UNIVERSITÉ PARIS 8
VINCENNES-SAINT-DENIS

Programmation Orientée Objet

Programmation Java

Saber HENI
saber.heni02@univ-paris8.fr
http://handiman.univ-paris8..fr/~saber/

ORACLE
Certified Professional
Java SE 6 Programmer

Plan du cours

- Chapitre 1 : Introduction au langage Java
- Chapitre 2 : Classes et objets
- Chapitre 3 : Concepts de base de l'OO
- **Chapitre 4 : La gestion des exceptions**
- Chapitre 5 : Les Entrées-sorties
- Chapitre 6 : Les classes de base de Java
- Chapitre 7 : Les interfaces graphiques
- Chapitre 8 : Les collections

UNIVERSITÉ PARIS 8
VINCENNES-SAINT-DENIS

Chapitre 4

La gestion des exceptions

Exemple

- Tapez ce code sur votre machine

```
public class TestExceptions {
        static String s;

        public static void divisionParZero () {
                int x = 12;
                int y = 0;

                int z = x/y;
                System.out.println(z);
        }

        public static void objetNull () {
                System.out.println(s.length());
                System.out.println("Coucou");
        }
        public static void main (String [] args) {
                divisionParZero();
                objetNull ();

        }
}
```

Introduction

- Une exception est une erreur se produisant dans un programme qui conduit le plus souvent à l'arrêt de celui-ci.

- Une exception est le gros message en rouge affiché dans la console suite à une erreur d'exécution : Ce message s'affiche car l'exception n'a pas été ***capturée/Interceptée***.

- Le fait de gérer les exceptions s'appelle aussi « la capture d'exception ».
  - Le principe consiste à repérer un morceau de code (par exemple, une division par zéro) dangereux, de capturer l'exception correspondante et enfin de la traiter,
  - C'est-à-dire d'afficher un message personnalisé et de continuer l'exécution.

Rôle des exceptions

- Une exception est chargée de signaler un comportement ***exceptionnel*** (mais prévu) d'une partie spécifique d'un logiciel.

- Dans les langages de programmation actuels, les exceptions font partie du langage lui-même.

- C'est le cas de Java qui intègre les exceptions comme une classe particulière : la classe Exception.
  - Cette classe contient un nombre important de classes dérivées.

Le bloc try{...} - catch{...} - 1

- Dans l'exemple précédent
  - Lorsque l'exception a été levée, le programme s'est arrêté.
  - Le type d'exceptions sont :
    - ArithmeticException dans le cas d'une division par zéro.
    - NullPointerException dans le cas d'accès à une référence nulle.
  - Les instructions qui suivent la ligne qui a causé le problème ne se sont pas exécutés.

Qu'est ce qu'il faut faire pour que le programme continue à s'exécuter quand même?

Le bloc try{…} - catch{…} - 2

- Il faut capturer les exceptions, avec un bloc
  - **try{…} catch{…}**,
  - puis réaliser un traitement en conséquence.
- Soit le code suivant

```
public class TestExceptions {
        public static void divisionParZero () {
                int x = 12;
                int y = 0;
                try {
                        int z = x/y;
                        System.out.println(z);
                        System.out.println ("Ligne dans le try");
                } catch (ArithmeticException e) {
                         System.err.println ("Exception : div par 0");
                }
                System.out.println ("Ligne après le catch");
        }
        public static void main (String [] args) {
                divisionParZero();
                System.out.println ("Ligne après la méthode");

        }
}
```

Le bloc try{…} - catch{…} - 3

- **Commentaires**
  - Pour capturer une exception
    - Mettre le code dangereux et le reste du code qui en dépend dans le bloc **try {…}**
    - Juste après déclarer un bloc **catch {…}** avec l'exception correspondante.
      - Afficher ou journaliser un **message d'erreur personnalisé(1)** ou le **message d'erreur de l'exception(2)** ou encore **imprimer la pile d'exécution(3)**.
      - Sachant qu'on peut combiner les messages.
    - Mettre le reste des instructions du programme après le **catch**.

(1) Comme celui de l'exemple précédent.

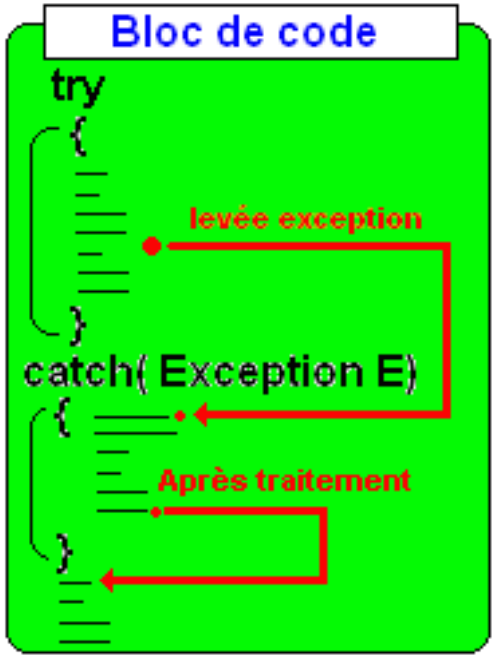
(2) `System.err.println(e.getMessage ()); // 'e' est le nom de l'instance de l'exception.`

(3) `e.printStackTrace();`

Le bloc try{...} - catch{...} - 4

- Qu'est ce qui se passe quand on capture une exception?

- Au moment de l'exécution de l'instruction de la division
  - Une exception se lève.
  - L'exécution du reste du bloc try s'interrompt.
  - Le programme passe directement au bloc catch
    - Il instancie un objet ArithmeticException et l'associe à 'e'.
    - Exécute le bloc d'instruction du catch.
- Le programme suit son cours.



Les types d'exceptions - 1

- **Java propose toute une hiérarchie d'exception de natures différentes,**
  - Les erreurs (la classe Error) – Techniquement ne sont pas des exceptions.
    - Représentent des erreurs graves intervenue dans la machine virtuelle Java ou dans un sous système Java.
    - L'application Java s'arrête instantanément suite à une erreur.
    - ex. *InternalError, OutOfMemoryError, StackOverflowError*…
    - Les erreur peuvent être capturés mais, ca sert à rien parce que la JVM ne peut pas survivre suite à une erreur. Elle s'arrête quand même.
  - Les exceptions non vérifiées (Unchecked exceptions)
    - Ces exceptions concernent des erreurs de programmation qui peuvent survenir à de nombreux endroits dans le code.
    - ex. *NullPointerException, NumberFormatException, IndexOutOfBoundsException, NumberFormatException, ClassCastException …*

Les types d'exceptions - 2

- Elles héritent de la classe **RuntimeException.**
- **Ce type d'exception n'exigent pas au programmeur de les traiter (capturer/intercepter ou propager).**

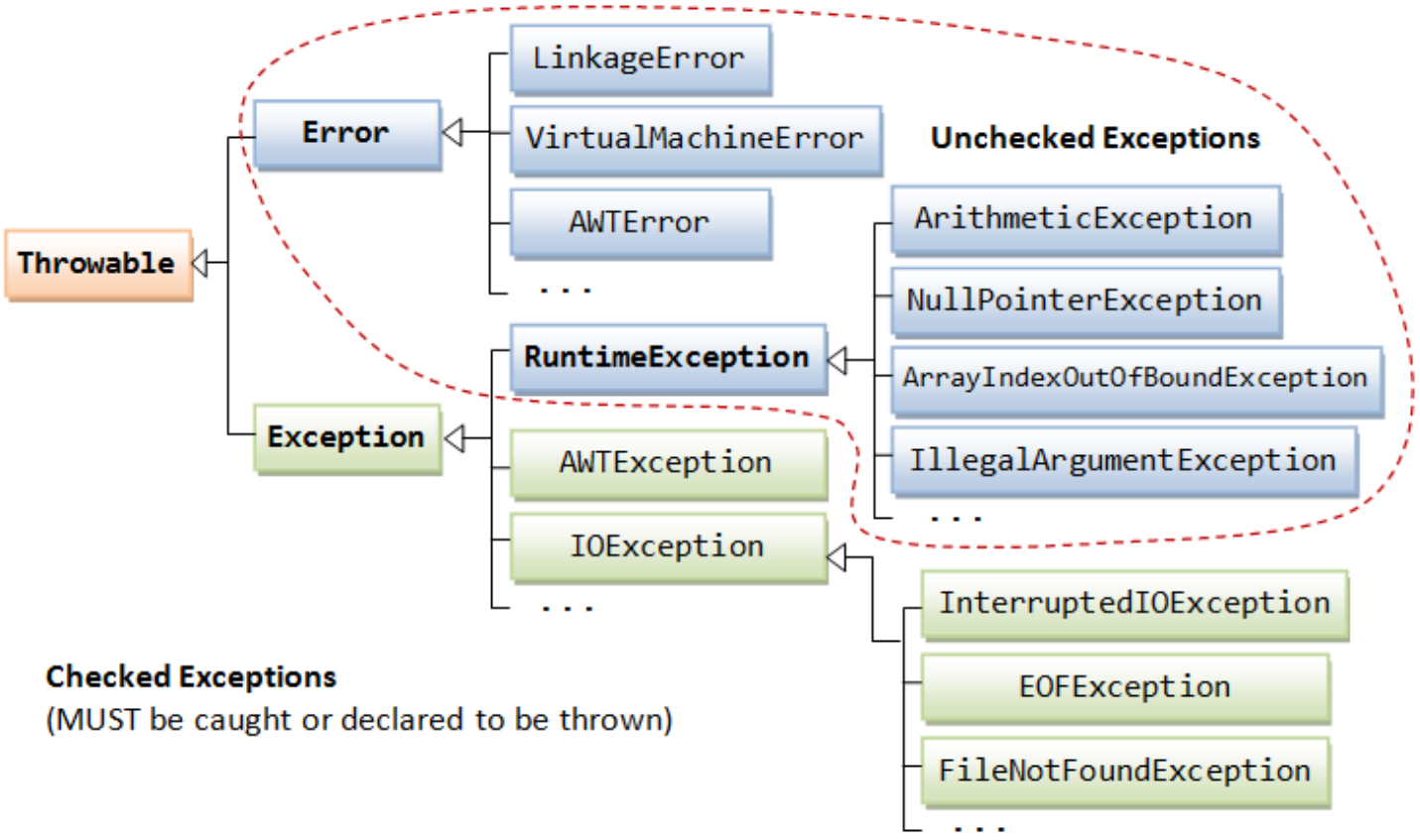
– Les exceptions vérifiées (Checked exceptions)

- Tout le reste ce sont des exceptions que le programmeur doit obligatoirement capturer/intercepter (Le compilateur l'exige) pour que le programme compile.

– Les exceptions personnalisées

- Java offre la possibilité de créer des exceptions personnalisés en héritant de la classe Exception.

Hiérarchie des classes d'Exceptions



Règles d'utilisation des Exceptions v1.0 - 1

- En Java on trouve deux types d'exceptions : Checked (vérifiées) et uncheked (non vérifiées).
- Les exceptions que le compilateur exige de traiter (Checked exceptions) sont tous les descendants de la classe **Exception** à l'exception des *sous-types* de **RuntimeException** qu'on peut ne pas traiter.
- Un **try** peut avoir plusieurs **catch**.
- Un bloc **catch** intercepte l'exception spécifié et tous ces *sous-types*.
- Les blocs **catch** doivent être triés du plus spécifique au plus générique.

Règles d'utilisation des Exceptions v1.0 - 2

- Une exception ne sera jamais exécutée si elle est précédée par un de ses *super-types*.
- Le compilateur se plaindra si vous mettez une exception dans le **catch** qui ne peut jamais être exécutée.

Règles d'utilisation des Exceptions v1.0 - 2

- **Illustration**.

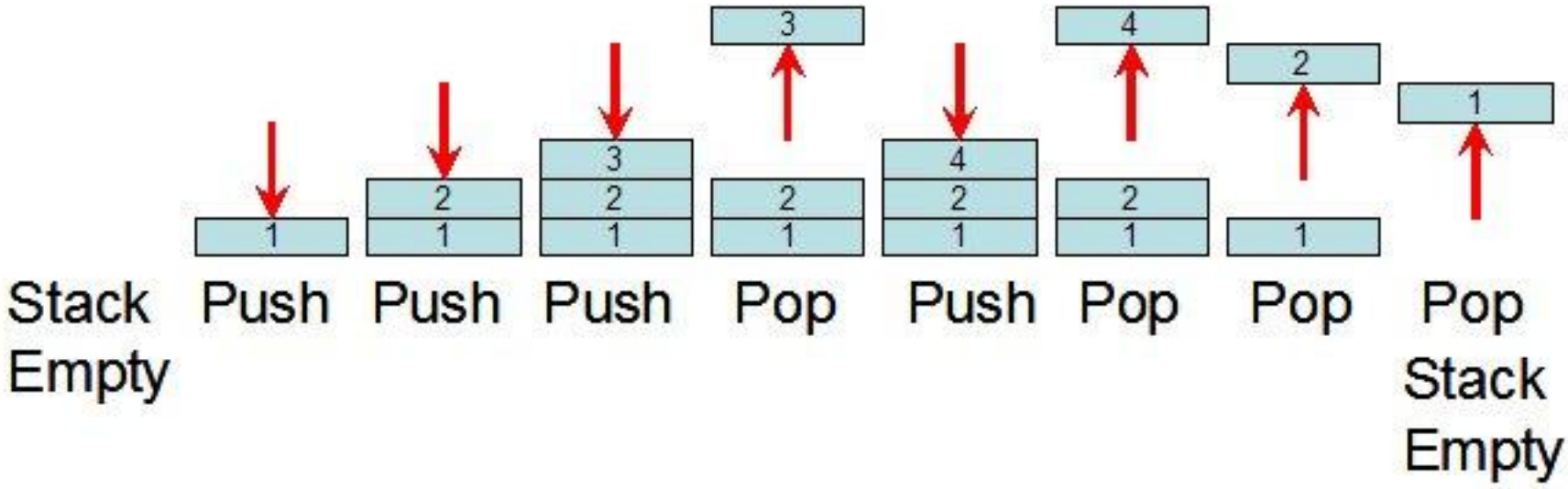
```
try {

        < bloc de code à protéger générant un objet exception>

}
catch ( TypeException1 E ) {
        <Traitement TypeException1 >
}
catch ( TypeException2 E ) {
        <Traitement TypeException2 >
}
.....
catch ( TypeExceptionk E ) {
        <Traitement TypeExceptionk >
}
```

La pile d'exécution Java (The stack) - 1

- Principe du LIFO (Last In First Out) ou encore FILO



Push → Empiler

Pop → dépiler

La pile d'exécution Java (The stack) - 2

- Soit ces exemples

1

```
public static void main (String [] args) {
        divisionParZero();
        objetNull ();

}
```

2

```
public class TestPile {
        public static A () { C (); D (); }
        public static B () { D (); }
        public static C () { D (); }
        public static D () { }
        public static void main (String [] args) {
                A ();
                B ();
        }
}
```

La pile d'exécution Java (The stack) - 3

- Dans L'exemple 1,
  1. La JVM empile la méthode main;
  2. Exécute son bloc de code;
  3. La méthode main appelle la méthode DivisionParZero ().
  4. La JVM, empile la méthode;
  5. Exécute son code;
  6. Quand l'exécution de la méthode se termine, la JVM la dépile.
  7. Elle continue L'exécution de la méthode main;
  8. Cette dernière méthode fait appel à objetNull().
  9. La JVM empile la méthode,
  10. Exécute tout son code,
  11. La dépile
  12. Termine l'exécution de la méthode principale
  13. Elle la dépile.
  14. Le programme s'arrête.

La pile d'exécution Java (The stack) - 4

- **Questions,**
  1. Dessiner l'évolution de la pile d'exécution de l'exemple 1
  2. Même question pour l'exemple 2.

La propagation d'exceptions - 1

- **Soit l'exemple suivant**

```
import java.io.File;
import java.io.IOException;

public class CheckedException {
        private File fichier ;

        public CheckedException (String url) {
                fichier = new File(url);
        }
        public void créerFichier () {
                fichier.createNewFile();
        }
        public static void main (String [] args) {
                CheckedException cke = new CheckedException ("T:/test.txt");
                cke.créerFichier();
                System.out.println("Fin du programme");
        }
}
```

  - Le programme ne compilera pas.
    - La méthode createNewFile() propage une exception vérifiée (Checked) qu'il faut traiter.

La propagation d'exceptions - 2

- **Solution 1**

```
import java.io.File;
import java.io.IOException;
public class CheckedException {
        private File fichier ;

        public CheckedException (String url) {
                fichier = new File(url);
        }
        public void créerFichier () {
                try {
                        fichier.createNewFile();
                } catch (IOException e) {
                        e.printStackTrace();
                }
        }
        public static void main (String [] args) {
                CheckedException cke = new CheckedException ("T:/test.txt");
                cke.créerFichier();
                System.out.println("Fin du programme");
        }
}
```

La propagation d'exceptions - 3

- **Solution 2**

```
import java.io.File;
import java.io.IOException;
public class CheckedException {
        private File fichier ;

        public CheckedException (String url) {
                fichier = new File(url);
        }
        public void créerFichier () throws IOException {
                fichier.createNewFile();
        }
        public static void main (String [] args) {
                CheckedException cke = new CheckedException ("T:/test.txt");
                try {
                        cke.créerFichier();
                } catch (IOException e) {
                        e.printStackTrace();
                }
                System.out.println("Fin du programme");
        }
}
```

La propagation d'exceptions - 4

- **Mauvaise solution**

```
import java.io.File;
import java.io.IOException;

public class CheckedException {
        private File fichier ;

        public CheckedException (String url) {
                fichier = new File(url);
        }
        public void créerFichier () throws IOException {
                fichier.createNewFile();
        }
        public static void main (String [] args) throws IOException {
                CheckedException cke = new CheckedException ("T:/test.txt");
                cke.créerFichier();
                System.out.println("Fin du programme");
        }
}
```

La propagation d'exceptions - 5

- Par traiter une exception, on entend soit la capturer tout de suite, soit la propager.
  - Par propager, on entend laisser le sous-programme appelant la traiter.
  - Il est possible lorsque l'on invoque un sous-programme levant une exception, de la propager.
  - Comme le montre l'exemple de la solution 2 et la mauvaise solution.
- Pour propager une exception dans la pile d'exécution, on utilise le mot-clé **throws** dans la signature de la méthode après les paramètres.
  - Dans la mauvaise solution et la solution 2, on se contente de transmettre (propager) l'exception au sous-programme appelant.
  - Ce mode de fonctionnement fait qu'une exception non capturée va se propager dans la pile d'appels des sous-programmes jusqu'à ce qu'elle soit capturée.
  - Et si elle n'est jamais capturée, c'est la JVM, qui va le faire. Mais, le programme va s'arrêter.

Le bloc finally {…} - 1

- Maintenant, on rajoute un nouveau bloc au mécanisme de gestion des exceptions. Le bloc **finally {…}**

- Grâce à la clause finally, un morceau de code est exécuté quoi qu'il arrive.

- Cela est surtout utilisé lorsque vous devez vous assurer d'avoir fermé
  - un fichier,
  - votre connexion à une base de données
  - un socket (une connexion réseau).

Le bloc finally {…} - 2

- **Exemples**

```
public class TestExceptions {
        public static void divisionParZero () {
                int x = 12;
                int y = 0;
                try {
                        int z = x/y;
                        System.out.println(z);
                        System.out.println ("Ligne dans le try");
                } catch (ArithmeticException e) {
                        System.err.println ("Exception : div par 0");
                } finally {
                        System.out.println ("Ligne du finally");
                }

        }
        public static void main (String [] args) {
                divisionParZero();
                System.out.println ("Ligne du main");
        }
}
```

Le bloc finally {...} - 3

- **Exemples**

```
public class TestExceptions {
        public static void divisionParZero () {
                int x = 12;
                int y = 3;
                try {
                        int z = x/y;
                        System.out.println(z);
                        System.out.println ("Ligne dans le try");
                } finally {
                        System.out.println ("Ligne du finally");
                }

        }
        public static void main (String [] args) {
                divisionParZero();
                System.out.println ("Ligne du main");
        }
}
```

Règles d'utilisation des Exceptions v1.1

- Le bloc **finally** dans un enchainement **try-catch** est optionnel.
- Le bloc **finally** sera toujours exécuté même s'il est précédé par un **return**, sauf force majeure (*arrêt de la JVM par accident ou sur appel de System.exit()*).
- On peut trouver un **try** sans catch à condition que ce bloc soit suivi par un bloc **finally**
- On ne trouve ni **catch** ni **finally** sans **try**.
- Le mot-clé **throws**, sert à propager une exception dans la pile d'exécution.
- Une méthode peut propager plusieurs exceptions

```
public void maMéthode () throws Exception1, Exception2, ..., ExceptionN { ... }
```

Déclencher manuellement une exception

- En java on peut déclencher manuellement une exception existante.

Soit l'exemple suivant :

```
public class ExceptionManuelle {
        int [] notes = {11, 21, 13, -15};
        public void vérifierNotes () {
            for (int note : notes) {
                if((note > 20) && (note < 0)) {
                  throw new ArithmeticException ("Mauvaise note, une note
        doit être comprise entre 0 et 20");
                }
            }
        }
        public static void main (String [] args) {
                ExceptionManuelle em = new ExceptionManuelle();
                em.vérifierNotes ();
        }
}
```

La classe Throwable - 1

- La classe **Throwable** est la classe mère de toutes les exceptions et erreurs :
  - seules des instances de **Throwable** ou de ses classes dérivées peuvent être levée par l'instruction **throw** ou être argument d'un **catch**.
- La classe **Throwable** a
  - un constructeur par défaut et
  - un constructeur qui a en argument une chaîne de caractères : le "message" de l'exception.

`Throwable()`	Le constructeur par défaut.
`Throwable(String message)`	Le constructeur avec un paramètre, le message de l'exception.

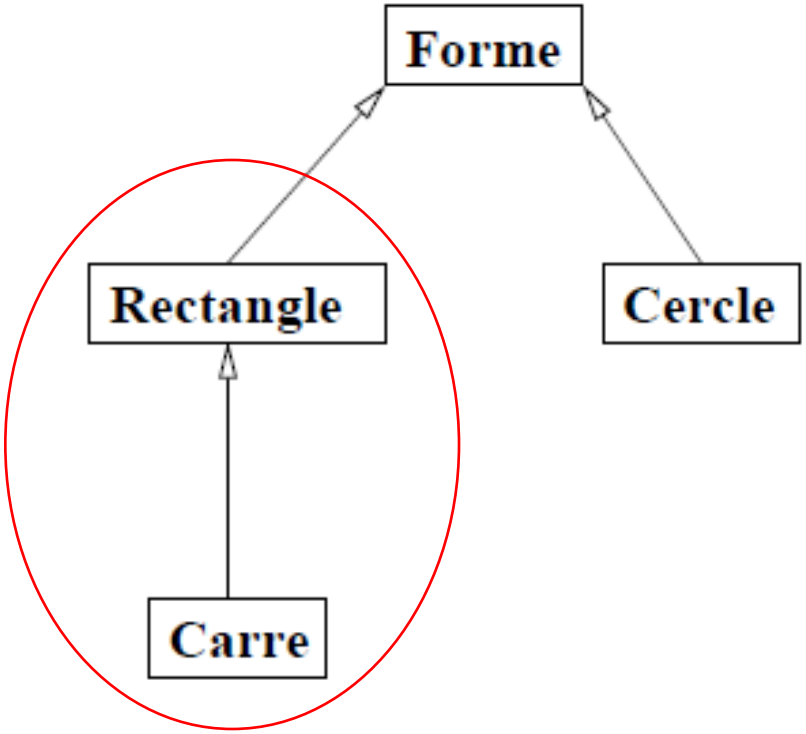
- Les méthodes de cette classes sont résumés dans le tableau suivant

La classe Throwable - 2

`StackTraceElement[]getStackTrace()`	Retourne un tableau représentant l'état de la pile, au moment où a été levée l'exception. Un élément de stack, contient les informations suivantes : - le nom de la méthode - le numéro de ligne où a été levée l'exception - le nom de la classe - le nom du fichier
`String getLocalizedMessage()`	Si la méthode n'est pas redéfinie, identique à *getMessage().*
`String getMessage()`	Retourne le message
`void printStackTrace()`	Imprime le contenu de la pile dans le fichier d'erreur *err*
`void printStackTrace (PrintStream s)`	Imprime le contenu de la pile dans *s*
`void printStackTrace(PrintWriter s)`	Imprime le contenu de la pile dans *s*
`String toString()`	

Les exceptions personnalisées - 1

- Nous allons reprendre notre exemple de formes géométriques



Les exceptions personnalisées - 2

- Nous n'allons pas utiliser les exceptions prédéfinies en java.
  - Nous allons créer les notre.

- Les exceptions personnalisées descendent de la classe **Exception**. Les **RunTimeException** sont inclues, mais pas de la classe **Error**.
  - Il est préférable (par convention) d'inclure le mot « *Exception* » dans le nom de la nouvelle classe et de préférence à la fin.

```
public class ExceptionPersonnalisée extends Exception {

}
```

- Les exceptions sont des sous-classes de **Throwable**
  - Elles héritent de toutes ses méthodes *publiques* et *protected*.
  - Elles peuvent les utiliser et les *redéfinir*.

Les exceptions personnalisées - 3

- Si on héritent de la classe **RuntimeException** ou l'une de ses dérivées.
  - On aura une exception *non vérifiée*.
- Si on héritent de l'une du reste de l'hiérarchie d'exceptions.
  - On aura, bien évidement, une exception *vérifiée*.

```
public class ExceptionPersonnalisée extends IndexOutOfBoundsException {
        // Exception non vérifiée
}
```

```
public class ExceptionPersonnalisée extends RuntimeException {
        // Exception non vérifiée
}
```

```
public class ExceptionPersonnalisée extends IOException {
        // Exception vérifiée
}
```

Les exceptions personnalisées - 4

- Créons une classe qui hérite directement **d'Exception** et qui se déclenche à chaque fois qu'on donne une valeur nulle ou négative à la longueur ou la largeur d'un rectangle :
  - Nom de la classe **ExceptionMesure**
  - Constructeur prenant en argument un entier représentant la *longueur* ou la *largeur*.
    - Ce constructeur fait appel au constructeur avec arguments de sa classe mère.
  - Redéfinition de la méthode *toString()*.

- Dans la classe **Rectangle**
  - Tester les valeurs de la longueur et de la largeur données en argument.
  - S'ils sont *<= 0*, lever l'exception **MesureException**.

Les exceptions personnalisées - 5

- Quand on lève une exception vérifié, il est obligatoire de la propager. Le compilateur exigera qu'on l'intercepte ou la propage.
- Mais au début il faut la lever, pour pouvoir par la suite la traiter.

- Voici, le code de la classe *MesureException* :

```
public class MesureException extends Exception {

        public MesureException(int val) {
                super("Mesure erroné : "+val+" n'est pas une mesure valide !");
        }

        public String toString() {
                return this.getMessage()+"\n"+super.toString();
        }

}
```

Les exceptions personnalisées - 6

- Voici, le code de la classe *Rectangle* :

```
public class Rectangle {
        private int longueur;
        private int largeur;

        public Rectangle(int longueur, int largeur) throws MesureException {
                if (longueur <= 0) {
                        throw new MesureException(longueur);
                }

                if (largeur <= 0) {
                        throw new MesureException(largeur);
                }

                this.longueur = longueur;
                this.largeur = largeur;
        }

        public int surface() {
                return this.longueur * this.largeur;
        }

}
```

Les exceptions personnalisées - 7

- Voici, le code de la classe *TestRectangle* :

```
public class TestRectangle {
        public static void main(String[] args) {
                try {
                        Rectangle r = new Rectangle(0, -2);
                        System.out.println(r.surface());

                } catch (MesureException me) {
                        me.printStackTrace();
                }
        }
}
```

Règles d'utilisation des Exceptions v2.0 - 1

- **Exception personnalisées**
  - Les exceptions sont des sous-classes de **Throwable**, elles héritent de ses méthodes.
  - Une exception personnalisée héritant de **RuntimeException** ne sera pas vérifiée par le *javac*. Tandis qu'en héritant du reste de l'hiérarchie, l'exception sera vérifiée.
  - Pour lancer une exception on utilise le mot-clé **throw** suivi d'une instance de l'exception.
  - Les exceptions vérifiées lancé dans le corps d'une méthode doivent être déclarées dans son entête en utilisant le mot-clé **throws** suivi du type de l'exception.

Règles d'utilisation des Exceptions v2.0 - 2

- **throw vs throws**
  - **throw** sert à lancer une exception dans les blocs de code.
  - **throws** sert à propager une exception. On l'utilise seulement dans les entêtes des méthodes.

- **Exceptions et redéfinition**
  - Rappelons
    - Pour redéfinir une méthode, il faut qu'elle ait le même nom et exactement les même paramètres que la méthode de la classe mère.
    - La type de retour de la méthode redéfinie peut être le même que la méthode originale ou un sous-type (Retour covariant).
    - La visibilité de la méthode redéfinie ne doit pas être plus restrictive que celle de la classe mère.

Règles d'utilisation des Exceptions v2.0 - 3

- On rajoute
  - Elle ne doit pas déclarer une nouvelle exception vérifiée.
  - Ni une exception vérifié plus générique que la méthode originale.
  - Elle peut déclarer des exception descendantes de **RuntimeException**.
  - Elle peut aussi ne pas ré-déclarer l'exception déclarée par la méthode originale.

Exceptions et surcharge

- Rappelons
  - En Java il possible de déclarer deux ou plusieurs méthodes avec le même nom à condition que :
    - Les types et/ou le nombre de paramètres soient différents.
    - La visibilité et le type de retour peuvent êtres différents.
- On rajoute
  - Les méthodes surchargées peuvent déclarer des exceptions (vérifiées ou non vérifiées) différentes.

UNIVERSITÉ PARIS 8
VINCENNES-SAINT-DENIS

Fin chapitre 4
(Les exceptions)