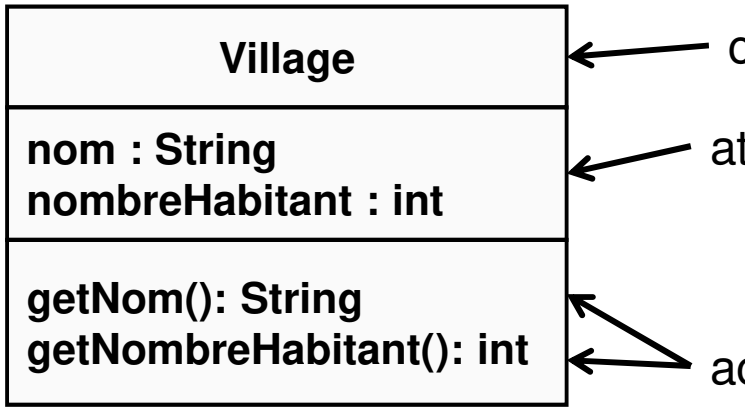
Cours 1 :
Rappel de points techniques

Rappel : UML / Java
Interfaces
Classes internes,
Exception,

Auteur : CHAUDET Christelle – Intervenants : BODEVEIX Jean-Paul, MIGEON Frédéric

Parallèle Diagramme de Classe /Java

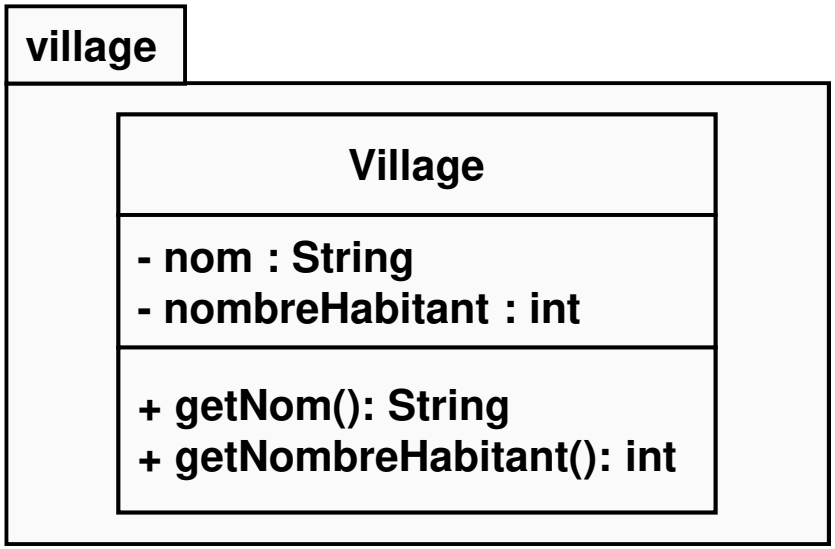
- **Classe = abstraction d'objets** ayant des propriétés (état, comportement et relations) communes




```
public class Village {
   String nom;
   int nombreHabitant;

   String getNom() {
      return nom;
   }

   int getNombreHabitant() {
      return nombreHabitant;
   }
}
```

Parallèle UML/Java : Visibilité

- **Visibilité =** ce qu'un objet a le droit de voir et de modifier. Les droits d'accès de chaque composant sont définis par un mot-clé, dit modifieur.

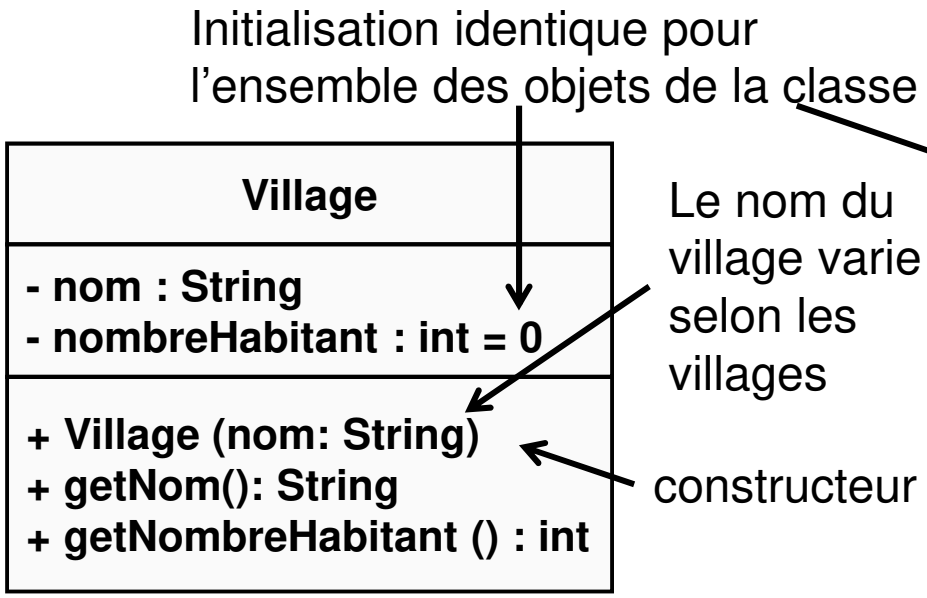


Symbole UML → Traduction JAVA
+ → public
# → protected
- → private
~ →

```
package village;

public class Village {
  private String nom;
  private int nombreHabitant;

  public String getNom() {
    return nom;
  }

  public int getNombreHabitant() {
    return nombreHabitant;
  }
}
```

Parallèle UML/Java : Constructeur

- **Constructeur =** permet de créer une instance d'une classe (un objet).



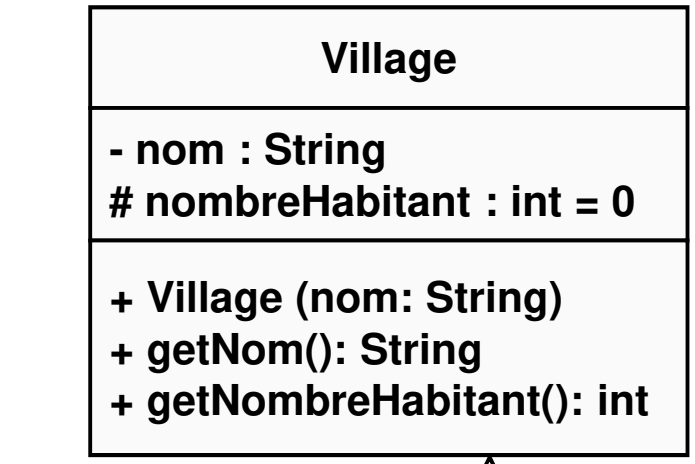

```
public class Village {
   private String nom;
   private int nombreHabitant = 0;

   public Village(String nom){
    this.nom = nom;
   }

   public String getNom() {
     return nom;
   }

   public int getNombreHabitant() {
     return nombreHabitant;
   }
}
```

Le constructeur est nécessaire SEULEMENT :

- si les valeurs initiales des attributs dépendent des instances (objets)
- OU s'il y a des initialisations à faire

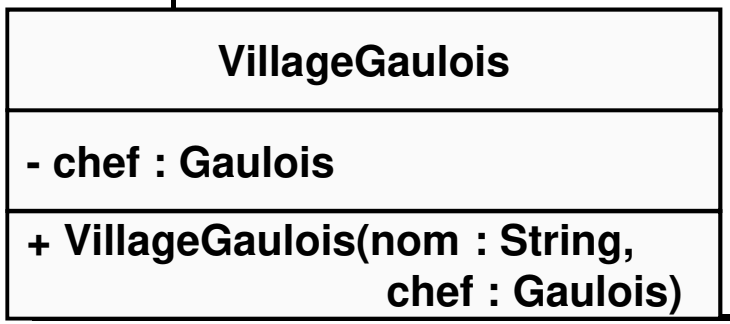
Parallèle UML/Java : Héritage

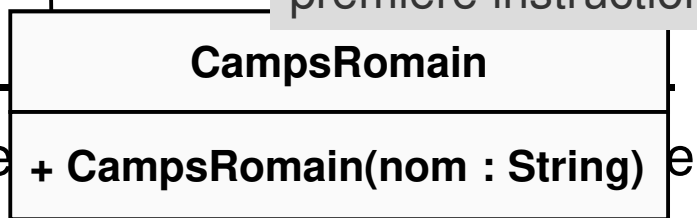
- **Héritage =** il s'agit ici de permettre à une classe (dite "classe fille") de récupérer les attributs et les méthodes d'une autre classe (dite "classe mère")




```
public class VillageGaulois extends Village {
   private Gaulois chef;
   public VillageGaulois(String nom, Gaulois chef) {
      super(nom);
      this.chef = chef;
      nombreHabitant = 1;
   }
}
```

héritage



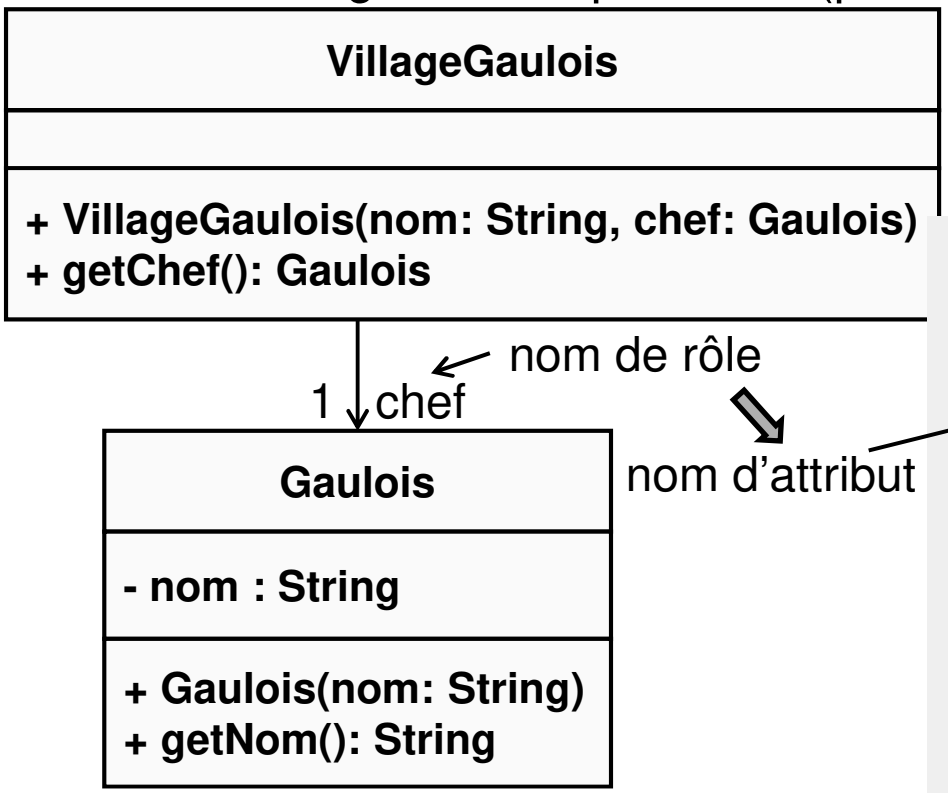


La classe mère Village a un constructeur avec un paramètre d'entrée (String nom) => La classe fille a donc OBLIGATOIREMENT un constructeur qui utilise le constructeur de la classe mère (super(nom));

Attention super(nom); doit être la première instruction du constructeur

Parallèle UML/Java : liens/attributs

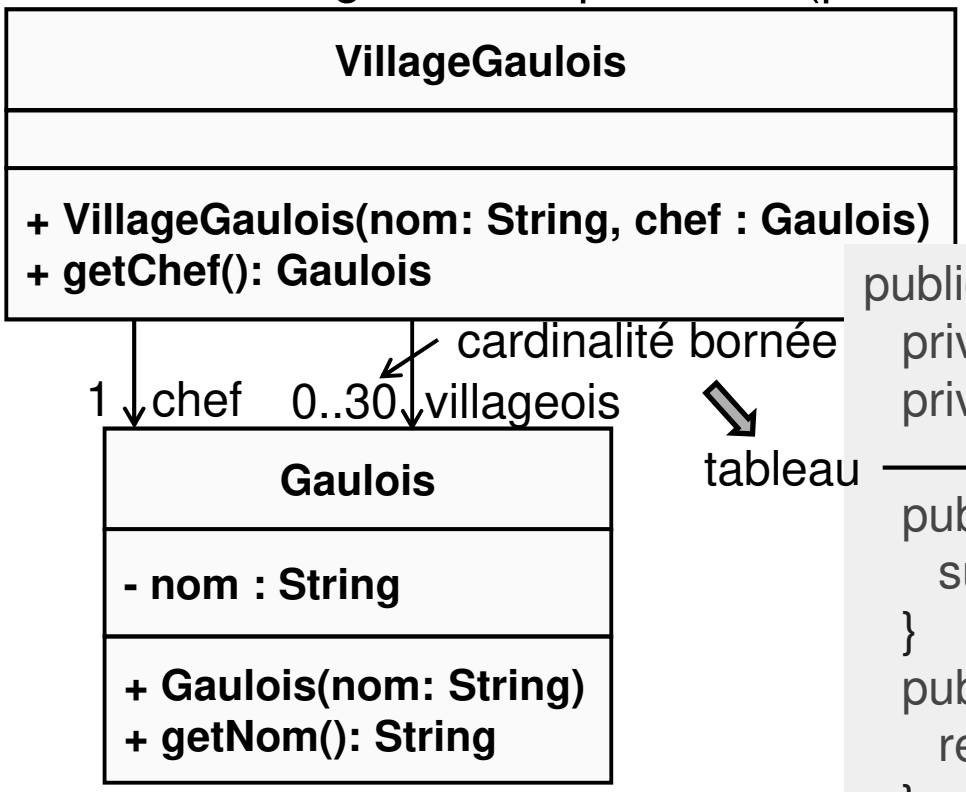
- **Association =** il y a association (dirigée ou pas) entre deux classes, lorsqu'une des deux classes sert à typer un attribut de l'autre classe. Le code de cette dernière peut donc contenir un envoi de message vers la première (par l'appel d'une méthode).




```
public class VillageGaulois extends Village {
  private Gaulois chef;

  public VillageGaulois(String nom,
    Gaulois chef) {
     super(nom);
     this.chef = chef;
     nombreHabitant = 1;
  }
     public String getChef() {
     return chef;
  }
}
```

Parallèle UML/Java : liens/attributs

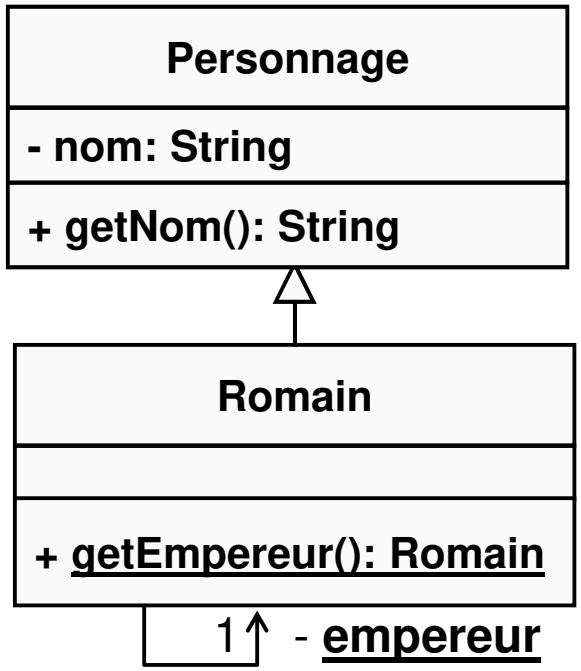
- **Association =** il y a association (dirigée ou pas) entre deux classes, lorsqu'une des deux classes sert à typer un attribut de l'autre classe. Le code de cette dernière peut donc contenir un envoi de message vers la première (par l'appel d'une méthode).




```
public class VillageGaulois extends Village {
  private Gaulois chef;
  private Gaulois[] villageois = new Gaulois[30];

  public VillageGaulois(String nom){
    super(nom);
  }
  public String getChef() {
    return chef;
  }
}
```

Parallèle UML/Java : Static

- **Static =** Pour une méthode, le modifieur static indique qu'elle peut être appelée sans instancier sa classe (NomClasse.methode()). Pour un attribut, le modifieur static indique qu'il s'agit d'un attribut de classe, et que sa valeur est donc partagée entre les différentes instances de sa classe.



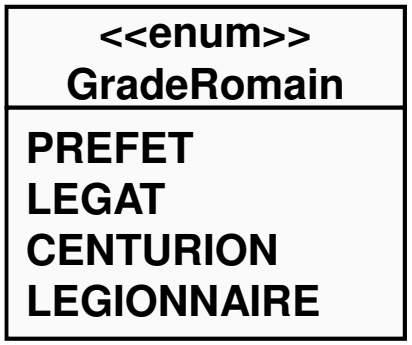

```
public class Romain extends Personnage {
  private static Romain empereur = new Romain("César");

  public Romain(String nom){
    super(nom);
  }

  public static Romain getEmpereur() {
    return empereur;
  }
}
```

```
System.out.println(Romain.getEmpereur());
```

⇒ César

Parallèle UML/Java

- **Enuméré** : type de données qui désigne un ensemble fini de valeurs ordonnées. Ces valeurs sont appelées : éléments, membre ou énumérateurs du type énuméré. Existe dans Java à partir de la version 1.5.

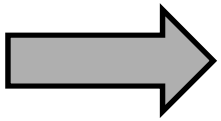



```
public enum GradeRomain {
   PREFET, LEGAT, CENTURION, LEGIONNAIRE;
}
```

Exemple d'utilisation d'un énuméré :

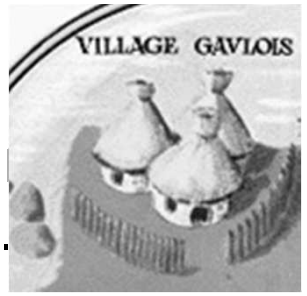
```
public static void main(String[] args) {
  System.out.println(GradeRomain.PREFET);
}
```

Affichage console

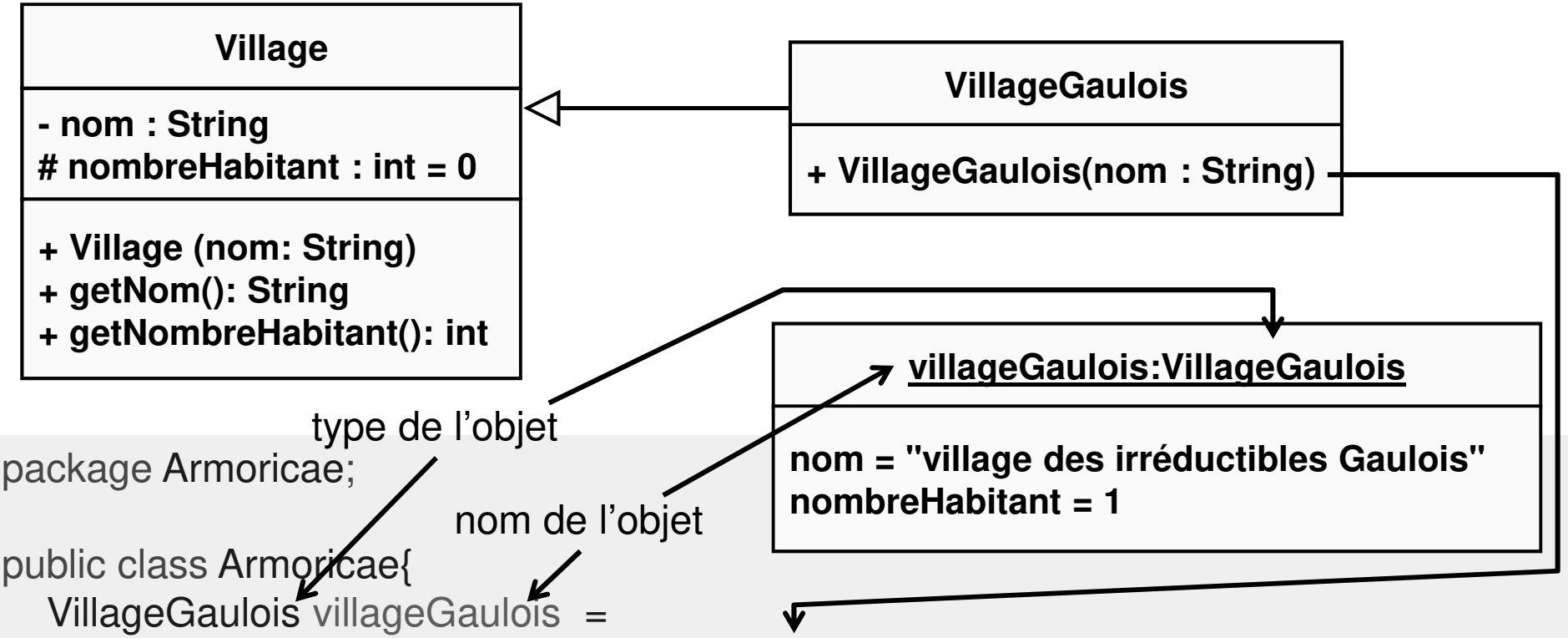


PREFET

Parallèle UML/Java : Les objets

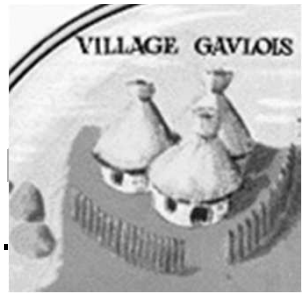


- Pour créer un objet il faut utiliser le constructeur de la classe.

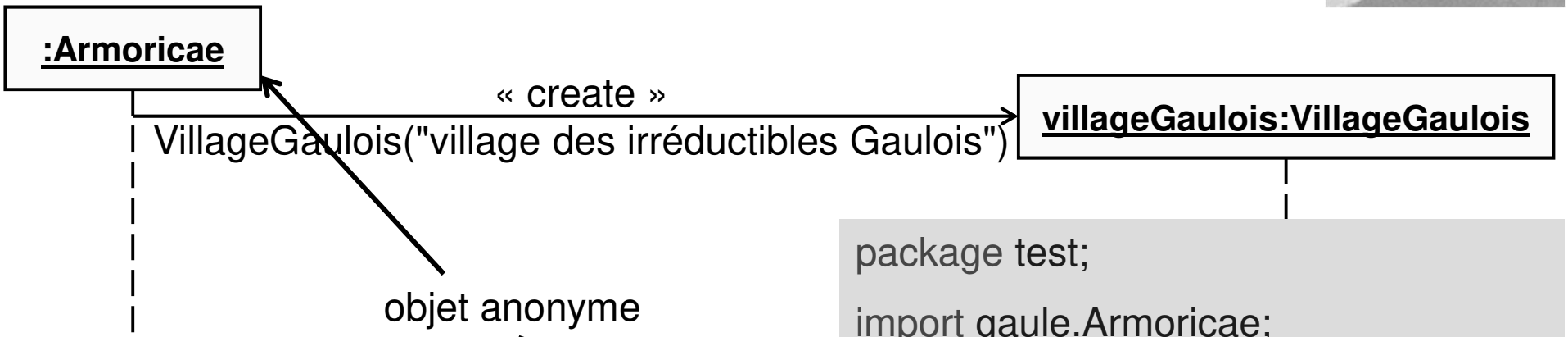



```
package Armoricae;

public class Armoricae{
   VillageGaulois villageGaulois  =
                                    new VillageGaulois("village des irréductibles Gaulois",
                                        new Gaulois ("Abraracourcix"));
…
}
```

Parallèle UML/Java : Les objets



- Pour créer un objet il faut utiliser le constructeur de la classe.




```
package test;

import gaule.Armoricae;

public class Test {
    public static void main(String[] args) {
        new Armoricae();
    }
}
```

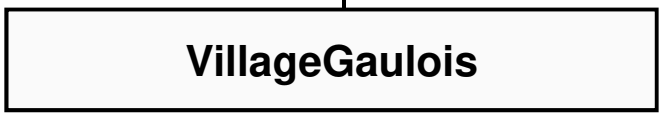
```
package gaule;

public class Armoricae {
    VillageGaulois villageGaulois  =
                                    new VillageGaulois("village des irréductibles Gaulois");
    CampRomain[] campsRomains = new CampRomain[4];
    int nombreCampsRomains = 0;
}
```

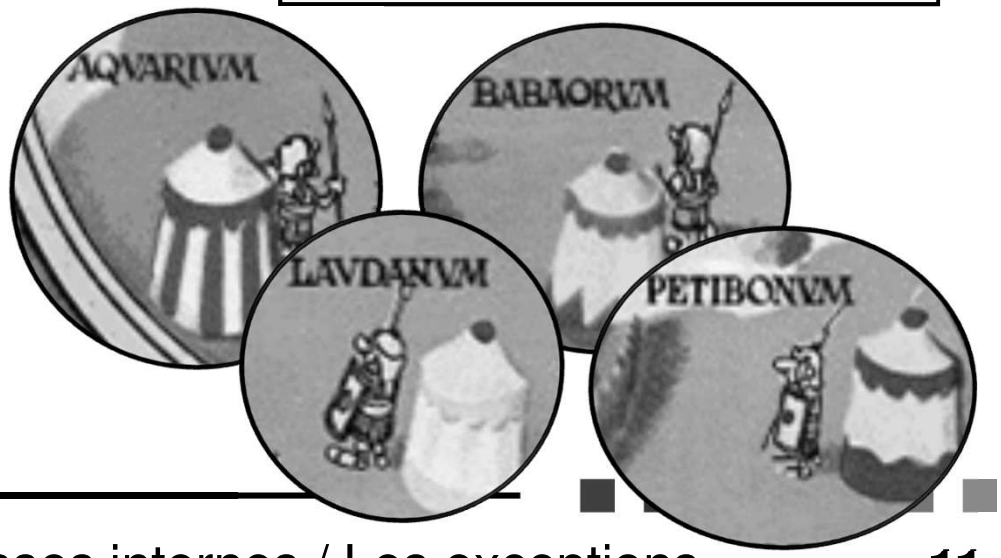
Parallèle UML/Java : Les classes abstraites

- **Classe abstraite** : c'est une classe qui n'est pas instanciable. Elle sert de base à d'autres classes dérivées (héritées).
- Deux notations :

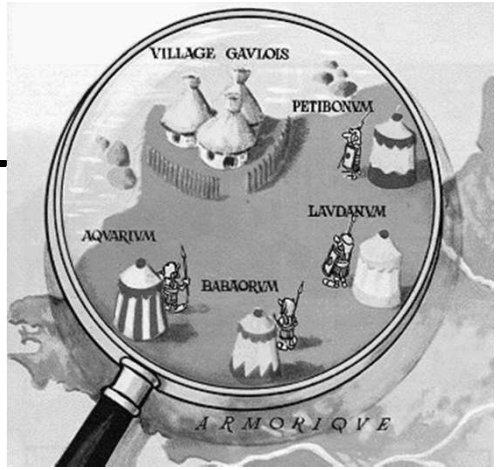
OU

Aucune instance :
un village Armoricain est soit romain, soit gaulois !

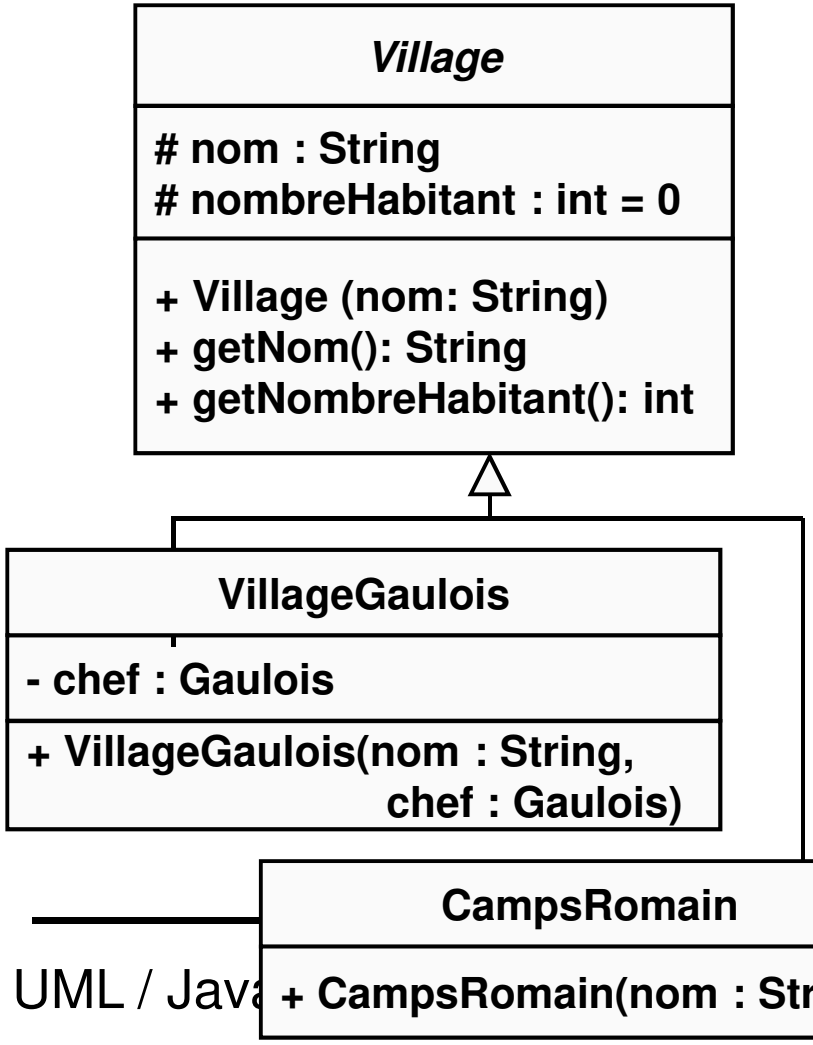




Parallèle UML/Java : Les classes abstraites



- **Classe abstraite** : c'est une classe qui n'est pas instanciable. Elle sert de base à d'autres classes dérivées (héritées).




```
public abstract class Village {
protected String nom;
protected int nombreHabitant = 0;

    public Village(String nom) {
        this.nom = nom;
    }

    public String getNom() {
        return nom;
    }

    public int getNombreHabitant() {
        return nombreHabitant;
    }
}
```

Interface

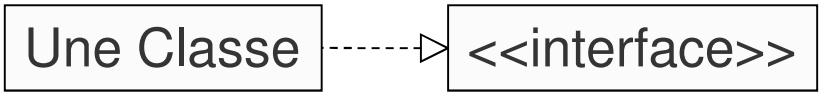
- Spécifiquement une interface ne contient :
  - que des **méthodes abstraites** et des **constantes**
  - pas d'attributs ni d'implantation de méthodes (hors Java 8).

- Mais alors à quoi ça sert ?
  - Sert de modèle de comportement à implanter par d'autres classes (spécification des services rendus)
  - Permet une grande flexibilité (changement de classe implémentant la même interface sans incidence sur le code)
  - Autorise toutes formes d'héritage (supporte l'héritage multiple)

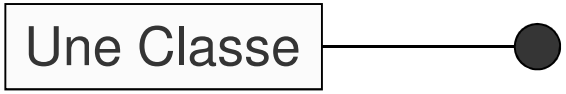
Syntaxe

- **Interface** : Spécification d'un comportement abstrait que des classes distinctes peuvent ensuite implanter.

- Notation UML
  - Notation à l'aide de stéréotype :



  - Notation iconifiée :



Syntaxe

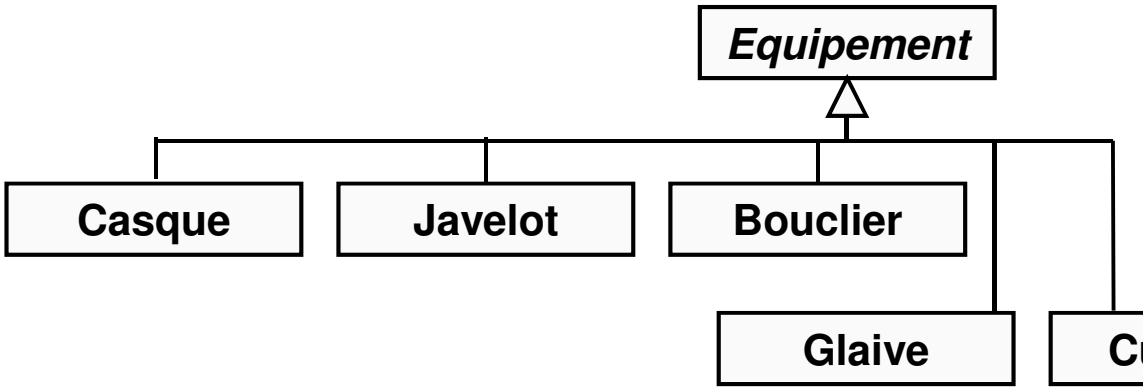
- Syntaxe
  `public interface <nom_interface> {}`

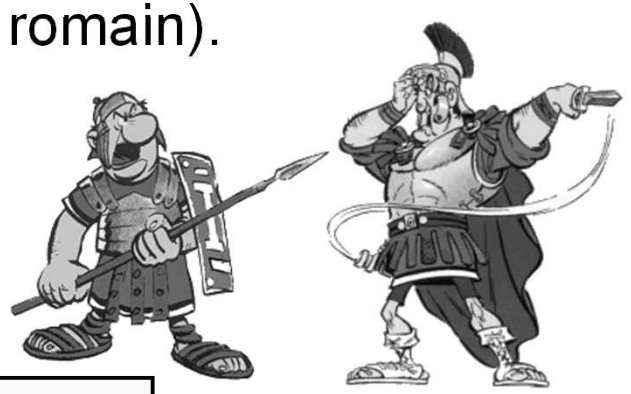
- Exemple :
  ```
  public interface IContrat {
        // corps de l'interface
  }
  ```

- Les interfaces ayant pour objectif de publier des services pour d'autres classes, les modificateurs suivants sur les méthodes sont :
  - non utilisables : protected, private, static et final.
  - par défaut : public et abstract.

Exemple applicatif

- Retournons chez les gaulois. Ils souhaitent pouvoir consulter leurs trophées (équipement d'un soldat romain).





- Il faut donc qu'une entité rende les services suivants :
  - ajouter un trophée,
  - récupérer l'ensemble des trophées du village,
  - récupérer l'ensemble des trophées apportés par un gaulois en particulier.
- Nous créerons donc une interface qui regroupe l'ensemble de ces services.

Création d'une interface

```
public interface GestionTrophee {
  void ajouterTrophee(Gaulois proprietaire,
                      Equipement trophee);
  String tousLesTrophees();
  String lesTrophees(Gaulois proprietaire);
}
```

- Deux aspirants à devenir informaticien (non confirmé du tout) **Goudurix** et **Keskonrix** souhaitent chacun apporter une implémentation à l'interface.

- En attendant leur création vous devez poursuivre votre programmation.

Goudurix

Keskonrix

Utilisation : design pattern Bridge

- Vous devez créer une classe « Musee ». Elle possède :
  - Un nom,
  - Un tarif d'entrée,
  - Un gestionnaire de trophée

Le type de l'attribut

correspond
.

```
public class Musee {
  private String nom;
  private int tarif;
  private                          gestionnaireTrophee;

   public Musee(String nom,
                                gestionnaireTrophee) {
      this.nom = nom;
      this.gestionnaireTrophee
                  = gestionnaireTrophee
   }
   public String getNom() {
      return nom;
   }
   public int getTarif() {
      return tarif;
  }
  public void setTarif(int tarif) {
    this.tarif = tarif;
  }
  public void ajouterTrophee(Gaulois proprietaire,
      Equipement trophee){
      gestionnaireTrophee
        .ajouterTrophee(proprietaire, trophee);
  }
  public String tousLesTrophees(){
    return gestionnaireTrophee.tousLesTrophees();
  }
  public String lesTrophees(Gaulois proprietaire) {
     return gestionnaireTrophee
       .lesTrophees(proprietaire);
} }
```

Implémentation de l'interface

Goudurix

- Goudurix, qui vient d'apprendre les tableaux en JAVA, est le premier à finir.
  Il propose l'implémentation suivante :

```
public class GoudurixGestion implements GestionTrophee {
  private Gaulois[] proprietaires = new Gaulois[30];
  private Equipement[] trophees = new Equipement[30];
  private int nombreDeTrophee = 0;


  public void ajouterTrophee(Gaulois proprietaire, Equipement trophee) {
    proprietaires[nombreDeTrophee] = proprietaire;
    trophees[nombreDeTrophee] = trophee;
    nombreDeTrophee++;
  }
…
```

Implémentation de l'interface

Goudurix

- La solution n'est pas très élégante, mais elle répond au besoin, vous pouvez donc tester votre musée.

```
public String lesTrophees(Gaulois proprietaire) {
    String leTrophee = "La liste de trophée de " + proprietaire.getNom() + " est ";
    for (int i = 0; i < nombreDeTrophee; i++)
        if(proprietaire.equals(proprietaires[i]))
            leTrophee += "\n-" + trophees[i];
    return leTrophee;
  }

public String tousLesTrophees() {
    String tousLesTrophees = "Tous les trophées du musée sont :\n";
    for (int i = 0; i < nombreDeTrophee; i++)
      tousLesTrophees += "- " + trophees[i] + "\n";
    return tousLesTrophees;
  }
}
```

Création d'un objet

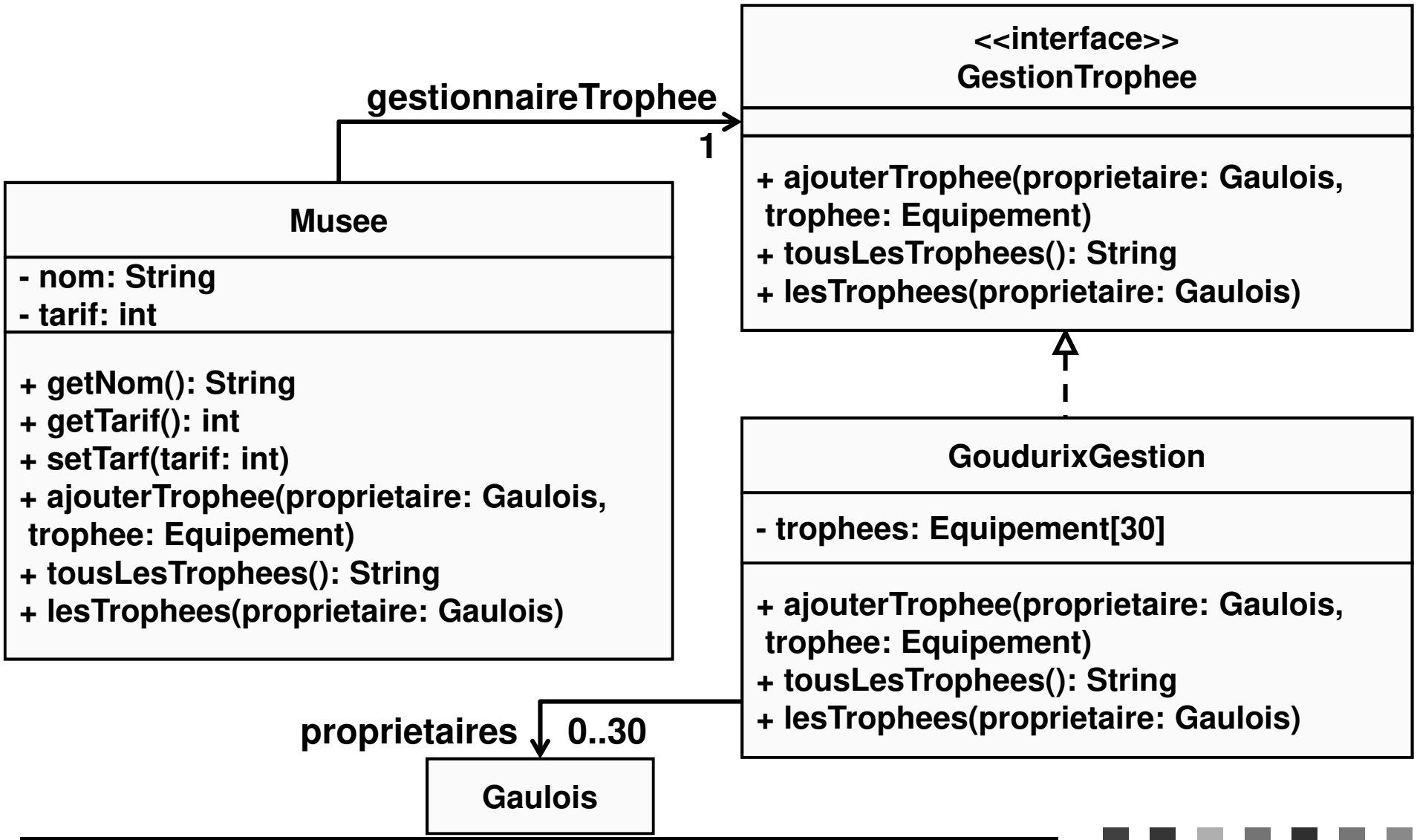
Goudurix

```
import villageGaulois.Gaulois;
import equipementRomain.Equipement;
import musee.GoudurixGestion;
import musee.Musee;

public class TestMusee {
  public static void main(String[] args) {
    Gaulois asterix = new Gaulois("Astérix");
    Gaulois obelix = new Gaulois("Obélix");
    Gaulois abraracourcix = new Gaulois("Abraracourcix");
    + création des armes
    GestionTrophee gestionnaireTrophees = new GoudurixGestion();
    Musee musee = new Musee("Museum", gestionnaireTrophees);
    musee.ajouterTrophee(asterix, bouclierMordicus);
    musee.ajouterTrophee(asterix, casqueAerobus);
    musee.ajouterTrophee(asterix, glaiveCornedurus);
    musee.ajouterTrophee(obelix, glaiveAerobus);
    musee.ajouterTrophee(abraracourcix, casqueHumerus);
    System.out.println("Le musé " + musee.getNom() + " est ouvert !\n");
    System.out.println(musee.tousLesTrophees());
    System.out.println(musee.lesTrophees(asterix));
  }
}
```

type : nom de l'interface,

objet créé à partir de la classe concrète

Diagramme de classes



Les interfaces (1/3)

- Attention il faut bien définir les services dont vous avez besoin avant de vous lancer dans la programmation.

```
public interface InterfaceA {
    void m1();
}
```

```
public class Classe2 implements InterfaceA {
    public void m1() {
        System.out.println("Méthode m1");
    }
}
```

m2() n'est pas dans l'interface donc inutilisable pour tous ceux qui utilisent votre interface !

```
public class Classe1 implements InterfaceA {
    public void m1() {
        System.out.println("Je suis la m1");
    }
    public void m2() {
        System.out.println("Je suis la m2");
    }
}
```

Les interfaces (2/3)

- Attention il faut bien définir les services dont vous avez besoin avant de vous lancer dans la programmation.

```
public interface InterfaceA {
    void m1();
}
```

```
public class Classe2 implements InterfaceA {
    public void m1() {
        System.out.println("Méthode m1");
    }
}
```

```
public class Classe1 implements InterfaceA {
    public void m1() {
        System.out.println("Je suis la m1");
        m2();
    }
    private void m2() {
        System.out.println(" de la ClasseA");
    }
}
```

Vous pouvez néanmoins utiliser des méthodes privées, utilisées par la méthode déclarée dans l'interface

Les interfaces (3/3)

```
public class Test {
    public static void main(String[] args) {
        InterfaceA objetA = new Classe1();
        objetA.m1();
    }
}
```

```
public class Test {
    public static void main(String[] args) {
        InterfaceA objetA = new Classe2();
        objetA.m1();
    }
}
```

- vous pouvez donc reprendre l'implémentation des services rendus autant de fois que vous le souhaitez

```
public interface InterfaceA {
    void m1();
}
```

```
public class Test {
    public static void main(String[] args) {
        InterfaceA objetA = new Classe1();
        objetA.m1();
        objetA.m2();
    }
}
```

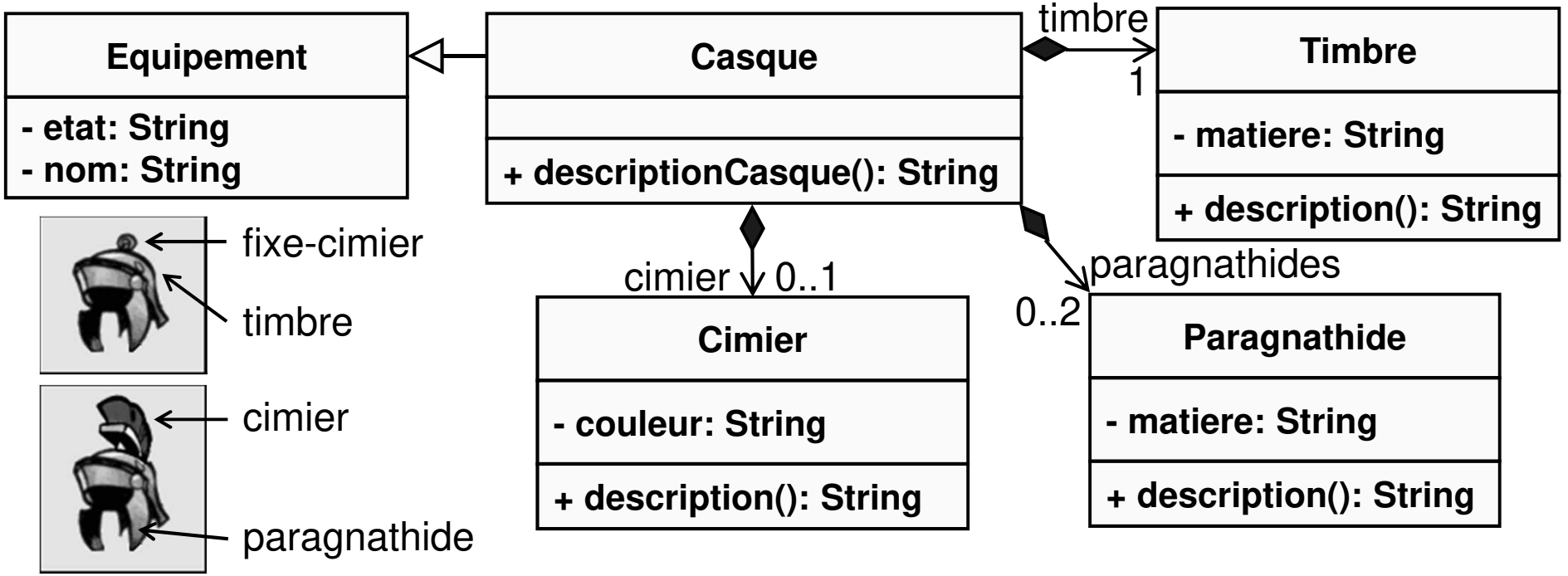
The method m2() is undefined for the type InterfaceA

- mais vous ne pouvez plus ajouter de services !

```
public class Classe1 implements InterfaceA {
    public void m1() {
        System.out.println("Je suis la m1");
    }
    public void m2() {
        System.out.println("Je suis la m2");
    }
}
```

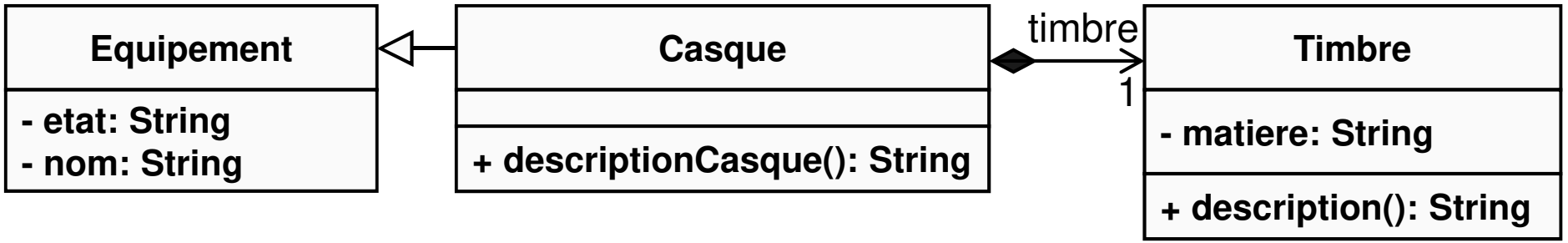
Classes imbriquées (1/4)

- Une classe interne (inner class) permet d'englober une ou plusieurs classes dans une autre classe.
- Sa représentation UML correspond à la composition.
- Par exemple : Le musée gaulois expose les casques rapportés lors des batailles.



Classes imbriquées (2/4)

- Les attributs publics ou privés et les paramètres de type de la classe externe sont **visibles** depuis la classe interne **si la classe interne n'est pas statique**.
- Les classes imbriquées :
  - non statiques sont implémentées en incluant une référence à l'instance englobante,
  - statiques sont d'un usage plus courant et plus efficaces.
- Reprenons l'exemple du casque romain en ne tenant compte que du sous-ensemble suivant :



Classes imbriquées (3/4)

```
public class Casque extends Equipement {
  private Timbre timbre;
  public Casque(String etat, String matiere) {
       super(etat, "casque");
       this.timbre = new Timbre(matiere);
   }
   public Timbre getTimbre() { return timbre; }
}
```

```
public class Timbre {
    private String matiere;
    private Timbre(String matiere) { this.matiere = matiere; }
    public String description() {
        return("Le casque romain est en " + etat +
        ". Son timbre est en " + matiere +".\n");
}   }
```

```
CasqueRomain casque = new Casque(" très mauvais état", "fer");
System.out.println(casque.getTimbre().description());
```

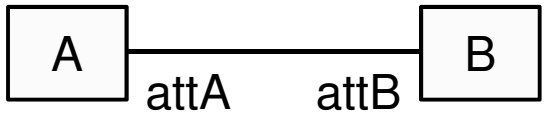
Classes imbriquées (4/4)

```
public class Casque extends Equipement {
  private Timbre timbre;
  public Casque (String etat, String matiere) {
    super(etat, "casque");
    this.timbre = new Timbre(matiere);   }
  private static class Timbre {
    private String matiere;
    private Timbre(String matiere) { this.matiere = matiere; }
    private String description() {
        return(" Son timbre est en " + matiere + "." );
    } }
  public String descriptionCasque() {
  return "Le casque romain est en " + etat + "." + timbre.description()
      +".\n";
} }
```

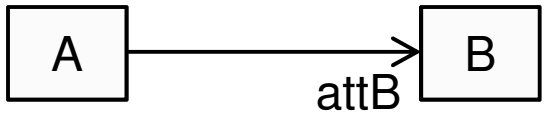
```
Casque casque = new Casque ("très mauvais état", "fer");
System.out.println(casque.descriptionCasque());
```

Récapitulatif des liens en UML

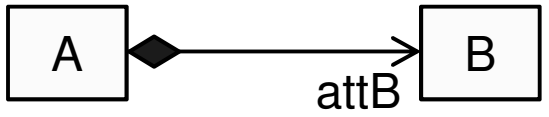
- Association
  - Bi-directionnelle



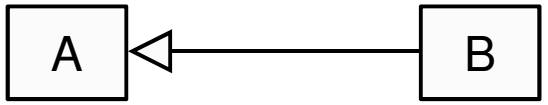
  - dirigée



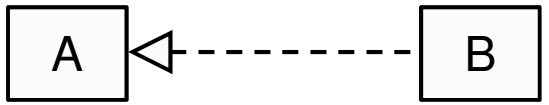
- Composition



- Généralisation



- Implémentation d'interface



Avec :

- *attA attribut de la classe B de type A*
- *attB attribut de la classe A de type B*

Nommage UML/Java

- Par convention que ce soit en UML ou en JAVA :
  - Les noms de **paquetage** sont entièrement en minuscule,
    Ex : village
  - Les noms de **classes** commencent par une majuscule, si le nom est composé de plusieurs mots ils seront accolés en mettant une majuscule à chaque nouveau mot.
    Ex : **V**illage**G**aulois
  - Les noms des **attributs** et des **variables** (attribut ou variable locale à une méthode) commencent par une minuscule, si le nom est composé de plusieurs mots ils seront accolés en mettant une majuscule à chaque nouveau mot.
    Ex : **n**ombre**H**abitant, **g**et**N**ombre**H**abitant

Nommage UML/Java

- Par convention que ce soit en UML ou en JAVA :
  - Les noms de **constantes** sont entièrement en majuscule, si le nom est composé de plusieurs mots ils seront accolés par le tiret du 8.
    Ex : COEFFICIENT_FORCE
  - Les noms des **types énuméré** commencent par une majuscule, si le nom est composé de plusieurs mots ils seront accolés en mettant une majuscule à chaque nouveau mot.
    Ex : **G**rade**R**omain
  - Les noms des **énumérateurs** d'un type énuméré sont entièrement en majuscule, si le nom est composé de plusieurs mots ils seront accolés par le tiret du 8.
    Ex : PREFET

Les exceptions

- Classes pour la gestion des exceptions
  - Classe Throwable
  - Classe Error
  - Classe Exception
  - Classe RuntimeException
- Personnalisation des exceptions
- Bonnes pratiques

```
Exception in thread "main" java.lang.NullPointerException
        at villageGaulois.VillageGaulois.ajouterHabitant(VillageGaulois.java:31)
        at villageGaulois.VillageGaulois.main(VillageGaulois.java:104)
```

Avant Propos

- Problème à résoudre : découpler le code utile de celui qui traite des situations exceptionnelles

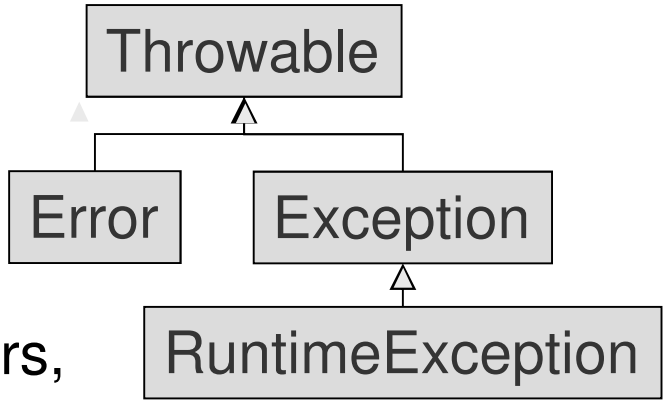
```
public static void main(String[] args) {
    int j = 20, i = 0;
    System.out.println(j/i);
    System.out.println("Poursuite du traitement");
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at cours1.EssaiException.main(EssaiException.java:7)
```

- Principe de l'exception :
  - repérer un morceau de code qui pourrait générer une exception,
  - capturer l'exception correspondante,
  - gérer l'exception : afficher un message personnalisé et continuer le traitement.

Qu'est-ce qu'une situation exceptionnelle ?

- Une situation exceptionnelle peut être assimilée à une erreur : **situation externe à la tâche principale** d'un programme.



- En Java, on distingue trois types d'erreurs, qui sont de degrés de gravité différents :
  - Erreurs graves : causent généralement l'arrêt du programme (classe **java.lang.Error**),
  - Checked exceptions : erreurs que le compilateur demande à traiter (classe **java.lang.Exception**),
  - Unchecked exceptions : erreurs que le compilateur ne peut pas reconnaître, « erreurs » de programmation (classe **java.lang.RuntimeException** qui hérite de **java.lang.Exception**).

Classe Error (1/2)

java.lang

Class Error

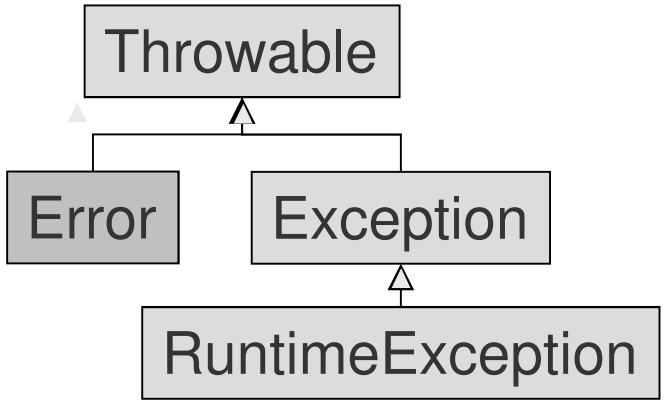
java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Error

All Implemented Interfaces:

Serializable

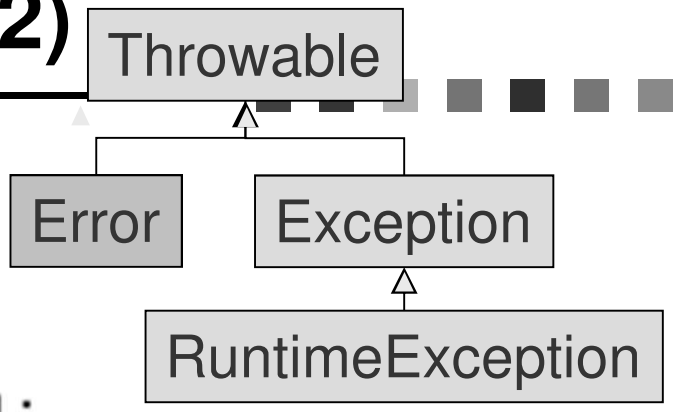
Direct Known Subclasses:

AssertionError, AWTError, CoderMalfunctionError, FactoryConfigurationError, LinkageError, ThreadDeath, TransformerFactoryConfigurationError, VirtualMachineError



- Cette classe est instanciée lorsque une erreur grave survient, c'est-à-dire une erreur empêchant la JVM de faire correctement son travail.
- Les objets de type Error ne sont **pas destinés à être traités** et il est même déconseillé de le faire.

Classe Error (2/2)



- Exemple

```
public class ErreurMemoire {

    public static void main(String[] args) {
        String[] tableau=new String[1000000000];
    }
}
```

```
Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
        at cours1.ErreurMemoire.main(ErreurMemoire.java:6)
```

java.lang

Class OutOfMemoryError

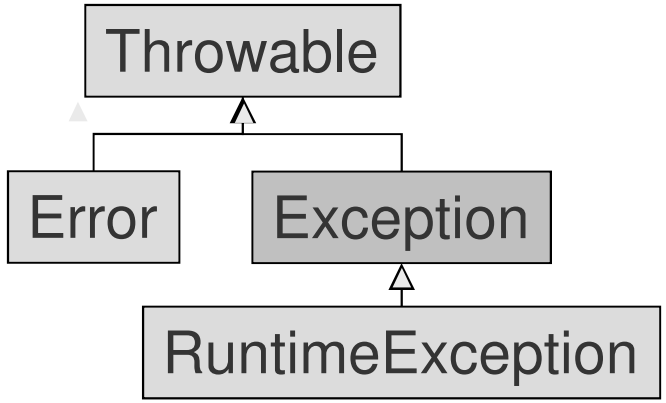
java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Error
      └ java.lang.VirtualMachineError
        └ java.lang.OutOfMemoryError

All Implemented Interfaces:
Serializable

Les erreurs, lorsqu'elles surviennent, ont la particularité **d'arrêter le thread en cours**, sauf si elles sont traitées par un catch (n'importe quel type de *Throwable* peut être "catché"). MAIS CETTE PRATIQUE DOIT ETRE EVITÉE.

Classe Exception (1/5)

- Les objets de type *Exception* (ou l'une de ses sous-classes) sont instanciés lorsqu'une erreur au niveau applicatif survient : **une exception est levée**.



- Exemple (1/4)
  Le compilateur demande à traiter ce type d'erreur :

Unhandled exception type FileNotFoundException Unhandled exception type IOException

```
public static void main(String[] args) {
    String chemin = "/Un/chemin/vers/un/fichier/qui/n'existe/pas";
    FileReader reader = new FileReader(chemin);
    int data = reader.read();
    do {
        System.out.println("Donneé suivante : " + (char) data);
        data = reader.read();
    } while (data != -1);
    reader.close();
    System.out.println("Fin des données");
}
```

Classe Exception (2/5)

- Lorsqu'une exception est levée, elle se propage dans le code en ignorant les instructions qui suivent et si aucun traitement ne survient, elle débouche sur la sortie standard.

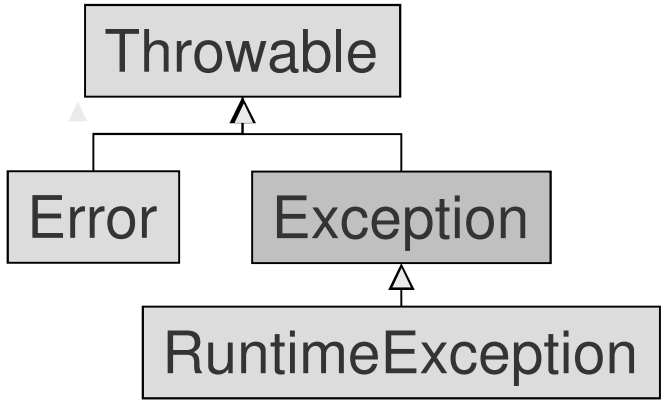
- Exemple (2/4)
  Si on s'obstine à ne pas corriger les erreurs on obtient :

```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
        Unhandled exception type FileNotFoundException
        Unhandled exception type IOException
        Unhandled exception type IOException
        Unhandled exception type IOException

        at cours1.PropagationException.main(PropagationException.java:11)
```

  L'instruction qui suit la levée de l'exception n'est pas exécutée
  => On n'obtient pas l'affichage *fin des données*

Classe Exception (3/5)

- L'exception étant de type *Exception* nous pouvons la traiter par un catch.




```
Unhandled exception type FileNotFoundException
Unhandled exception type IOException
```

java.io
Class FileNotFoundException

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.io.IOException
              └java.io.FileNotFoundException
```

java.io
Class IOException

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.io.IOException
```

Classe Exception (4/5)

- Les exceptions sont traitées via des blocs **try/catch** qui veulent littéralement dire essayer/attraper.
  - bloc try : instructions susceptibles de lever une exception
  - bloc catch : instructions qui seront exécutées en cas d'erreur
- Exemple (3/4)

```
public static void main(String[] args) {
    try {
        String chemin = "/Un/chemin/vers/un/fichier/qui/n'existe/pas";
        FileReader reader = new FileReader(chemin);
        int data = reader.read();
        do {
            System.out.println("Donneé suivante : " + (char) data);
            data = reader.read();
        } while (data != -1);
        reader.close();
        System.out.println("Fin des données");
    } catch (IOException ioE) {
        System.out
                .println("Une exception concernant les entrées/sorties a été levée");
    }
}
```

```
Une exception concernant les entrées/sorties a été levée
```

Classe Exception (5/5)

- On peut également mettre plusieurs blocs catch qui se suivent afin de fournir un traitement spécifique pour chaque type d'exception.
- Cela doit être fait en **respectant la hiérarchie** des exceptions.
- Exemple (4/4)

```
java.io.IOException
  └ java.io.FileNotFoundException
```

```
public static void main(String[] args) {
    try {
        String chemin = "/Un/chemin/vers/un/fichier/qui/n'existe/pas";
        FileReader reader = new FileReader(chemin);
        int data = reader.read();
        do {
            System.out.println("Donneé suivante : " + (char) data);
            data = reader.read();
        } while (data != -1);
        reader.close();
        System.out.println("Fin des données");
    } catch (FileNotFoundException fnfe) {
        System.out.println("Le fichier n'a pas été trouvé");
    } catch (IOException ioE) {
        System.out
                .println("Une exception concernant les entrées/sorties a été levée");
    }
}
```

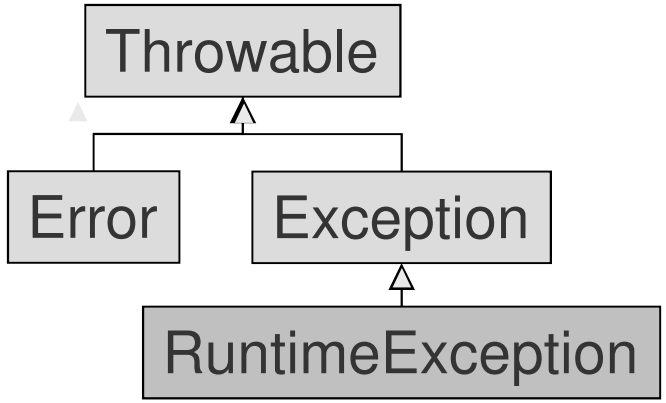
```
Le fichier n'a pas été trouvé
```

Classe runtimeException (1/2)

java.lang

Class RuntimeException

```
java.lang.Object
  └java.lang.Throwable
      └java.lang.Exception
          └java.lang.RuntimeException
```



- Erreurs qui peuvent survenir lors de l'exécution du programme.
- Le compilateur n'oblige le programmeur ni à les traiter ni à les déclarer dans une clause **throws**.

Direct Known Subclasses:

ArithmeticException, ArrayStoreException, BufferOverflowException, BufferUnderflowException, CannotRedoException, CannotUndoException, ClassCastException, CMMException, ConcurrentModificationException, DOMException, EmptyStackException, IllegalArgumentException, IllegalMonitorStateException, IllegalPathStateException, IllegalStateException, ImagingOpException, IndexOutOfBoundsException, MissingResourceException, NegativeArraySizeException, NoSuchElementException, NullPointerException, ProfileDataException, ProviderException, RasterFormatException, SecurityException, SystemException, UndeclaredThrowableException, UnmodifiableSetException, UnsupportedOperationException

Classe RuntimeException (2/2)

- Exemple :

```
public static void main(String[] args) {
    int valeur = 10;
    int part = 0;
    int erreur = valeur / part;
    System.out.println(erreur);
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at cours1.ErreurArithmetic.main(ErreurArithmetic.java:7)
```

```
public static void main(String[] args) {
    int valeur = 10;
    int part = 0;
    try {
        int erreur = valeur / part;
        System.out.println(erreur);
    }
    catch (ArithmeticException aE){
        System.out.println("Une exception a été levée");
    }

}
```

Type d'exception personnalisé (1/5)

- Création de son propre type d'exception : écrire une classe héritant de la classe **Exception**.
- Exemple : exception d'indice de tableau incorrect (appel à une case dont la valeur n'est pas initialisée).

```
public class ValeurNonInitialiseeException extends Exception {
    public ValeurNonInitialiseeException() {}
}
```

- Pourrait suffire, MAIS il est préférable d'utiliser les mêmes constructeurs que la classe Exception
  => simplifie leurs créations et l'encapsulation d'exception

Type d'exception personnalisé (2/5)

```
public class ValeurNonInitialiseeException extends Exception {
    private static final long serialVersionUID = 1L;
    //Crée une nouvelle instance de ValeurNonInitialiseeException
    public ValeurNonInitialiseeException() {}
    /* Crée une nouvelle instance de ValeurNonInitialiseeException
     * @param message Le message détaillant exception */
    public ValeurNonInitialiseeException(String message) {
        super(message);
    }
    /* Crée une nouvelle instance de ValeurNonInitialiseeException
     * @param cause L'exception à l'origine de cette exception */
    public ValeurNonInitialiseeException(Throwable cause) {
        super(cause);
    }
    /* Crée une nouvelle instance de IndiceIncorrectException
     * @param message Le message détaillant exception
     * @param cause L'exception à l'origine de cette exception */
    public ValeurNonInitialiseeException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

Type d'exception personnalisé (3/5)

- Utilisation d'un type d'exception provenant de la bibliothèque

```
public class TestIndiceTableau {
    int[] tableau = { 5, 2, 4, 0, 0 };

    public void getCase(int numeroCase) throws ArrayIndexOutOfBoundsException,
            ValeurNonInitialiseeException {

        if (numeroCase < 0)
            throw new ArrayIndexOutOfBoundsException("indice trop petit");
        else if (numeroCase >= tableau.length)
            throw new ArrayIndexOutOfBoundsException("indice trop grand");
        else if (tableau[numeroCase] == 0)
            throw new ValeurNonInitialiseeException(
                    "La case n'a pas été initialisée");
        else
            System.out.println(tableau[numeroCase]);
    }
}
```

L'attribut tableau contient 0 si la valeur n'a pas été initialisée

Précision de l'erreur dans le message

Type d'exception personnalisé (4/5)

- Utilisation d'un type d'exception personnalisé

L'attribut tableau contient 0 si la valeur n'a pas été initialisée

```
public class TestIndiceTableau {
    int[] tableau = { 5, 2, 4, 0, 0 };

    public void getCase(int numeroCase) throws ArrayIndexOutOfBoundsException,
            ValeurNonInitialiseeException {

        if (numeroCase < 0)
            throw new ArrayIndexOutOfBoundsException("indice trop petit");
        else if (numeroCase >= tableau.length)
            throw new ArrayIndexOutOfBoundsException("indice trop grand");
        else if (tableau[numeroCase] == 0)
            throw new ValeurNonInitialiseeException(
                    "La case n'a pas été initialisée");
        else
            System.out.println(tableau[numeroCase]);
    }
}
```

Type d'exception personnalisé (4/5)

- Test de l'exception personnalisée

```
public static void main(String[] args) {
    TestIndiceTableau liste = new TestIndiceTableau();
    try {
        liste.getCase(3);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Exception : " + e.getMessage());
```

```
public class TestIndiceTableau {
    int[] tableau = { 5, 2, 4, 0, 0 };

    public void getCase(int numeroCase) throws ArrayIndexOutOfBoundsException,
            ValeurNonInitialiseeException {

        if (numeroCase < 0)
            throw new ArrayIndexOutOfBoundsException("indice trop petit");
        else if (numeroCase >= tableau.length)
            throw new ArrayIndexOutOfBoundsException("indice trop grand");
        else if (tableau[numeroCase] == 0)
            throw new ValeurNonInitialiseeException(
                    "La case n'a pas été initialisée");
        else
            System.out.println(tableau[numeroCase]);
    }
```

Type d'exception personnalisé (5/5)

- Test de l'exception personnalisée

```
public class TestIndiceTableau {
    int[] tableau = { 5, 2, 4, 0, 0 };

    public void getCase(int numeroCase) throws ArrayIndexOutOfBoundsException,
            ValeurNonInitialiseeException {

        if (numeroCase < 0)
            throw new ArrayIndexOutOfBoundsException("indice trop petit");
        else if (numeroCase >= tableau.length)
            throw new ArrayIndexOutOfBoundsException("indice trop grand");
```

```
public static void main(String[] args) {
    TestIndiceTableau liste = new TestIndiceTableau();
    try {
        liste.getCase(3);
    } catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("Exception : " + e.getMessage());
    } catch (ValeurNonInitialiseeException e) {
        System.out.println("Exception : " + e.getMessage());
    }
    System.out.println("fin de l'application");
}
```

```
Exception : La case n'a pas été initialisée
fin de l'application
```

Les bonnes pratiques
Ne jamais ignorer une exception

- Une erreur fréquente : mettre un bloc catch vide sans aucune instruction afin de pouvoir compiler le programme.
- Conséquence : l'exception sera passée sous silence et le programme continuera de fonctionner ce qui peut déboucher sur des bugs incompréhensibles.
- Bonne pratique : traiter les exceptions dans les blocs catch ou au moins mettre un *printStackTrace*.

Ne jamais catcher "**Exception**" directement

```
public static void main(String[] args) {
    int valeur = 10;
    int part = 0;
    try{
        int erreur = valeur / part;
        System.out.println(erreur);
    }catch(Exception ex){
        ex.printStackTrace();
    }
    System.out.println("fin de la méthode");
}
```

```
java.lang.ArithmeticException: / by zero
        at cours1.NePasIgnorerUneException.main(NePasIgnorerUneException.java:8)
fin du programme
```

Les bonnes pratiques
Utiliser les exceptions standards

- Bien qu'il soit aisé de créer son propre type d'exception, l'API Java en fournit suffisamment en standard pour vous éviter cette tâche.

AclNotFoundException, ActivationException, AlreadyBoundException, ApplicationException, AWTException, BackingStoreException, BadLocationException, CertificateException, ClassNotFoundException, CloneNotSupportedException, DataFormatException, DestroyFailedException, ExpandVetoException, FontFormatException, GeneralSecurityException, GSSException, IllegalAccessException, InstantiationException, InterruptedException, IntrospectionException, InvalidMidiDataException, InvalidPreferencesFormatException, InvocationTargetException, IOException, LastOwnerException, LineUnavailableException, MidiUnavailableException, MimeTypeParseException, NamingException, NoninvertibleTransformException, NoSuchFieldException, NoSuchMethodException, NotBoundException, NotOwnerException, ParseException, ParserConfigurationException, PrinterException, PrintException, PrivilegedActionException, PropertyVetoException, RefreshFailedException, RemarshalException, RuntimeException, SAXException, ServerNotActiveException, SQLException, TooManyListenersException, TransformerException, UnsupportedAudioFileException, UnsupportedCallbackException, UnsupportedFlavorException, UnsupportedLookAndFeelException, URISyntaxException, UserException, XAException

Les bonnes pratiques
Utiliser l'encapsulation des exceptions

- L'exception qui apparaît sur la sortie standard n'est pas forcément celle qui est à l'origine de l'erreur
- Exemple

```
public class TestException {

    public static void main(String[] args) throws Exception {
        try{

            throw new Exception("1");

        }catch(Exception ex){

            throw new Exception("2");
        }
    }
```

Ne jamais lever "**Exception**" directement

```
Exception in thread "main" java.lang.Exception: 2
```

Les bonnes pratiques
Traitement interrompu

- Mauvaise pratique :
  code utilisant la réflexion pour instancier un objet

```
Class type = null;
Object object = null;

try {
        type = Class.forName("monpackage.MaClasse");
} catch (ClassNotFoundException e) {
    e.printStackTrace();
    // + traitement particulier à ClassNotFoundException
}

try {
    object = type.newInstance();
} catch (InstantiationException e) {
    e.printStackTrace();
    // + traitement particulier à InstantiationException
} catch (IllegalAccessException e) {
    e.printStackTrace();
    // + traitement particulier à IllegalAccessException
}

String string = object.toString();
```

- Le code est non sécurisé : l'exception n'interrompt qu'une partie du traitement.

  Exemple : si la méthode Class.forName() remonte une exception, l'objet type restera toujours à null, mais on tentera quand même d'appeler la méthode newInstance() dessus, ce qui provoquera une NullPointerException...

Les bonnes pratiques
Traitement interrompu

- Bonne pratique : les blocs try/catch doivent englober la totalité du traitement à interrompre en cas de problème.

```
try {

    Class type = Class.forName("monpackage.MaClasse"); // throws ClassNotFoundException
    Object object = type.newInstance(); // throws InstantiationException, IllegalAccessException
    String string = object.toString();

} catch (ClassNotFoundException e) {
    e.printStackTrace();
    // + traitement particulier à ClassNotFoundException
} catch (InstantiationException e) {
    e.printStackTrace();
    // + traitement particulier à InstantiationException
} catch (IllegalAccessException e) {
    e.printStackTrace();
    // + traitement particulier à IllegalAccessException
}
```

- De plus ce code a le mérite d'être bien plus lisible :
  - Tout le code utile est regroupé à l'intérieur du **try**.
  - Tous les **catch** sont au même niveau, ce qui pourrait permettre d'utiliser un traitement commun