1 avertissement

Programmation Objet Avancée

Kevin Heraud

Cours 2 : classes abstraites, interfaces et exceptions

Département Informatique
Université de Tours

Les classes abstraites

Définition
Une **classe abstraite** ne peut pas être instanciée. Elle ne peut que servir de **classe de base** pour une dérivation.

```
1 public abstract class Abs {
2  public void f(float n){System.out.println(n + 1);}
3  public abstract void g (int n);
4 }
```

- elle peut contenir des **champs** ainsi que des **méthodes** dont hériteront ses classes filles
- elle peut aussi définir des **méthodes abstraites** (0 ou N) pour lesquelles on ne fournit que la **signature**

Déclaration et instanciation

Il est possible de déclarer une variable de type `Abs`

```
1 Abs a; // OK
```

Mais il est impossible de l'instancier à l'aide de l'opérateur `new`

```
1 Abs a = new Abs(); // Erreur
```

- il faut **dériver** une classe non abstraite de `Abs` pour pouvoir l'instancier. On parle alors de **classe concrète**

Déclaration et instanciation

- cette classe doit définir la(es) méthode(s) abstraite(s) de `Abs`

```
1  public class NotAbs extends Abs{
2   @Override
3   public void g(int n) {
4           System.out.println("Plus abstrait");
5       }
6  }
7  ...
8  Abs a = new NotAbs(); //
9  a.g();
```

Utilisation des classes abstraites

- l'utilisation de classes abstraites facilite la conception objet
- garantie que les classes concrètes disposeront de certaines fonctionnalités
- cela permet d'utiliser le mécanisme de **polymorphisme**

Exemple avec la classe `Geometrie`

- une classe `Geometrie` qui décrit toute forme géométrique et autorise à définir un point de départ, une méthode d'affichage et une méthode de calcul de la surface
- utilise la classe `Point` étudiée au cours prédédent

```
1 public abstract class Geometrie {
2  private Point coord;
3  public Geometrie(Point p) {coord = p;}
4  public void affiche() {coord.affiche();}
5  public abstract double surface();
6 }
```

Classe dérivée `CarreG`

```
1  class CarreG extends Geometrie {
2   private double largeur;
3   public CarreG(Point p, double l) {
4    super(p);
5    largeur = l;
6   }
7   public void affiche() {
8    super.affiche();
9    System.out.println("largeur "+ largeur);
10  }
11  public double surface() {
12   return largeur*largeur;
13  }
14 }
```

Classe dérivée `CercleG`

```
1  class CercleG extends Geometrie {
2   private double rayon;
3   public CercleG(Point p, double r) {
4    super(p);
5    rayon = r;
6   }
7   public void affiche() {
8    super.affiche();
9    System.out.println("rayon "+ rayon);
10  }
11  public double surface() {
12   return Math.PI*rayon*rayon;
13  }
14 }
```

Exemple d'appel

```
1 Point p1 = new Point(4,2);
2 Point p2 = new Point(8,9);
3 CarreG ca = new CarreG(p1,5);
4 ca.affiche();
5 System.out.println("surface "+ ca.surface());
6 CercleG ce = new CercleG(p2,4.3);
7 ce.affiche();
8 System.out.println("surface "+ ce.surface());
```

avec la sortie console suivante

```
1 Je suis en (4, 2)
2 largeur 5.0
3 surface 25.0
4 Je suis en (8, 9)
5 rayon 4.3
6 surface 58.08804816487527 5
```

En pratique !

- définir un tableau de géométries `tabG`
- insérer un cercle dans `tabG`
- insérer un carré dans `tabG`
- définir une classe `Rectangle`
- insérer un rectangle dans `tabG`
- calculer la somme des surfaces des géométries contenues dans `tabG`

Les interfaces

- une **interface** peut être vue comme une "contrat". Celle-ci se limite à définir :
  - une liste de méthodes dont on donne seulement la signature
  - et éventuellement des constantes
- toute classe qui implémente une interface s'engage à implémenter toutes les méthodes

Avantages liés aux interfaces

- une classe peut implémenter plusieurs interfaces (simule l'**héritage multiple**)
- une interface peut être étendue par une ou plusieurs autre(s) interface(s)
- peut être utiliser en tant que type

Interface : syntaxe

Définition d'une interface

```
1 public interface NomInterface {
2     // methode abstraite
3 }
```

Exemple

```
1 public interface I {
2  public void A();
3  public String B();
4 }
```

Implémentation de l'interface `I` par la classe `X`

```
1 public class X implements I{
2  public void A(){...}
3  public String B(){...}
4 }
```

Implémentation de 2 interfaces

Soit une deuxième interface `I2`

```
1 public interface I2 {
2  public void C();
3  public double D();
4 }
```

Implémentation de `I` et `I2`

```
1 public class X implements I, I2 {
2  public void A() { ... }
3  public String B() { ... }
4  public void C() { ... }
5  public double D() { ... }
6 }
```

Interface et polymorphisme

- il est possible de déclarer des variables de type interface

```
1 I i; // i est une reference a une classe qui
2      // implemente l'interface I
```

- mais il est impossible d'instancier une variable de type interface
- en revanche si nous avons une classe `X` qui implémente `I`

```
1 public class X implements I{ ...}
2 I i = new X(); // OK
```

Interface et classe dérivée

- la clause `implements` est indépendante de la clause `extends` utilisée pour l'héritage
- une classe dérivée peut implémenter une interface
- attention toutefois à l'**ordre** des clauses
- la clause d'héritage doit être placée avant la clause d'interface **sinon le code ne compilera pas** !

```
1 public class Y extends X implements I, I2 { ... } // OK
2 public class Z implements I extends X { ... } // ERREUR
```

Interface et constantes

- une interface peut également contenir des constantes
- elles sont considérées comme `static` et `final`
- elles sont donc accessibles en dehors de la classe implémentant l'interface

Interface et constantes

```
1  public interface I {
2   public void A();
3   public String B();
4   [public static final] int MAXI = 100;
5  }
6
7  public class A {
8   // definition de A() et B()
9   ...
10  if (x <= I.MAXI) { ... }
11 }
```

Dérivation d'interfaces

- une interface peut hériter d'une autre interface
- dans ce cas, cela revient à une "fusion"

```
1 public interface I1 {
2  public void A();
3 }
4
5 public interface I2 extends I1 {
6  public String B();
7 }
```

Dérivation d'interfaces

- la dérivation précédente est équivalente à

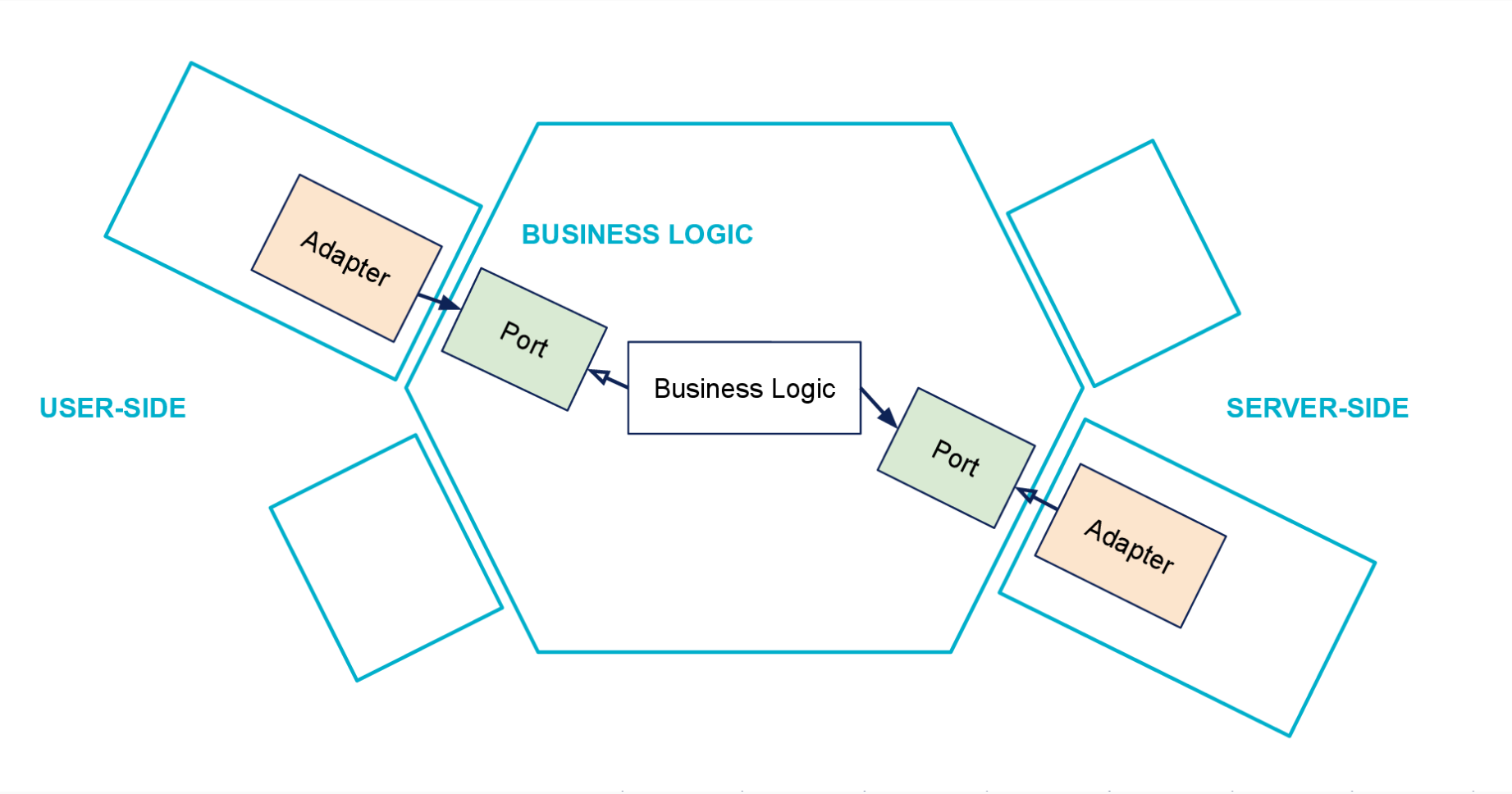
```
1 public interface I2 {
2  public void A();
3  public String B();
4 }
```

Méthodes par défaut

- depuis **Java 8** (nous sommes actuellement à la version 17), il est possible d'utiliser le mot-clé `default` pour définir des méthodes implémentées et utilisables par défaut
- ces méthodes peuvent être utilisées par toute classe qui implémente l'interface

```
1 public interface Quoi {
2  default void quiSuisJe(){
3   System.out.println("Je suis " + this);
4  }
5 }
```

Exemple : Architecture hexagonale

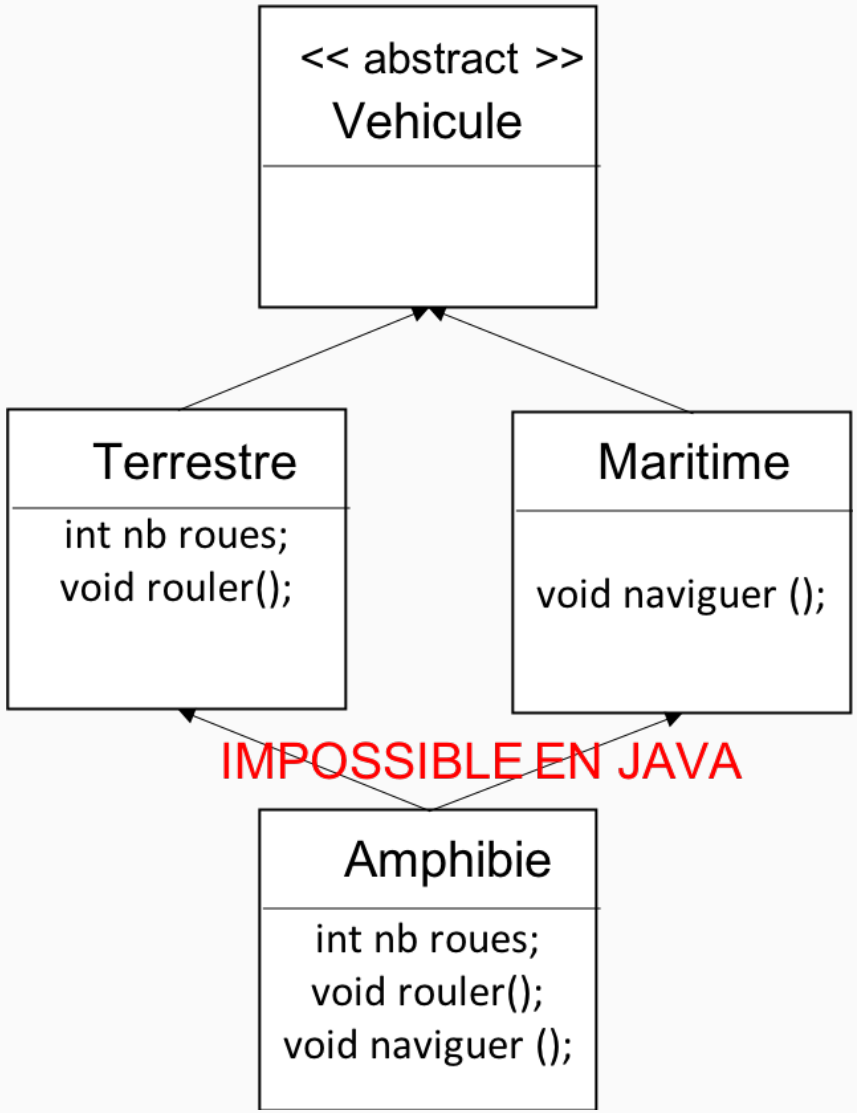


Exemple : les véhicules amphibies

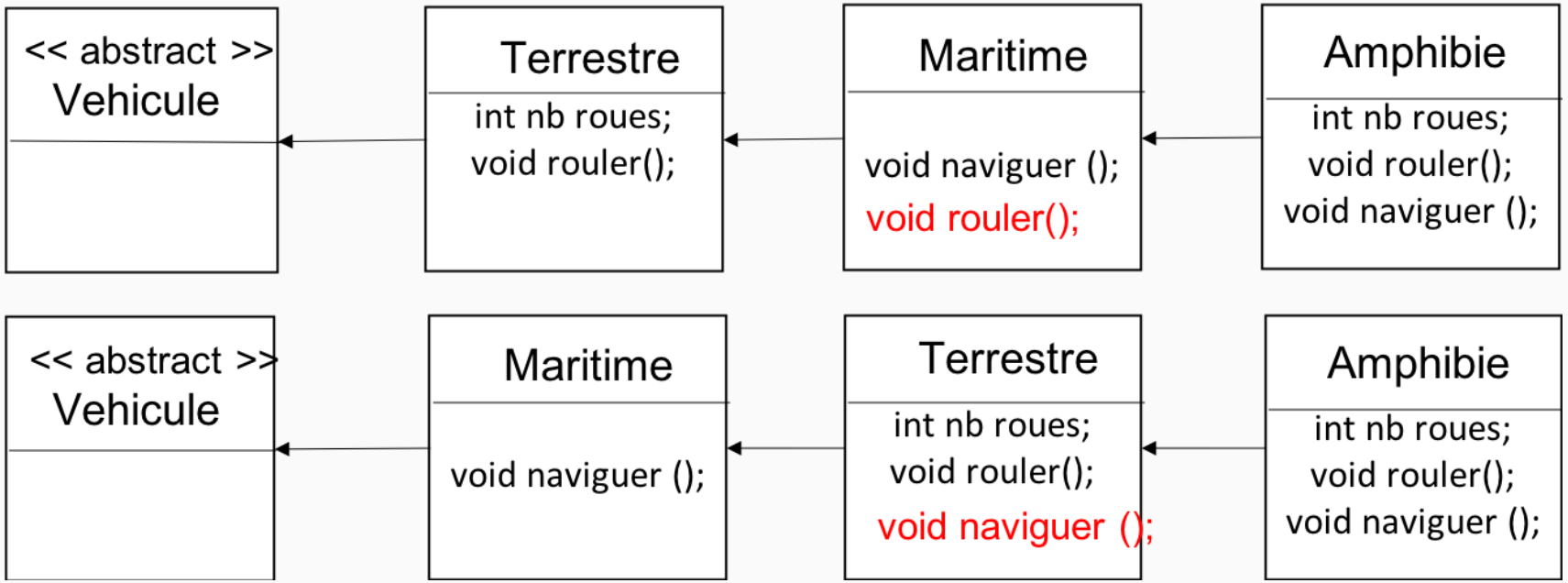
- comment gérer des véhicules
    - terrestres qui roulent sur terre avec des roues
    - maritimes qui naviguent sur l'eau
    - amphibies qui peuvent faire les deux ?
- plusieurs solutions envisageables
    1. utiliser une classe abstraite `Vehicule`
    2. utiliser des interfaces

Utiliser une classe abstraite

- possibilité de travailler indifféremment à partir de `Vehicule`
- les méthodes et attributs sont corrects, pas de redondances inutiles
- dans le schéma, les flèches indiquent un héritage
- mais l'héritage multiple est impossible en Java



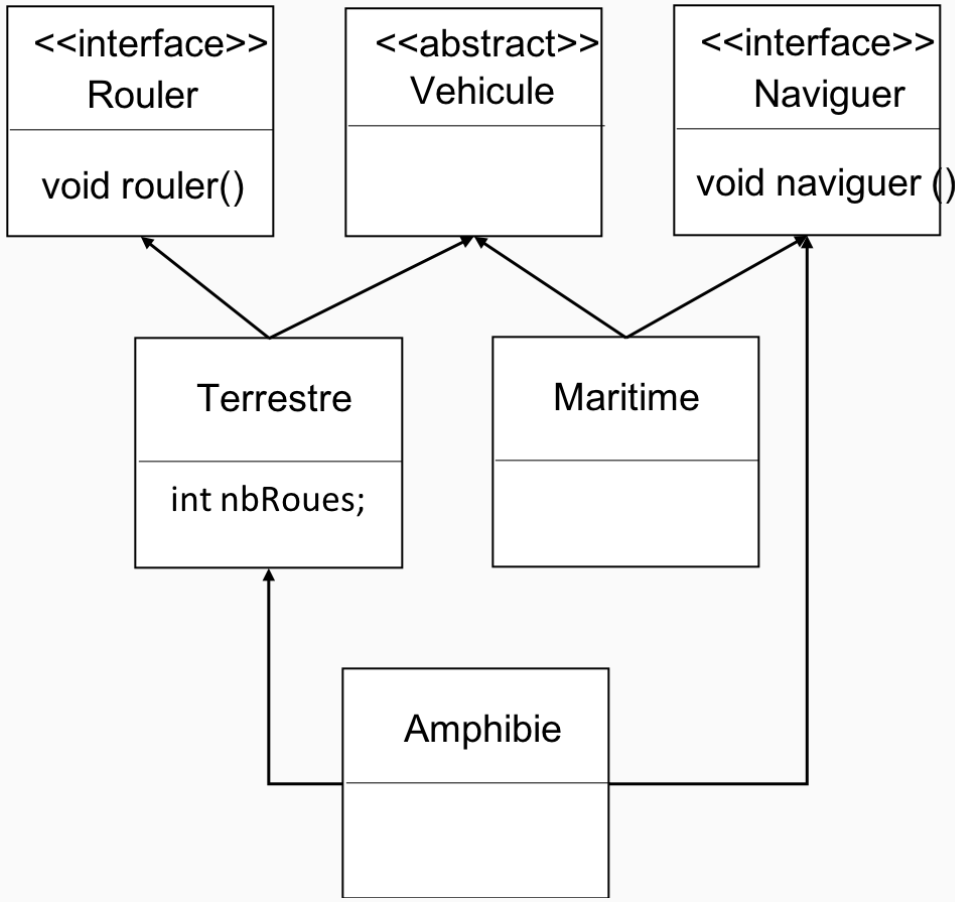
Utiliser une classe abstraite sans héritage multiple



- selon l'ordre de `Terrestre` ou `Maritime`, une voiture peut savoir naviguer et un bateau peut avoir des roues !
- cette solution ne fonctionne pas non plus !

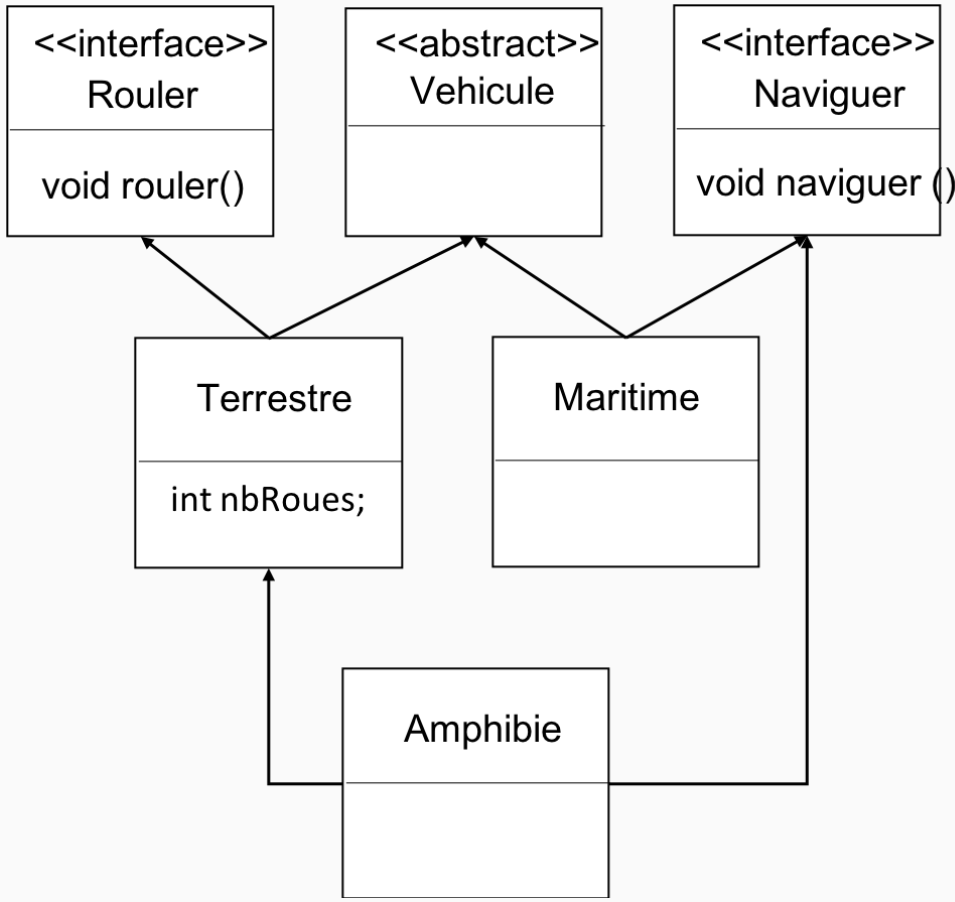
Modéliser les fonctionnalités avec des interfaces

- définition de 2 interfaces `Rouler` et `Naviguer`
- `Terrestre` possède un attribut `NbRoues` et implémente l'interface `Rouler`
- `Maritime` implémente l'interface `Naviguer`



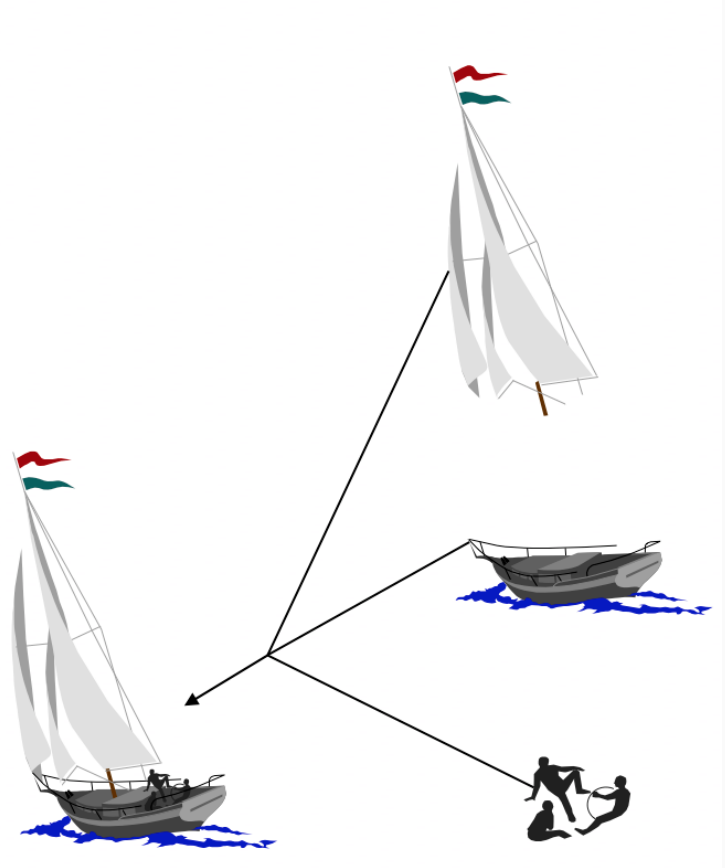
Modéliser les fonctionnalités avec des interfaces

- `Amphibie` dérive de `Terrestre` et possède un attribut `NbRoues` et implémente l'interface `Rouler`
- `Amphibie` implémente aussi l'interface `Naviguer`



Associations entre objets

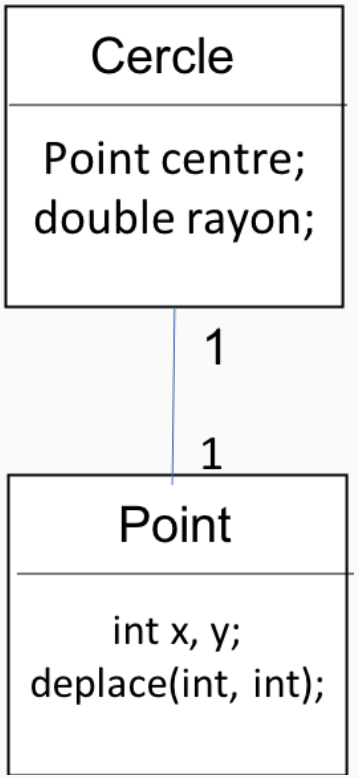
- une **association** définit une relation entre objets
- par exemple : voiture ↔ propriétaire
- l'association peut également représenter l'**agrégation** de différents objets pour en constituer un nouveau



Un objet attribut d'un autre

- un `Cercle c` possède un centre défini par un objet `Point centre`
- il est possible de déplacer `centre` depuis l'objet `c`

```
1  public class Cercle {
2   private Point centre;
3   private double rayon;
4   public Cercle(Point p, double r)
5   { centre = p; rayon = r; }
6   public void deplacer(int dx, int dy)
7   { centre.deplace(dx, dy); }
8  }
```



Les exceptions

Définition
Les exceptions représentent le mécanisme de gestion des erreurs intégré au langage Java

Mots-clés associées aux exceptions :

- détection et traitement des erreurs : `try`, `catch` et `finally`
- lever les exceptions ou les propager : `throw` et `throws`

Principe de fonctionnement

Lors de la détection d'une erreur :

```
1 public class TestException {
2   public static void main(String[] args) {
3     int i = 1;
4     int j = 0;
5     System.out.println("resultat = " + (i / j));
6   }
7 }
```

1. un objet qui hérite de la classe `Exception` est créé : une exception est **levée**
2. l'exception est propagé à travers la pile d'exécution
3. jusqu'à ce qu'elle soit traitée

Try, catch et finally

- Le bloc `try` rassemble les appels de méthodes susceptibles de produire des exceptions
- Plusieurs exceptions peuvent être capturées à la suite
- Eviter de laisser vide le traitement d'une exception

Try, catch et finally

- La clause `finally` définit un bloc qui sera **toujours exécuté**, qu'une exception soit levée ou non
- Ce bloc est **facultatif**

Try, catch et finally - exemple

```
1  public class TestException {
2    public static void main(String[] args) {
3      int i = 1; int j = 0;
4      try {
5        System.out.println("resultat = " + (i / j));
6      } catch (ArithmeticException e) {
7       System.out.println("Division par 0");
8      }
9    }
10 }
```

Throws

- Une autre façon de "gérer" une exception est d'ajouter le mot clé `throws` dans la signature de la méthode
- Dans ce cas on délègue la gestion de l'exception à la méthode appelante
- la gestion de l'erreur est reportée à plus tard, cela peut alléger le code localement mais cela oblige le développeur/développeuse qui utilise le code à les gérer lui/elle-même !

Classe `Throwable`

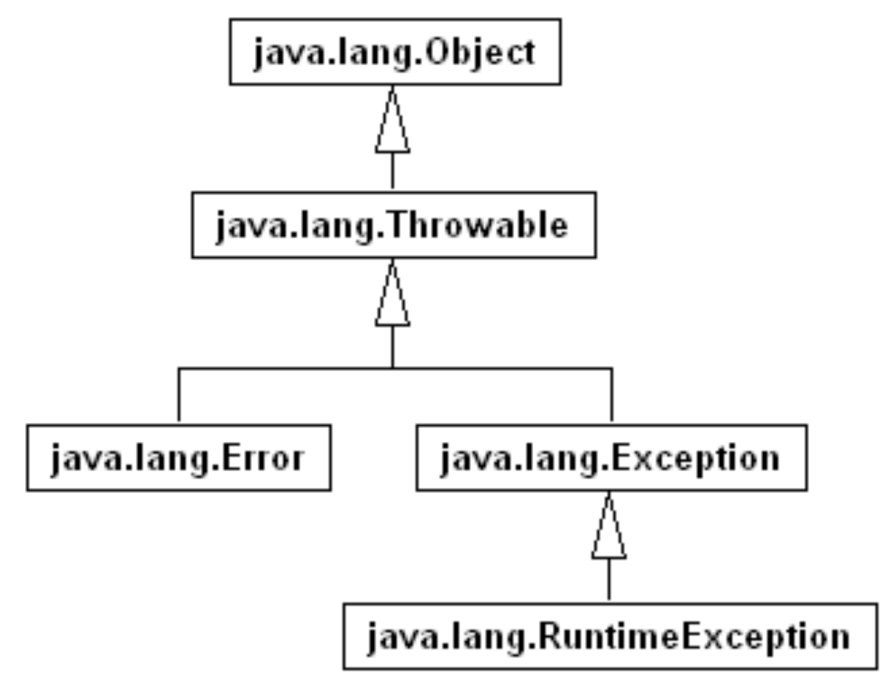
- Classe de base pour le traitement des erreurs
- Deux constructeurs : `Throwable()` et `Throwable(String)` dont la chaîne en paramètre permet de définir un message qui décrit l'exception et qui pourra être consulté dans un bloc `catch`

- Principales méthodes :
  - `String getMessage( )` : lecture du message
  - `void printStackTrace( )` : affiche l'exception et l'état de la pile d'exécution au moment de son appel
  - `void printStackTrace(PrintStream s)` : idem mais envoie le résultat dans un flux

Les classes Exception, RunTimeException et Error

- `Error (unechecked exception)` : modélise des erreurs d'exécution d'une application que l'on ne gère pas (`OutOfMemoryError`, `StackOverflowError`). Arrêt du programme
- `RunTimeException (unechecked exception)` : modélise des erreurs d'exécution d'une application que l'on ne gère pas. Elles signifient qu'une opération non prévisible a eu lieu (`NullPointerException`, `ArithmeticException`, `ArrayIndexOutOfBoundsException`)
- `Exception (checked exceptions)` : modélise les erreurs d'exécution que l'on doit prévoir

Les classes Exception, RunTimeException et Error



Définir ses propres exceptions

- Il faut créer une classe qui **hérite** de la classe `Exception` ou `RuntimeException`

```
1  public class SaisieErroneeException extends Exception {
2
3    public SaisieErroneeException() {
4      super();
5    }
6
7    public SaisieErroneeException(String s) {
8      super(s);
9    }
10 }
```

Utiliser ses propres exceptions

- Utiliser le mot clé `throw`, suivi d'un objet dont la classe dérive de `Throwable`
- Pour générer une exception dans une méthode avec `throw`, il faut l'indiquer dans la déclaration de la méthode, en utilisant le mot clé `throws`

Utiliser ses propres exceptions

```
1  public class TestSaisieErroneeException {
2    public static void controle(String chaine)
3     throws SaisieErroneeException {
4      if (chaine.equals("") == true)
5        throw new SaisieErroneeException("Chaine vide");
6    }
7
8    public static void main(java.lang.String[] args) {
9      String chaine1 = "bonjour";
10     try {
11       controle(chaine1);
12     } catch (SaisieErroneeException e) {
13       System.out.println("Chaine1 saisie erronee");
14     }
15   }
16 }
```