaré comme étant un `Animal`, c'est bien la méthode redéfinie qui est résolue (classe réelle) et donc appelée.

Question 4 *:* Dans la classe `Dog`, créez une méthode `public void woof()` affichant un message. Pouvez-vous faire aboyer l'objet `medor` ?

Réponse :

`medor` est un chien, il peut donc aboyer...

Question 5 *:* Dans la classe `Chat`, créez une méthode `public void meow()` affichant un message. Pouvez-vous faire miauler l'objet `felix` ?

Réponse :

```java
package tp.animaux;

abstract class Animal {
  private final String name;

  public Animal(String name) {
    this.name = name;
  }

  public String name() { return this.name; }

  @Override
  public String toString() {
    return "Animal<name=" + name() + ">";
  }
}
```

Listing 2.1 – Code de la classe Animal

```java
package tp.animaux;

class Dog extends Animal {

  public Dog(String name) { super(name); }

  @Override
  public String toString() {
    return "Dog<name=" + name() + ">";
  }

  public void woof() {
    System.out.println(name() + " says \"woof\"");
  }
}
```

Listing 2.2 – Code de la classe Dog

```java
package tp.animaux;

class Cat extends Animal {
  public Cat(String name) { super(name); }

  @Override
  public String toString() {
    return "Cat<name=" + name() + ">";
  }

  public void meow() {
    System.out.println(name() + " says \"meow\"");
  }
}
```

Listing 2.3 – Code de la classe Cat

felix est déclaré comme étant un Animal, et ne peut donc pas miauler.

Question 6 *:* L'objet felix ne peut pas miauler car il n'est pas considéré comme un Chat mais comme un Animal au moment de la compilation. Réessayez en appelant un chat un chat : ((Cat)felix).meow();

Réponse : La classe réelle de felix est bien Cat, il est donc possible de le transtyper en chat, et dans ce cas de le faire miauler.

Question 7 *:* Que se passe-t-il si vous essayez de faire miauler medor ?

Réponse : On ne peut pas appeler directement la méthode meow sur medor puisque c'est un chien. Si l'on essai de le transtyper comme précédemment, le code ne compile plus. En effet, sa classe réelle est Dog, qui n'est pas liée à Cat, il n'est donc pas possible de le convertir.

Exercice 2 (Mots) La classe String[1] du paquetage java.lang permet de manipuler des chaînes de caractère.

Question 1 *:* Écrire une classe Mot qui permet de stocker une chaîne de caractères lue au clavier.

> On peut lire une chaine saisie au clavier avec System.console().readLine().

Question 2 *:* Ajoutez la méthode toString pour votre classe Mot.

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

```java
package tp.animaux;

public final class Animals {
  public static void main(final String[] args) {
    Dog medor = new Dog("Medor");
    Animal felix = new Cat("Felix");

    System.out.println(medor);
    System.out.println(felix);

    medor.woof();
    ((Cat) felix).meow();
  }
}
```

Listing 2.4 – Classe principale exécutable

Question 3 *:* Écrivez une méthode `afficheVoyelles` qui affiche les voyelles du mot, dans l'ordre où elles apparaissent. Par exemple, cette fonction affiche `ioaiue` à partir du mot `informatique`.

> Pour obtenir la longueur d'une chaîne de caractères, la classe `String` met à disposition la méthode `public int length()`. Pour obtenir le caractère en position *index*, la classe `String` expose la méthode `public char charAt(int index)`.

Question 4 *:* Ajoutez une méthode `estPalindrome` qui permet de savoir si le mot est un *palindrome*. Un palindrome est un mot qui peut être lu dans les deux sens.

Question 5 *:* Écrivez une méthode `estContenu` qui indique si un `Mot` donné en paramètre contient ou non le mot en cours.

Question 6 *:* Avec des boucles `for` et sans tableau, triez les lettres d'un `Mot`. Par exemple, pour le mot `informatique`, cela donne `aefiimnoqrtu`.

TP 3

Exercice 1 (Le jeu de la bataille (simplifiée)) Nous allons programmer une version simplifiée du jeu de la bataille. Dans ce jeu de cartes, chaque joueur joue la carte de son choix (de son paquet) à chaque tour. La carte ayant la plus forte valeur remporte, ou la plus grande couleur si les valeurs sont égales. Le joueur ayant mis la carte la plus forte ramasse les deux cartes posées et le jeu recommence, jusqu'à ce que l'un des deux joueurs n'ait plus de carte [1]

Le diagramme de classes en figure 3.1 présente une version simplifiée de l'architecture de ce programme.

Question 1 *:* Programmez la classe `Carte`. La méthode `compareTo` retourne une valeur négative (strictement) si la carte actuelle est plus faible que l'autre carte, 0 si elles sont égales, et une valeur positive (strictement) sinon. Elle est définie dans l'interface `Comparable<T>`. Votre classe `Carte` devra donc l'implémenter.

Réponse : Le code correspondant est donné listing 3.1 page 23.

Question 2 *:* Programmez la classe `Paquet`. Pour le moment, ne programmez pas le contenu de la méthode `public void melanger()`.

Réponse : Le cœur principal de la classe pour le paquet, `Deck`, est donné listing 3.2 page 24. On peut ajouter des méthodes utiles par délégation sur la liste des cartes, comme montré en listing 3.3.

Question 3 *:* Programmez la classe abstraite `Joueur`.

Réponse : Le code est donné listing 3.4 page 25

Question 4 *:* Programmez la classe `Ordinateur`. Pour jouer, l'ordinateur choisi une carte aléatoirement dans son paquet.

Réponse : Le code est donné listing 3.5 page 26

1. Ce jeu n'est certe pas très distrayant, dans le sens où le joueur ayant la carte la plus forte initialement est sûr de ne pas pouvoir perdre (et même de gagner s'il joue cette carte à chaque tour). Qu'à cela ne tienne, c'est la programmation du jeu qui sera distrayante.

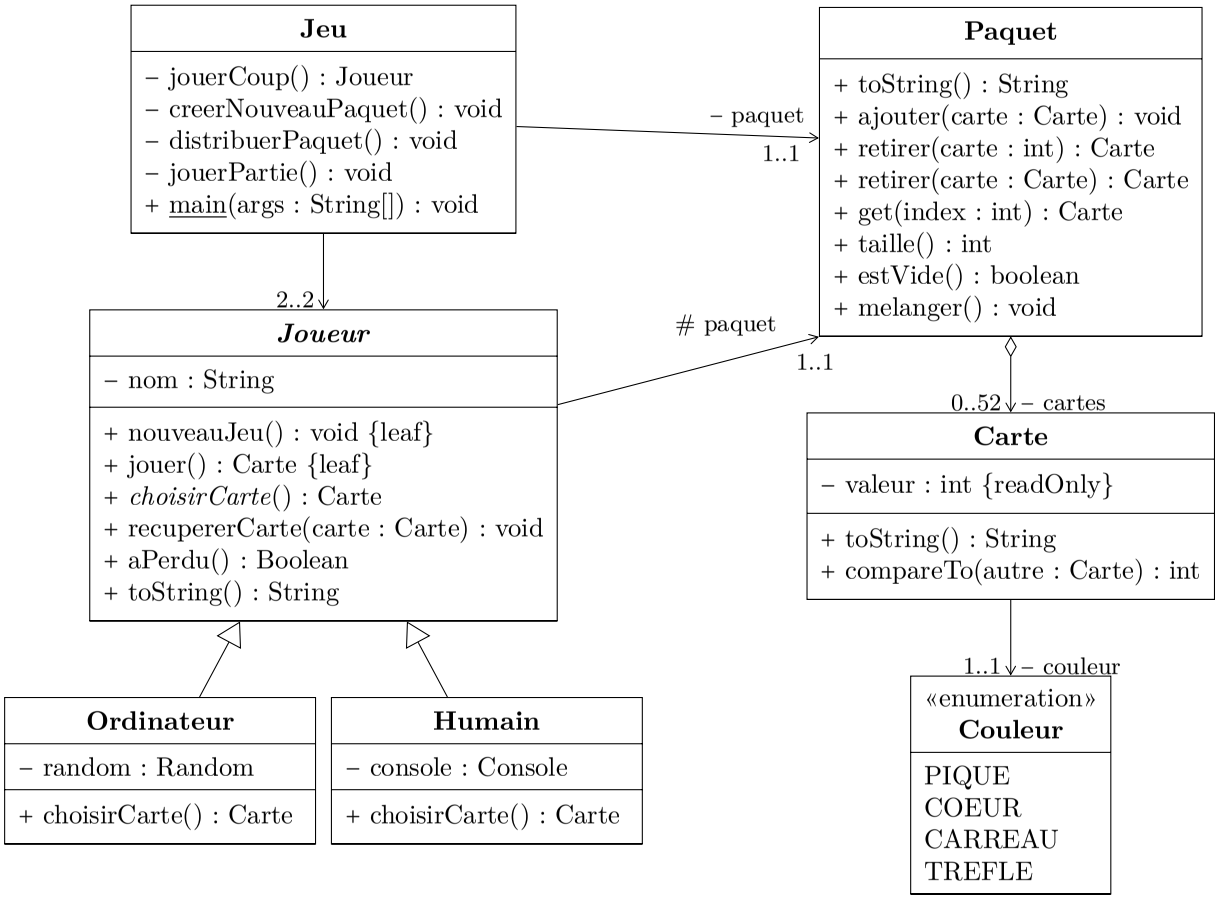

FIGURE 3.1 – Diagramme de classes du jeu de la bataille

```java
package tp.bataille;

class Card implements Comparable<Card> {

  public enum Suit { SPADE, HEART, DIAMOND, CLUB; }

  private final Suit suit;
  private final int value;

  Card(Suit s, int v) {
    this.suit = s;
    this.value = v;
  }

  @Override
  public String toString() {
    return String.format("%d of %s", this.value, this.suit);
  }

  @Override
  public int compareTo(Card other) {
    if (other == null) {
      return 1;
    }
    if (this.value == other.value) {
      return this.suit.compareTo(other.suit);
    }
    return Integer.compare(this.value, other.value);
  }
}
```

Listing 3.1 – Code Java pour la carte

```
class Deck {

  private static final int NB_VALS = 13;

  private final Random rnd = new Random();
  private final ArrayList<Card> cards = new ArrayList<>();

  @Override
  public String toString() {
    return "Deck" + this.cards.toString();
  }

  public void init() {
    this.clear();
    for (Card.Suit c  : Card.Suit.values()) {
      for (int v = 1; v <= NB_VALS; v++) {
        this.add(new Card(c, v));
      }
    }
  }

  public Card pop() {
    return this.cards.remove(this.cards.size() - 1);
  }
}
```

Listing 3.2 – Cœur du paquet

```
  public boolean add(Card c) { return this.cards.add(c); }
  public void add(int index, Card c) { this.cards.add(index, c); }
  public Card remove(int idx) { return this.cards.remove(idx); }
  public boolean remove(Card c) { return this.cards.remove(c); }
  public int size() { return this.cards.size(); }
  public Card get(int idx) { return this.cards.get(idx); }
  public void clear() { this.cards.clear(); }
  public boolean isEmpty() { return this.cards.isEmpty(); }
  public Iterator<Card> iterator() { return this.cards.iterator(); }
```

Listing 3.3 – Méthodes utiles par délégation

```java
package tp.bataille;

abstract class Player {

  protected Deck deck = new Deck();
  private final String name;

  Player(String n) {
    this.name = n;
  }

  public abstract Card chooseCard();

  public abstract Card chooseCard(Card card);

  public final void newDeal() {
    this.deck.clear();
  }

  public final Card play(Card c) {
    Card card = this.chooseCard(c);
    this.deck.remove(card);
    return card;
  }

  public void take(Card card) {
    this.deck.add(card);
  }

  public boolean hasLost() {
    return this.deck.isEmpty();
  }

  public String toString() {
    return this.name;
  }
}
```

LISTING 3.4 – Classe abstraite pour les joueurs

```java
package tp.bataille;

import java.util.Random;

final class Computer extends Player {

  private final Random rnd = new Random();

  Computer(String name) {
    super(name);
  }

  @Override
  public Card chooseCard() {
    return this.deck.get(this.rnd.nextInt(this.deck.size()));
  }

  @Override
  public Card chooseCard(Card other) {
    if (other == null) {
      return chooseCard();
    }
    Card best = this.deck.max();
    if (other.compareTo(best) < 0) {
      return best;
    }
    return this.deck.min();
  }
}
```

LISTING 3.5 – Classe abstraite pour l'ordinateur

```java
public void shuffle() {
  for (int i = 0; i < 2 * this.size(); i++) {
    int a = rnd.nextInt(this.size());
    int b = rnd.nextInt(this.size());
    this.add(b, this.remove(a));
  }
}
```

LISTING 3.6 – Méthodes pour mélanger un paquet

```java
public void sort() { Collections.sort(this.cards); }
public Card max() { return Collections.max(this.cards); }
public Card min() { return Collections.min(this.cards); }
```

LISTING 3.7 – Méthodes utilitaires d'un paquet

Question 5 *:* Programmez finalement la classe `Jeu`.

Réponse : Voir annexe A.1.2 page 37.

Question 6 *:* Programmez la méthode `melanger` de la classe `Paquet`. Pour mélanger le jeu, on échangera x fois deux cartes choisies aléatoirement. La variable x aura pour valeur deux fois la taille du paquet.

Réponse : Le code est donné listing 3.6. Bien sur, cette méthode de mélange n'est ici que pour l'exercice. En situation « réelle », on utiliserait `Collections.shuffle`[2], sur le même modèle que les méthodes outils ajoutées en listing 3.7.

Question 7 *:* Programmez une classe `Humain` héritant de `Joueur` et permettant à un joueur humain de jouer, en affichant son jeu à l'écran et demandant quelle carte il décide de jouer.

Réponse : Le code est donné listing 3.8 page suivante. La partie interface en ligne de commande (CLI pour *command-line interface*) est externalisée dans une classe utilitaire `UI`, dont le code est donné en annexe A.1.1 page 35.

Question 8 *:* Ajouter enfin la méthode `main` à la classe `Jeu`. Elle initialise le jeu avec un joueur humain et un ordinateur, distribue les cartes et joue la partie. Lorsque la partie est terminée, le gagnant est affiché et le programme se termine.

Réponse : Voir annexe A.1.2 page 37.

2. https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Collections.html#shuffle(java.util.List)

```java
package tp.bataille;

final class Human extends Player {

  private UI console;

  Human(String name, UI console) {
    super(name);
    this.console = console;
  }

  @Override
  public Card chooseCard(Card c) {
    return chooseCard();
  }

  @Override
  public Card chooseCard() {
    this.deck.sort();
    return this.console.chooseCard(this, this.deck);
  }
}
```

LISTING 3.8 – Code pour le joueur humain

TP 4

Listes, tableaux et comparaison de structures de données

Il est possible de stocker des données dans plusieurs structures. Le choix d'une structure de donnée adaptée peut avoir un impact sur la performance du programme. Dans ce TP, nous allons étudier différentes structures de données et comparer leur efficacité sur un exemple simple.

Exercice 1 (Tableaux)

Question 1 *:* Implémentez une classe `Matrice` qui permet de manipuler des matrices à deux dimensions.

Question 2 *:* Créez une méthode `public Matrice ajouter(Matrice autreMatrice)` retournant la matrice obtenue par la somme de la matrice actuelle avec une autre matrice.

Question 3 *:* Écrivez une méthode permettant de créer une structure de données triangulaire de la forme :

$$\begin{bmatrix} X & \varnothing & \varnothing \\ X & X & \varnothing \\ X & X & X \end{bmatrix}$$

Il est possible d'accéder à la case $(3, 3)$, mais pas à la case $(2, 3)$. Faites en sorte de ne pas allouer des cases inutilement.

Exercice 2 (Listes)

La classe utilisée pour manipuler les listes ici est la classe `java.util.ArrayList`.
Parmi les méthodes les plus utilisées d'une liste, on trouve :

- `boolean add(E elt);`
- `boolean contains(Object o);`
- `boolean remove(Object o);`

— `int size()`.

Il existe plusieurs classes implémentant l'interface `List` dans le JDK. Ces implémentations alternatives présentent des propriétés différentes. Il faut donc choisir la plus adaptée au problème visé.

Avant la version 1.2, on ne disposait que de la classe `Vector`, non générique. Depuis l'ajout de l'ensemble des collections dans la version 1.2, la classe `ArrayList` doit être utilisée de préférence, sauf dans quelques cas précis.

Comme toutes ces classes, `ArrayList` est une classe *générique*.

Question 1 *:* Comment peut-on ajouter les entiers 1, 2 et 3 dans une `ArrayList` ? Comment récupérer les valeurs entières ?

Question 2 *:* Implémentez vos classes permettant de manipuler des listes comme le fait `ArrayList`.

Question 3 *:* Implémentez une méthode `public void reverse()` qui renverse la liste. Ainsi, la liste (a, b, c, d) deviendrait (d, c, b, a).

Exercice 3 (Tableaux associatifs) La classe utilisée pour manipuler les tableaux associatifs (ou table de hashage), c'est-à-dire les tableaux indexés par un objet et non pas par un entier, est la classe `java.util.HashMap`. Une `HashMap` est implémentée au moyen d'un tableau de listes. Un couple $(clef, valeur)$ se trouve dans la liste de la case `clef.hashCode()` du tableau. Remarquons que la méthode `int hashCode()` est une méthode de la classe `Object`.

Comme pour les listes, plusieurs classes du JDK implémentent l'interface `Map`. Avant l'ajout dans 1.2 des interfaces de collection, la classe de référence était `Hashtable`, qui comme `Vector`, est désormais déconseillée au profis de `HashMap`.

Comme toutes ces classes, `HashMap` est une classe *générique*.

Parmi les fonctions les plus utilisées de la `HashMap`, on trouve :
— `V put(K key, V value)` ;
— `boolean containsKey(Object key)` et `boolean containsValue(Object val)` ;
— `V get(Object key)`.

Question 1 *:* Implémentez vos propres classes permettant de gérer des tableaux associatifs comme le fait `HashMap`. Vous pouvez utiliser la classe `ArrayList`.

Question 2 *:* Est-il possible de modifier votre classe de manière à avoir une méthode `void put(K key, int value)` ?

Pouvez-vous faire la différence entre un `int` et un `Integer` qui seraient ajoutés à votre tableau associatif ?

Est-il possible de modifier votre classe de manière à avoir aussi une méthode `int get(Object key)` lorsque la valeur entrée était un `int` ?

Exercice 4 (Comparaison de structures de données) Nous allons considérer un ensemble de 100 étudiants, ayant chacun un numéro d'étudiant et une note allant de 0 à 20. Nous allons comparer trois structures de données disponibles en Java : les tableaux, les listes (`ArrayList`) et les tables de hashage (`HashMap`)

Question 1 (Insertion des éléments) *:*

- Créez les classes dont vous aurez besoin. Ajoutez les constructeurs et accesseurs.
- Insérez 100 étudiants (générés aléatoirement) dans les structures de données.
- Cherchez dans la documentation de Java sur la classe `java.lang.System` la méthode permettant d'obtenir l'heure actuelle (à la milliseconde près).
- Mesurez le temps que prennent n insertions de 100 étudiants dans chacune de vos structures de données, avec n choisi de sorte que le temps d'exécution des n insertions de 100 étudiants prenne quelques dizaines de secondes.

Question 2 (Parcours des éléments) *:*

- Écrivez une méthode permettant de calculer la moyenne des étudiants, pour chacune des structures de données.
- Réorganisez vos classes de manière à ce que l'on puisse mesurer le temps de n insertions ou de n calcul de moyennes, de manière objet.

Question 3 (Recherche d'un élément) *:* Écrivez une méthode permettant de rechercher un étudiant n'existant pas, pour chacune des structures de données.

Question 4 (Récapitulatif) *:* Présenter la sortie du programme de manière à ce que la structure de données la plus efficace pour chacun des critère soit présentée clairement.

TP 5

Projet

Exercice 1 (Expressions arithmétiques) On se propose ici de créer une structure de donnée pour représenter et évaluer une expression arithmétique.

Une expression arithmétique sera représentée sous la forme d'une structure composite de type arbre, dont les nœuds pourront être de trois types :

— une constante, qui contient une valeur numérique (`double`),
— une variable, qui possède un nom,
— une opération binaire, qui possède deux fils qui sont eux même des expressions.

Question 1 (Structure) *:* Créer les différentes classes impliquées dans la structure décrite précédemment. On créera une classe concrète par type d'opération binaire.

Question 2 (Affichage) *:* Redéfinir la méthode `toString` pour afficher une expression sous forme de texte d'une manière lisible.

Question 3 (Évaluation) *:* Ajouter une méthode `eval` qui retourne la valeur de l'expression étant donnée une affectation de variable. Cette affectation pourra par exemple être passée en paramètre via une structure de type `Map<String, Double>` associant une valeur numérique à chaque nom de variable. On utilisera judicieusement la redéfinition.

Question 4 (Construction) *:* Créer une méthode permettant de créer une instance d'expression précédente à partir d'une chaîne de caractères en notation préfixée.

Rappel : en notation préfixée, l'expression $(2 \times x) + \frac{y}{3}$ sera notée `(+ (* 2 x) (/ y 3))`

Question 5 (Programme principal) *:* Écrire un programme exécutable qui permet de saisir une expression, et d'afficher sa valeur étant donner une affectation de variable.

Par exemple (la syntaxe n'est qu'une suggestion) :

```
> p <- (+ (* 2 x) (/ y 3))
(2 * x) + (y / 3)
> p(x=3, y=6)
8
```

Annexe A

Code supplémentaire

A.1 Jeu de bataille

Le code des classes annexes de l'exercice 1 page 21.

A.1.1 Classe pour l'interface CLI

```
package tp.bataille;

import java.io.Console;

class UI {

  private final Console console;

  UI(Console console) {
    this.console = console;
  }

  void play(Player player, Card card) {
    this.console.printf("-> %s plays %s%n", player, card);
  }

  void gameOver(Player winner, int turns) {
    this.console.printf("Game Over in %d turns.%n", turns);
    this.console.printf("%s is the winner.%n", winner);
  }
```

```
private void clear() {
  this.console.printf("\033[H\033[2J");
  this.console.flush();
}

void endOfTurn(Player winner) {
  this.console.printf("%n==> %s wins.%n", winner);
  this.console.readLine("Press Enter to continue...");
}

void startTurn() {
  clear();
}

String askName(String name) {
  String newName = this.console.readLine("%s give me your name : ", name);
  return newName == null ? name : newName.trim();
}

void displayDeck(Deck deck) {
  System.out.println("Your deck :");
  for (int i = 0; i < deck.size(); i++) {
    this.console.printf("%d : %s%n", i, deck.get(i));
  }
}

Card chooseCard(Player player, Deck deck) {
  this.console.printf("%n=== It's %s turn ===%n", player);
  this.displayDeck(deck);
  this.console.printf("%n----------------%n");
  while (true) {
    try {
      String rep = this.console.readLine("Your choice : ");
      if (rep == null) {
        return null;
      }
      return deck.get(Integer.parseInt(rep));
    } catch (Exception e) {
      this.console.printf("Invalid card number. Try Again%n");
    }
  }
}
```

```
  boolean playAgain() {
    String rep = this.console.readLine("Play again (y|N) ? ");
    return rep != null && rep.toUpperCase().startsWith("Y");
  }
}
```

A.1.2 Classe pour le jeu

```
package tp.bataille;

public final class Game {

  private static final UI CONSOLE_UI = new UI(System.console());
  private final Deck deck = new Deck();
  private Player player1;
  private Player player2;
  private int turns = 0;

  private Game(Player p1, Player p2) {
    this.player1 = p1;
    this.player2 = p2;
    this.createDeck();
  }

  private Player turn() {
    Card c1 = this.player1.play(null);
    CONSOLE_UI.play(this.player1, c1);
    Card c2 = this.player2.play(c1);
    CONSOLE_UI.play(this.player2, c2);
    Player winner = c1.compareTo(c2) > 0
                  ? this.player1
                  : this.player2;
    winner.take(c1);
    winner.take(c2);
    this.turns++;
    this.rankPlayers(winner);
    return winner;
  }

  private void rankPlayers(Player winner) {
    if (winner == this.player2) {
```

```
    this.player2 = this.player1;
    this.player1 = winner;
  }
}

private void createDeck() {
  this.deck.init();
  this.deck.shuffle();
}

private void deal() {
  this.player1.newDeal();
  this.player2.newDeal();
  while (this.deck.size() >= 2) {
    this.player1.take(this.deck.pop());
    this.player2.take(this.deck.pop());
  }
}

private boolean gameOver() {
  return this.player1.hasLost() || this.player2.hasLost();
}

private void play() {
  this.createDeck();
  this.deal();
  this.turns = 0;
  Player winner = this.player1;
  while (!this.gameOver()) {
    CONSOLE_UI.startTurn();
    winner = this.turn();
    CONSOLE_UI.endOfTurn(winner);
  }
  CONSOLE_UI.gameOver(winner, this.turns);
}

private static Player createPlayer(String type, String name) {
  if ("c".equals(type)) {
    return new Computer(name);
  }
  if ("h".equals(type)) {
    return new Human(CONSOLE_UI.askName(name), CONSOLE_UI);
```

```
    }
    throw new IllegalArgumentException();
  }

  void run() {
    do {
      this.play();
    } while (CONSOLE_UI.playAgain());
  }

  private static void usage() {
    System.err.println("args : [c|h] [c|h]");
    System.exit(1);
  }

  public static void main(String[] args) {
    if (args.length != 2) {
      usage();
    }
    try {
      Game game = new Game(
        createPlayer(args[0], "Player 1"),
        createPlayer(args[1], "Player 2")
      );
      game.run();
    } catch (IllegalArgumentException e) {
      usage();
    }
  }
}
```

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.

analyseur syntaxique (parser) TODO

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.

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

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.

CLI interface en ligne de commande (CLI pour *command-line interface*). Interface utilisateur où l'interaction se fait par saisie de commande au clavier, généralement dans un shell. Les environnement de commande interactif (REPL pour *read-eval-print loop*) peuvent aussi être considérées comme des CLI. *Voir aussi* interface humain-machine.

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.

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.

CSS (*Cascading Style Sheet*) langage permettant la mise en forme de données formatées en XML (*eXtensible Markup Language*), plus particulièrement les documents HTML (*Hypertext Markup Language*).

DTD (*Document Type Definition*) définition de la grammaire formelle utilisée par un langage basé sur SGML (*Standard Generalized Markup Language*) ou XML, et donc de sa syntaxe abstraite.

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

ECMAScript langage dynamique utilisé notamment dans les navigateurs Web. Voir *ECMAScript Language Specification* 2020.

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.

HTML (*Hypertext Markup Language*) langage de structuration de documents utilisé pour le Web. Il permet de représenter le contenu d'un document de manière sémantique, indépendamment de la manière dont il est présenté.

HTTP (*Hypertext Transfer Protocol*) Protocole de communication réseau (niveau application) conçu pour le Web.

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).

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.

JVM (Java Virtual Machine) machine virtuelle JAVA.

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.

navigateur logiciel permettant de se déplacer sur le Web de manière interactive (par opposition par exemple à un robot). C'est un client générique (par opposition à une application spécifique), qui peut donc servir à consulter n'importe quel site. Il contient généralement un client HTTP (*Hypertext Transfer Protocol*), un moteur de rendu HTML et CSS (*Cascading Style Sheet*), ainsi qu'une machine virtuelle pour ECMASCRIPT.

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`.

REPL environnement de commande interactif (REPL pour *read-eval-print loop*), où l'on interagit avec un programme en tapant des commandes directement dans un interpréteur interactif. Le résultat de l'exécution de la commande est affiché immédiatement. La plupart du temps, il s'agit d'un mode textuel dans un terminal, mais il existe des environnements plus riches (p. ex. la console des navigateur). *Voir aussi* interface humain-machine.

robot logiciel (agent) navigant sur le web de manière autonome, en suivant les hyperliens présents dans les pages. Il est généralement utilisé pour extraire de l'information des pages visités (*web scraping*), par exemple pour créer un index de recherche.

SGML (*Standard Generalized Markup Language*) méta-langage servant de support pour la création de langages à balises. Une instance de langage basée sur SGML doit définir l'ensemble des balises valides et leur imbrication (la grammaire) dans une DTD (*Document Type Definition*) associée.

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.

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.

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

URL (*Uniform Resource Locator*) système de nommage unique d'une ressource sur le Web, dépendant du protocole utilisé. C'est une référence de la ressource qui est déréférençable. Une URL est opaque pour le client. Voir Berners-Lee, Masinter et McCahill, 1994.

Web voir WWW (*World Wide Web*)

WWW (*World Wide Web* ou simplement *le Web*) système d'information global. C'est une interconnexion de ressources (pages) via des hyperliens, identifiées par leur URL (*Uniform Resource Locator*) dans le protocole HTTP.

XML (*eXtensible Markup Language*) méta-langage, dérivé de SGML. Il est plus strict que ce dernier, moins ambigu, et est donc plus facile à analyser.

Bibliographie

Berners-Lee, T. J., L. M. Masinter et M. McCahill (déc. 1994). *Uniform Resource Locators*. RFC 1738. Network Working Group. URL : http://tools.ietf.org/html/rfc1738 (cf. p. 44).

Bloch, J. (mai 2008). *Effective Java*. 2e éd. Addison-Wesley, p. 346. ISBN : 978-0321356680 (cf. p. 9).

ECMAScript Language Specification (juin 2020). ECMA 262. Ecma International. URL : https://www.ecma-international.org/publications-and-standards/standards/ecma-262/ (cf. p. 42).

Fowler, M. (nov. 2002). *Patterns of Enterprise Application Architecture*. Signature Series. Addison-Wesley, p. 560. ISBN : 978-0321127426 (cf. p. 9).