L3 SRI - COO

Cours 2 : JAVA

Les interfaces
Les exceptions

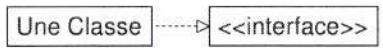
Auteur : CHAUDET Christelle

Interface

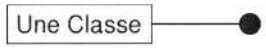
- Spécifiquement une interface ne contient :
  - que des **méthodes abstraites** et des **constantes**
  - pas d'attributs ni d'implantation de méthodes (hors Java 8).
- Mais alors à quoi ça sert ?
  - Sert de modèle de comportement à implanter par d'autres classes (spécification des services rendus)
  - 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.

Implémentation de l'interface

- 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

- 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 = "Le trophée de " + proprietaire.getNom() + " est ";
  int indiceProprietaire = 0;
  for (int i = 0; i < nombreDeTrophee; i++)
    if(proprietaire.equals(proprietaires[i]))
      indiceProprietaire = i;
  leTrophee += trophees[indiceProprietaire];
  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;
 }
}
```

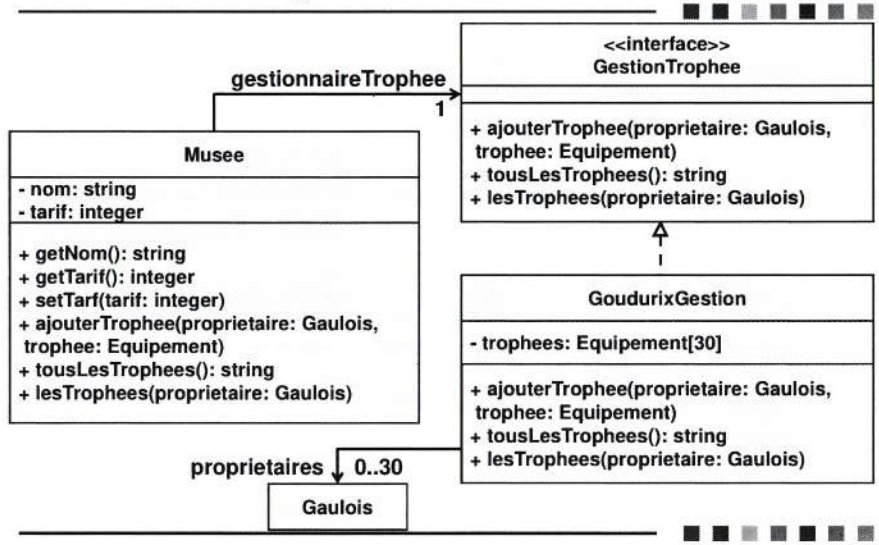
Création d'un objet


```
import villageGaulois.Gaulois;              import musee.GoudurixGestion;
import equipementRomain.Equipement;         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 créer à partir de la classe concrète

Diagramme de classes



Implémentation de l'interface

- 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

- 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 = "Le trophée de " + proprietaire.getNom() + " est ";
   int indiceProprietaire = 0;
   for (int i = 0; i < nombreDeTrophee; i++)
     if(proprietaire.equals(proprietaires[i]))
       indiceProprietaire = i;
   leTrophee += trophees[indiceProprietaire];
   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;
 }
}
```

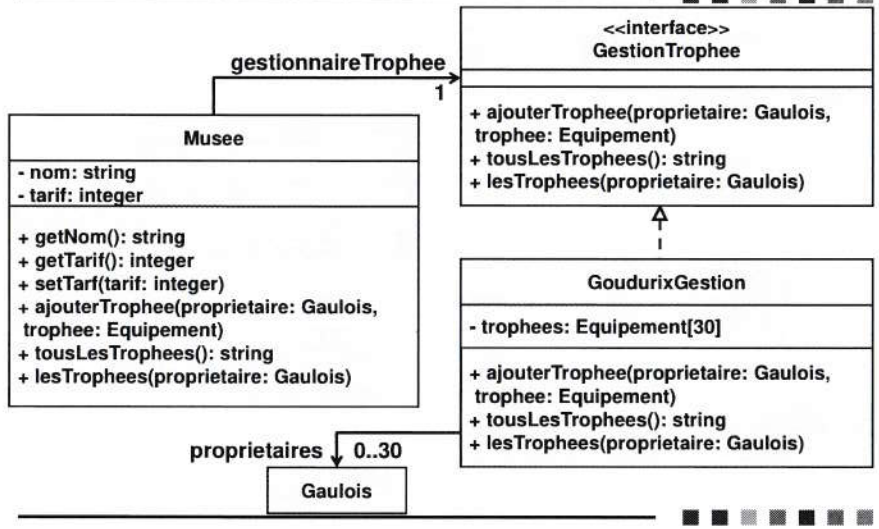
Création d'un objet


```
import villageGaulois.Gaulois;              import musee.GoudurixGestion;
import equipementRomain.Equipement;         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 créer à partir de la classe concrète

Diagramme de classes



Les interfaces (1/3)

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

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

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

Vous pouvez néanmoins utiliser des méthodes privée, utilisé par la méthode déclarée dans l'interface

```
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");
    }
}
```


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");
    }
}
```

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

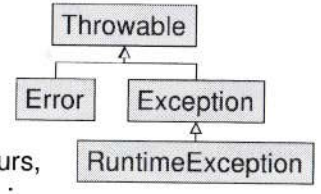
```
public static void main(String[] args) {
    int j = 20, i = 0;
    System.out.println(j/i);
    System.out.println("Poursuite du traitement");
}
```

```
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 ?

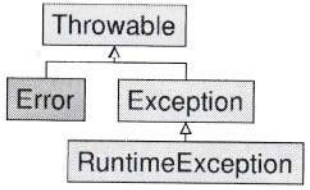
- Une situation exceptionnelle peut être assimilée à une erreur : **situation externe à la tâche principale** d'un programme.



- 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



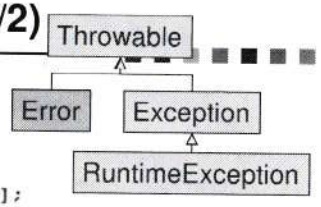
java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Error

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

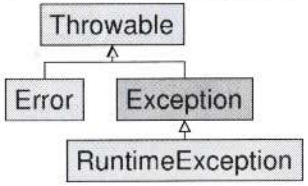
java.lang.Object
  └ java.lang.Throwable
    └ java.lang.Error
      └ java.lang.VirtualMachineError
        └ java.lang.OutOfMemoryError

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 :

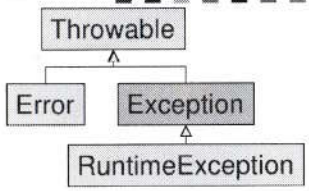
```
Exception in thread "main" java.lang.Error: Unresolved compilation problems:
        Unhandled exception type FileNotFoundException
        Unhandled exception type IOException
        Unhandled exception type IOException
        Unhandled exception type IOException

        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

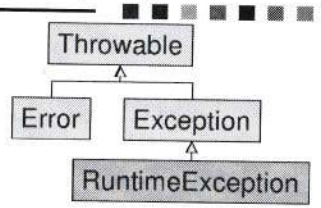
```
public static void main(String[] args) {
    try {
        String chemin = "/Un/chemin/vers/un/fichier/qui/n'existe/pas";
        FileReader reader = new FileReader(chemin);
        int data = reader.read();
        do {
            System.out.println("Donneé suivante : " + (char) data);
            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 pas 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 **RuntimeException**.
- Exemple : exception d'indice de tableau incorrect (appel à une case dont la valeur n'est pas initialisée).

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

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

Type d'exception personnalisé (2/5)

```
public class ValeurNonInitialiseeException extends RuntimeException{
    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 ValeurNonInitialiseeException
     * @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

```
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

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
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...

```
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();
```

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

Traitement interrompu

```
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
}
```


Clause finally

- Le mot clé **finally**, généralement associé à un **try**, permet l'exécution du code situé dans son bloc et ceci quelle que soit la manière dont s'est déroulée l'exécution du bloc try.
- Exemple

```
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");
    }
    finally {
        System.out.println("La valeur est : "+ valeur+", le nombre de part est : "+part);
    }
}
```

```
Une exception a été levée
La valeur est : 10, le nombre de part est : 0
```

Mise en garde sur le bloc finally

- Bonne pratique : évitez d'employer des instructions de rupture de séquence telles que *break*, *continue* ou *return* à l'intérieur d'un bloc try.
  Si c'est inévitable, assurez-vous qu'aucune clause **finally** ne modifie la valeur de retour de votre méthode.


```
public class ReturnFinally {

    public int methode1(){
        try{
            return 1;
        }catch(Exception e){
            return 2;
        }
    }

    public int methode2(){
        try{
            return 3;
        }finally{
            return 4;
        }
    }

    public static void main(String[] args) {
        ReturnFinally rf=new ReturnFinally();
        System.out.println("methode1 renvoie : "+rf.methode1());
        System.out.println("methode2 renvoie : "+rf.methode2());
    }
}
```

```
methode1 renvoie : 1
methode2 renvoie : 4
```


Les exceptions & les entrées/sorties

- Bonne pratique : Concernant les entrées/sorties, utiliser le pattern suivant.

```
try{
        //déclaration de la ressource
        try{
                //utilisation de la ressource
        }finally{
                //fermeture de la ressource
        }
}catch(ExceptionEntreeSortie ex){
        //traitement de l'exception
}
```


Les exceptions : libération de ressources (1/4)

- Il existe plusieurs types de ressources qui doivent être libérées explicitement, (appel à la méthode **close()**).
  - toutes les ressources gérées par le système d'exploitation (fichiers, sockets),
  - et assimilés (connexion JDBC).

- Res r = ... // 1. Création de la ressource
  try {
    // 2. Utilisation de la ressource
    ...
  } finally {
    // 3. Fermeture de la ressource
    r.close();
  }

Les exceptions : libération de ressources (2/4)

- Problème : pollution du code
  - chaque ressource doit être associée à un bloc **try/finally** -> plusieurs blocs d'indentation.
  - on doit utiliser un bloc **try/catch** supplémentaire pour un traitement des erreurs efficace, sous peine de ne pas intercepter toutes les exceptions...

```
try {
    InputStream input = new FileInputStream(in.txt);
    try {
        OutputStream output = new FileOutputStream(out.txt);
        try {
            byte[] buf = new byte[8192];
            int len;

            while ( (len=input.read(buf)) >=0 )
                output.write(buf, 0, len);
        } finally {
            output.close();
        }
    } finally {
        input.close();
    }
} catch (IOException e) {
    System.err.println("Une erreur est survenue lors de la copie");
    e.printStrackTrace();
}
```


Les exceptions : libération de ressources (3/4)

- **Java 7** :
- Le **try-with-resources** vient pallier tous ces problèmes via une nouvelle syntaxe plus simple. Les ressources déclarées dans un **try()** seront automatiquement libérées à la fin du bloc correspondant, quoi qu'il arrive.
- Le code précédent en **Java 7** :

```
try (InputStream input = new FileInputStream(in.txt);
  OutputStream output = new FileOutputStream(out.txt)) {
    byte[] buf = new byte[8192];
    int len;

    while ( (len=input.read(buf)) >=0 )
        output.write(buf, 0, len);
} catch (IOException e) {
    System.err.println("Une erreur est survenue lors de la copie");
    e.printStrackTrace();
}
```


Les exceptions : libération de ressources (4/4)

- **Java 7** :
- introduit un nouveau concept, les "Suppressed Exceptions :
  - gère proprement les multiples exceptions qui peuvent survenir lors de la fermeture des flux.
  - Si ces dernières surviennent alors qu'une exception est déjà en train de remonter, elles seront "ajoutées" à l'exception originale via **addSuppressed()** au lieu de la remplacer, ce qui permet d'éviter de perdre de l'information (elles seront bien visibles dans le stacktrace).
- Le **try-with-ressources** ne peut être utilisé qu'avec des instances d'objets implémentant la nouvelle interface **java.lang.AutoCloseable**. Cette dernière se contente de définir la méthode **close() throws Exception** qui sera utilisée pour libérer la ressource.

Les exceptions : MultiCatch

- **Java 5/6 :**

```
try {
    ...
} catch(IOException e) {
    // traitement
} catch(SQLException e) {
    // traitement
}
```

- **Java 7** : Si les traitements sont identiques, le code ci-dessus peut désormais s'écrire avec un seul et unique bloc catch

```
try {
    ...
} catch(IOException | SQLException e) {
    // traitement
}
```