ITI 1521. Introduction à l'informatique II*

Marcel Turcotte
École d'ingénierie et de technologie de l'information

Version du 14 février 2011

Résumé

– Traitement d'erreurs en Java :
  – Déclaration, gestion, création des exceptions
  – ≪Checked≫ et ≪unchecked≫
  – Créer ses propres types d'exceptions

∗. Ces notes de cours ont été conçues afin d'être visualiser sur un écran d'ordinateur.

Traitement des erreurs d'exécution en Java

On distingue deux types d'erreurs : les erreurs de compilation et les erreurs d'exécution.

Les erreurs de syntaxes sont détectées lors de la compilation.

Java étant un langage fortement typé, le compilateur vérifie aussi le type de chaque expression, ce qui permet la détection de certaines erreurs le plus tôt possible, avant l'exécution du programme. La vérification des types permet de s'assurer que les opérations sur une valeur sont valides pour le type de la valeur.

Certains types d'erreurs ne peuvent être détectés au moment de la compilation, par exemple, retirer un élément d'une pile vide !

En Java, les erreurs d'exécution sont modélisées à l'aide d'exceptions (des objets).

Sources des erreurs

– Erreurs de logique ;
– Causes extérieures : quantité de mémoire insuffisante, erreur en lecture, etc.

La détection et le traitement des situations d'erreurs contribuent à rendre les programmes plus robustes.

Idéalement, les mécanismes mis en place devraient indiquer la source de l'erreur de façons précises.

Par exemple, un appel à la méthode **pop()** de la classe **ArrayStack** peut causer l'erreur **IndexOutOfBoundsException : -1**. Doit-on conclure que l'erreur se trouve forcément dans la méthode **pop()** ?

Non, la méthode appelante aurait dû s'assurer que la pile n'était pas vide.

Préconditions

Les **préconditions** sont l'ensemble des conditions auxquelles les **paramètres** et l'**état de l'objet** doivent se conformés afin que la méthode puisse être exécutée avec succès.

Le calcul de la racine carrée nous fournit un bon exemple, afin de calculer la racine carrée d'un nombre, il faut d'abord s'assurer que le nombre est non négatif.

Dans le cas de la racine carrée, si la précondition n'est pas respectée, selon le langage de programmation, ou la librairie de méthodes utilisée, il se peut que le programme boucle à l'infini, ou encore, termine en catastrophe.

Une bonne habitude de programmation consiste à toujours vérifier les préconditions au début de chaque méthode.

Pour la suite du semestre, nous tenterons de toujours identifier les préconditions et de les traiter de façons appropriées.

Au sujet de la valeur de top. . .

Traitements des erreurs

Que faire ?

Terminer en catastrophe $(1/0)$, quitter le programme, . . . certainement pas.

Imprimer un message ? Très bien, mais c'est insuffisant.

Retourner à l'appelant une valeur spéciale indiquant le type d'erreur.

Prenez l'exemple d'une fouille binaire dans un tableau, la méthode retourne un entier désignant la position de la valeur recherchée dans le tableau, et -1 si elle ne s'y trouve pas.

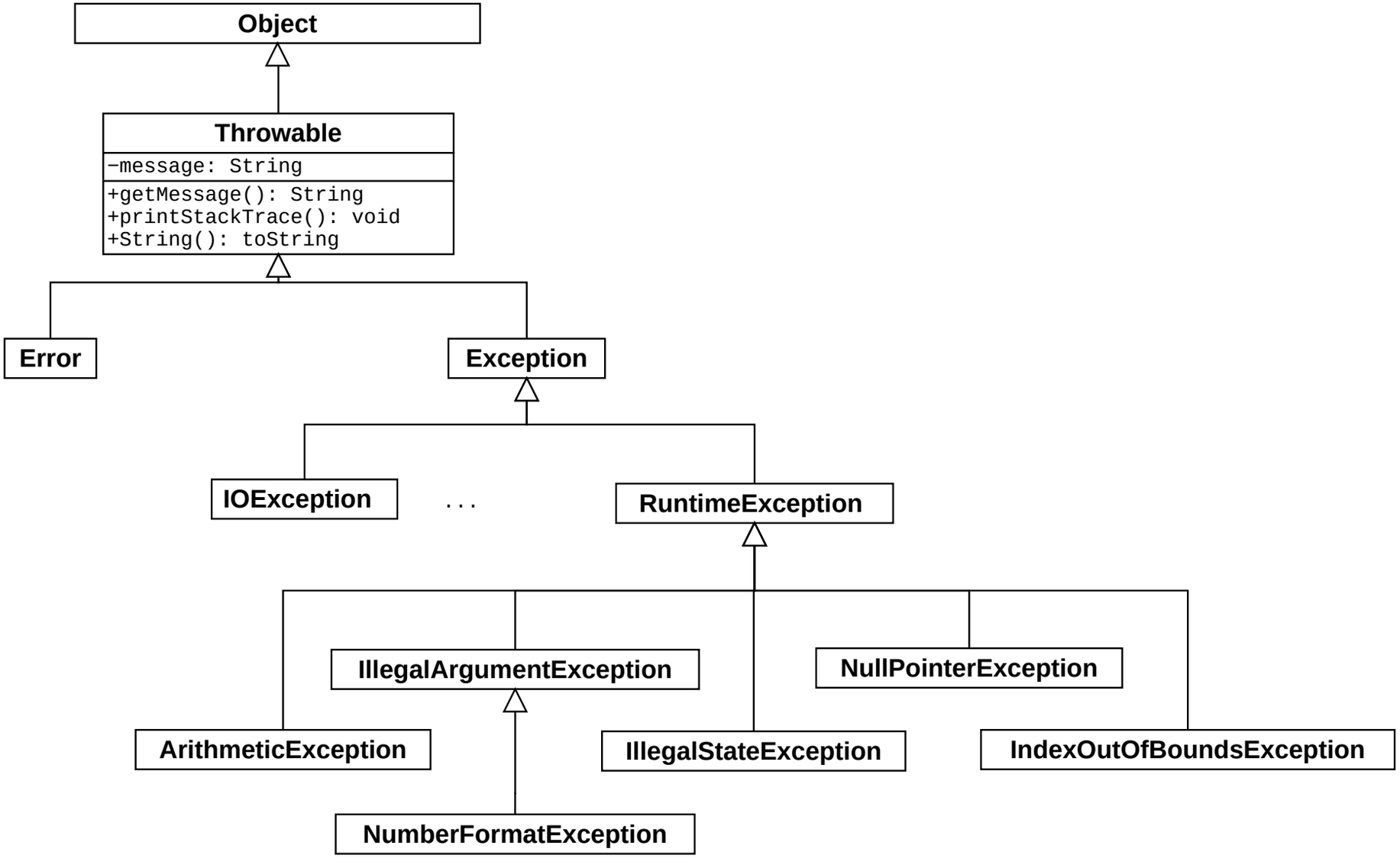
Cette solution n'est ni suffisante, ni assez générale :

- **Certaines méthodes n'ont pas de valeur de retour, ou encore, aucune des valeurs de retour n'est disponible** ;
- **L'appelant n'est pas forcé d'effectuer les actions nécessaires afin de traiter l'erreur**.

Traitements des erreurs

Tous les langages de programmation modernes offrent des mécanismes pour traiter les erreurs, et ces mécanismes varient d'un langage à l'autre ; en Java, ce mécanisme s'appelle le traitement des exceptions.

Exception est une classe !



Exception est une classe !

Les exceptions sont des objets !

Une situation d'erreur est modélisée à l'aide d'un objet de la classe **Throwable**, ou l'une de ses sous-classes. L'objet encapsule un message d'erreur.

Entre autres, la classe **Throwable** déclare les méthodes **String getMessage()** et **void printStackTrace()**.

La déclaration d'une référence de type **Exception** n'a rien d'exceptionnel.

```
Exception e;
```

De même, la création d'un objet de la classe **Exception** n'a rien d'exceptionnel.

```
e = new Exception( "Houston, we have a problem" );
```

Signaler une erreur

Considérons l'exemple d'une pile et de sa méthode **pop()**.

Le retrait d'un élément lorsque la pile est vide constitue une situation d'erreur. Jusqu'à maintenant nous avons indiqué cette condition à l'aide d'un commentaire.

```
// pre-condition: la pile n'est pas vide
```

En Java, nous pouvons utiliser un énoncé ≪**throw**≫ afin de modifier le flot de contrôle normal.

```
if ( top == null ) {
   throw new IllegalStateException( "Empty stack" );
}
```

L'énoncé **throw** :

throw expression

où **expression** est une référence désignant un objet de la classe **Throwable**, ou l'une de ses sous-classes.

```
IllegalStateException e;

e = new IllegalStateException( "Empty stack" );

if ( top == null ) {
   throw e;
}
```

C'est l'énoncé **throw** qui signale la situation d'erreur ; pas la déclaration d'une référence, pas la création d'un objet.

Transfère du contrôle

Lorsqu'une exception est signalée,

- L'énoncé ou l'expression **termine abruptement** ;
- Si elle n'est pas récupérée, la pile des appels de méthodes sera déroulée complètement, c'est-à-dire que chaque appel de méthode se trouvant sur la pile d'exécution terminera abruptement, et l'exécution du programme se terminera avec l'impression de la pile des appels au moment de l'erreur (≪stack trace≫) ;
- Aucun des énoncés et aucune des parties de l'expression se trouvant après l'expression ayant causée l'erreur ne seront exécutés ;
- Suite à une exception, les prochains énoncés exécutés sont ceux se trouvant dans un bloc **catch** ou **finally**.

```
public class Test {
    public static void main( String[] args ) {

        System.out.println( "-1-" );

        throw new RuntimeException( "an Exception" );

        System.out.println( "-2-" );

    }
}
```

Ça ne compile pas. Pourquoi ? Comment contourner ce problème ?

```
javac Test.java
Test.java:8: unreachable statement
        System.out.println( "-2-" );
        ^
1 error
```

```
class Test {
    public static void main( String[] args ) {
        System.out.println( "-1-" );
        if ( true ) {
            throw new RuntimeException( "an Exception" );
        }
        System.out.println( "-2-" );
    }
}
```

Lorsqu'une exception est lancée, la méthode termine abruptement.

```
> java Test
-1-
Exception in thread "main" java.lang.RuntimeException:
an Exception
        at Test.main(Test.java:5)
```

La chaîne ≪-2-≫ ne sera donc pas imprimée.

De même, les sous parties d'une expression ne sont pas exécutées.

```
public class Test extends Object {
    public static boolean error() {
        if ( true ) {
            throw new RuntimeException( "Ouille, ouille, ouille" );
        }
        return true;
    }
    public static boolean display() {
        System.out.println( "-2-" );
        return true;
    }
    public static void main( String[] args ) {
        System.out.println( "-1-" );
        if ( error() || display() ) {
            System.out.println( "-3-" );
        }
        System.out.println( "-4-" );
    }
}
```

```
> java Test
-1-
Exception in thread "main" java.lang.RuntimeException: an Exception
        at Test.error(Test.java:5)
        at Test.main(Test.java:16)
```

Dérouler la pile des appels.

```
public class Test extends Object {
    public static void c() {
        System.out.println( "c: -1-" );
        if ( true ) {
            throw new RuntimeException( "dessus de la pile des appels" );
        }
        System.out.println( "c: -2-" );
    }
    public static void b() {
        System.out.println( "b: -1-" );
        c();
        System.out.println( "b: -2-" );
    }
    public static void a() {
        System.out.println( "a: -1-" );
        b();
        System.out.println( "a: -2-" );
    }
    public static void main(String[] args) {
        System.out.println( "m: -1-" );
        a();
        System.out.println( "m: -2-" );
    }
}
```

```
> java Test
m: -1-
a: -1-
b: -1-
c: -1-
Exception in thread "main" java.lang.RuntimeException:
top of the stack
        at Test.c(Test.java:6)
        at Test.b(Test.java:11)
        at Test.a(Test.java:16)
        at Test.main(Test.java:21)
```

Traitement des exceptions

Une méthode faisant appel à une autre pouvant lancer une exception d'une classe autre qu'**Error**, **RuntimeException**, ou une sous-classe de celles-ci, doit :

– Traiter l'exception (**catch**) ;
– Laisser passer l'exception et doit alors la déclarer (**throws**).

Traiter les exceptions

La construction syntaxique **try/catch** est utilisée afin de récupérer le contrôle lors de situations d'erreurs.

```
try {
    ...
} catch ( exception_type1 id1 ) {
    statements;
} catch ( exception_type2 id2 ) {
    statements;
...
} finally {
    statements;
}
```

Si aucune exception n'est lancée, seuls les énoncés du bloc **try** et du bloc **finally** seront exécutés.

```
public class Grill {
    private Burner burner = new Burner();
    public void cooking() {
        try {
            burner.on();
            addSteak();
            addSaltAndPepper();
            boolean done = false;
            while (! done) {
                done = checkSteak();
            }
        } catch ( OutOfGazException e1 ) {
            callRetailer();
        } catch ( FireException e2 ) {
            extinguishFire();
        } finally {
            burner.off();
        }
    }
}
```

```
int DEFAULT_VALUE = 0;
int value;

try {

    value = Integer.parseInt( "100" );

} catch ( NumberFormatException e ) {

    value = DEFAULT_VALUE;

}

System.out.println( "value = " + value );
```

$\Rightarrow$ affiche 100.

public static int parseInt(String s) throws NumberFormatException

Parses the string argument as a signed decimal integer. The characters in the string must all be decimal digits, except that the first character may be an ASCII minus sign '-' ('\u002D') to indicate a negative value. The resulting integer value is returned, exactly as if the argument and the radix 10 were given as arguments to the parseInt(java.lang.String, int) method.

Parameters:
s - a String containing the int representation to be parsed

Returns:
the integer value represented by the argument in decimal.

Throws:
NumberFormatException - if the string does not contain a parsable integer.

```
int DEFAULT_VALUE = 0;
int value;

try {

    value = Integer.parseInt( "humm" );

} catch ( NumberFormatException e ) {

    value = DEFAULT_VALUE;

}

System.out.println( "value = " + value );
```

$\Rightarrow$ affiche 0.

Lorsqu'une exception est lancée, l'exécution des énoncés du bloc **try** se termine (abruptement) et se poursuit avec les énoncés du premier bloc **catch** dont le paramètre est du même type que celui de l'objet modélisant la situation d'erreur, ou d'un type plus général, suivi de l'exécution des énoncés du bloc **finally**, si présent.

Aucun autre bloc ne sera exécuté.

Si aucun bloc **catch** n'est adéquat alors l'exception percole.

Les énoncés du bloc **finally** sont toujours exécutés : avec ou sans erreur.

Les blocs finally sont utilisés afin fermer des fichiers ouverts, par exemple, ou de façon générale, afin de traiter les ≪post≫-conditions.

```
public class Test {
    public static void c() {
        System.out.println( "c() :: about to throw exception" );
        throw new RuntimeException( "from c()" );
    }
    public static void b() {
        System.out.println( "b() :: pre-" );
        c();
        System.out.println( "b() :: post-" );
    }
    public static void a() {
        System.out.println( "a() :: pre-" );
        try {
            b();
        } catch ( RuntimeException e ) {
            System.out.println( "a() :: caught exception" );
        }
        System.out.println( "a() :: calling b, no try block" );
        b();
        System.out.println( "a() :: post-" );
    }
    public static void main( String[] args ) {
        System.out.println( "main( ... ) :: pre-" );
        a();
        System.out.println( "main( ... ) :: post-" );
    }
}
```

```
main( ... ) :: pre-
a() :: pre-
b() :: pre-
c() :: about to throw exception
a() :: caught exception
a() :: calling b, no try block
b() :: pre-
c() :: about to throw exception
Exception in thread "main" java.lang.RuntimeException: from c
        at Test.c(Test.java:4)
        at Test.b(Test.java:8)
        at Test.a(Test.java:19)
        at Test.main(Test.java:24)
```

On doit créer des blocs **catch** qui sont le plus spécifiques possible. Les énoncés suivants attraperaient tout type d'exception.

```
try {

    value = Integer.parseInt( "humm" );

} catch ( Exception e ) {

    value = DEFAULT_VALUE;

}
```

Qu'en pensez-vous ?

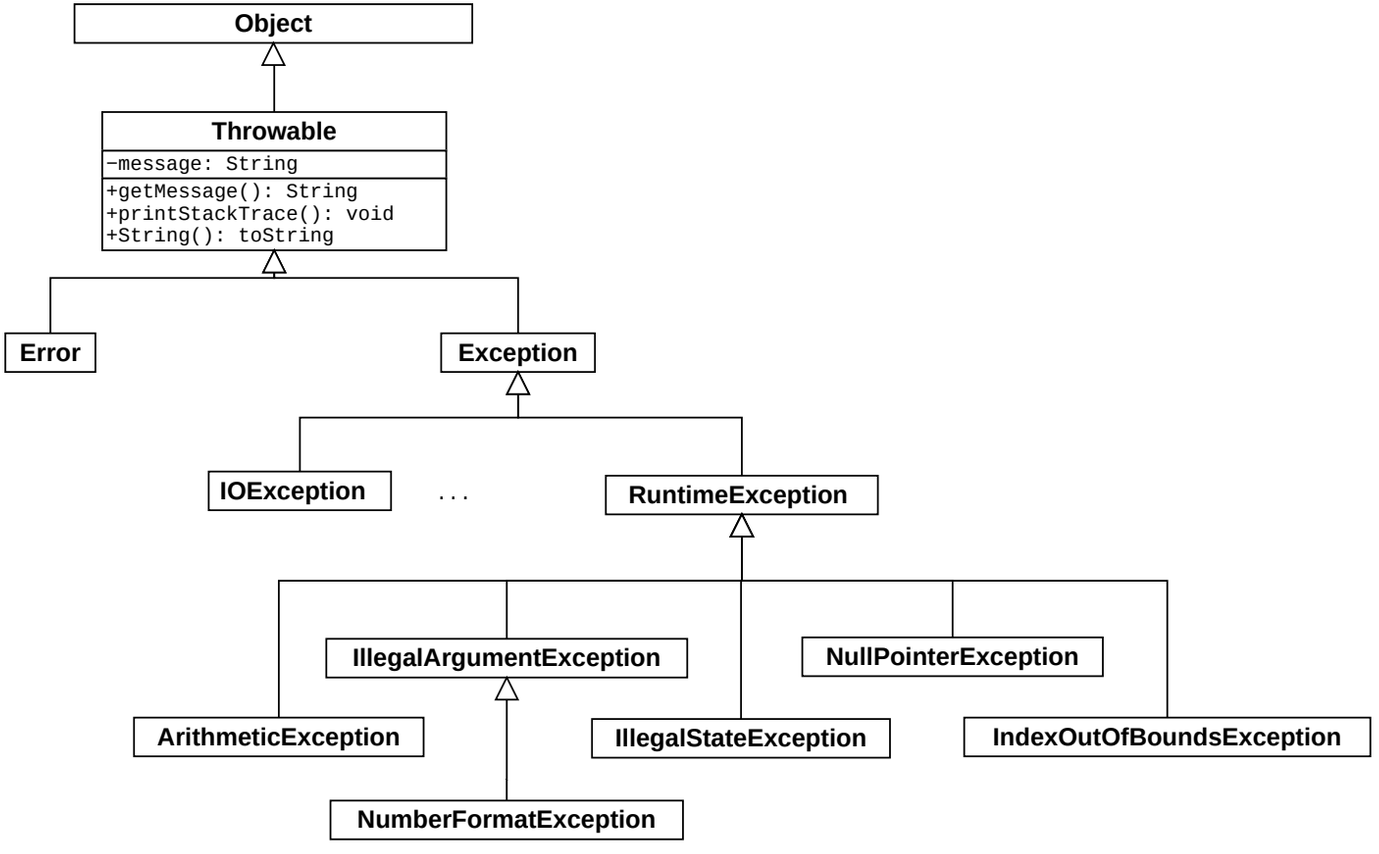
Le bloc **catch** est conçu afin de traiter un type bien particulier d'exceptions, les erreurs causées lorsque la chaîne ne représente pas un entier. Le bloc ci-haut intercepterait aussi une exception de type **ClassNotFoundException** !

```
public class Test {
    public static void main( String[] args ) {
        boolean valid = false;
        int value = -1;
        for ( int i=0; i<args.length && ! valid; i++ ) {
            try {
                value = Integer.parseInt( args[i] );
                valid = true;
            } catch ( NumberFormatException e ) {
                System.out.println( "not a valid number: " + args[ i ]
            }
        }
        if ( valid ) {
            System.out.println( "value = " + value );
        } else {
            System.out.println( "no valid number was found" );
        }
    }
}
```

```
> java Test a 1.1 "" 2 3
not a valid number: a
not a valid number: 1.1
not a valid number:
value = 2
```

```
int DEFAULT_VALUE = 0;
int value;

try {

    value = Integer.parseInt ( "a" );

} catch ( NumberFormatException e ) {

    System.out.println( "warning: " + e.getMessage() );

    value = DEFAULT_VALUE;

}
```

$\Rightarrow$ **e** est une référence désignant l'objet créé dans le contexte d'un énoncé **throw**, comme pour tout autre objet, la notation pointée est utilisée afin d'accéder aux méthodes de l'objet.



Une exception est dite ≪**checked**≫ ou ≪**unchecked**≫. Par défaut, toutes les exceptions sont ≪**unchecked**≫, ainsi toutes les sous-classes de la classe ≪**Throwable**≫ sont des exceptions ≪**unchecked**≫. Par contre, toutes les sous-classes de la classe **Exception** sont ≪checked≫, sauf celles qui sont des sous-classes de la classe **RuntimeException**.

Deux genres d'exceptions : ≪checked≫ ou ≪unchecked≫

Une méthode pouvant causer une exception dite ≪**checked exception**≫ doit déclarer l'exception ou la traiter.

La compilation de la classe test suivante causera une **erreur de compilation**.

```
class Test {
    public static void main( String[] args ) {
        System.out.println( "-1-" );
        if ( true ) {
            throw new Exception( "an Exception" );
        }
        System.out.println( "-2-" );
    }
}
> javac Test.java
Test.java:5: unreported exception java.lang.Exception;
must be caught or declared to be thrown
            throw new Exception("an Exception");
            ^
```

Deux genres d'exceptions : ≪checked≫ ou ≪unchecked≫

Les exceptions dites ≪**unchecked exception**≫ regroupent les exceptions telles que **NullPointerException** et **IndexOutOfBoundsException**.

Ces erreurs pourraient survenir si fréquemment que Java n'impose pas le traitement systématique de ces erreurs.

En effet, toute méthode utilisant des références est susceptible de générer des exceptions de type **NullPointerException**.

De même, toute méthode utilisant des tableaux est susceptible de générer des exceptions de type **IndexOutOfBoundsException**.

À moins d'erreurs de logique, ces méthodes ne devraient pas générer d'exceptions, ainsi la déclaration, à l'aide de l'énoncé `throws`, est **optionnelle**.

Deux genres d'exceptions : ≪**checked**≫ ou ≪**unchecked**≫

Les exceptions dites ≪**checked**≫ sont utilisées afin de forcer l'appelant à définir une stratégie pour traiter une situation d'erreur particulière : **IOException**.

Souvent utilisé dans un contexte où une situation d'erreur peut être causée par un événement extérieur.

Utiliser dans les cas où une telle stratégie est possible.

Traitement des exceptions

Une méthode faisant appel à une méthode pouvant lancer une exception dite «**checked**» doit :
– Traiter l'exception (catch) ;
– **Laisser passer l'exception et doit alors la déclarer (throws)**.

```
public static void main( String[] args )
    throws IOException, FileNotFoundException, IllegalAccessException {

    ...
}
```

Traitement des exceptions

Une méthode peut déclarer des exceptions ≪**checked**≫ et ≪**unchecked**≫ ; pour les exceptions ≪**unchecked**≫ la déclaration est **optionnelle**.

```
public static void main( String[] args )
    throws IOException, FileNotFoundException, NullPointerException {

    ...
}
```

Effets transitifs (1/5)

```
import java.io.*;

public class Keyboard {
    public static int getInt() {
        byte[] buffer = new byte[ 256 ];

        System.in.read( buffer ); // throws IOException

        String s = new String( buffer );
        int num = Integer.parseInt( s.trim() );
        return num;
    }
    public static void main( String[] args ) {
        System.out.print( "Please enter a number: " );
        int n = Keyboard.getInt();
        System.out.println( "You've entered the number: " + n );
    }
}
```

Effets transitifs (1/5)

```
> javac Keybord.java
Keyboard.java:9: unreported exception java.io.IOException;
must be caught or declared to be thrown
     System.in.read(buffer);
                    ^
1 error
```

Effets transitifs (2/5)

```
import java.io.*;

public class Keyboard {
    public static int getInt() throws IOException {
        byte[] buffer = new byte[ 256 ];

        System.in.read( buffer ); // throws IOException

        String s = new String( buffer );
        int num = Integer.parseInt( s.trim() );
        return num;
    }
    public static void main( String[] args ) {
        System.out.print( "Please enter a number: " );

        int n = Keyboard.getInt(); // throws IOException

        System.out.println( "You've entered the number: " + n );
    }
}
```

Effets transitifs (2/5)

```
> javac Keyboard.java
Keyboard.java:22: unreported java.io.IOException;
must be caught or declared to be thrown
     int n = Keyboard.getInt();
                              ^
1 error
```

Effets transitifs (3/5)

```
import java.io.*;

public class Keyboard {
    public static int getInt() throws IOException {
        byte[] buffer = new byte[ 256 ];

        System.in.read( buffer ); // throws IOException

        String s = new String( buffer );
        int num = Integer.parseInt( s.trim() );
        return num;
    }
    public static void main( String[] args )
    throws IOException {

        System.out.print( "Please enter a number: " );

        int n = Keyboard.getInt(); // throws IOException

        System.out.println( "You've entered the number: " + n );
```

```
        }
    }
```

Effets transitifs (3/5)

```
> java Keyboard
Please enter a number: oups
Exception in thread "main" java.lang.NumberFormatException
For input string: "oups"
     at java.lang.NumberFormatException.
        forInputString(NumberFormatException.java:48)
     at java.lang.Integer.parseInt(Integer.java:468)
     at java.lang.Integer.parseInt(Integer.java:518)
     at Keyboard.getInt(Keyboard.java:13)
     at Keyboard.main(Keyboard.java:23)
```

Effets transitifs (4/5)

```
import java.io.*;

public class Keyboard {
    public static int getInt() throws IOException {
        byte[] buffer = new byte[ 256 ];
        System.in.read( buffer );
        String s = new String( buffer );
        int num = Integer.parseInt( s.trim() );
        return num;
    }
```

Effets transitifs (4/5)

```
    public static void main(String[] args)
    throws IOException {
        int n;
        boolean done = false;
        while ( ! done) {
            System.out.print( "Please enter a number: " );
            try {
                n = Keyboard.getInt();
                System.out.println( "The number is " + n );
                done = true;
            } catch ( NumberFormatException e ) {
                System.out.println( "Not a number!" );
            }
        }
    }
}
```

Effets transitifs (4/5)

```
> java Keyboard
Please enter a number: oups
Not a number!
Please enter a number: a1
Not a number!
Please enter a number:  1
The number is 1
```

Effets transitifs (5/5)

```
import java.io.*;

public class Keyboard {

   public static int getInt()
       throws IOException, NumberFormatException {
       byte[] buffer = new byte[ 256 ];
       System.in.read( buffer );
       String s = new String( buffer );
       int num = Integer.parseInt( s.trim() );
       return num;
   }
```

Effets transitifs (5/5)

```
    public static void main( String[] args ) {
        int n, i=0;
        boolean done = false;
         while ( ! done ) {
            i++;
            System.out.print( "Please enter a number: " );
            try {
                n = Keyboard.getInt();
                System.out.println( "The number is " + n );
                done = true;
            } catch ( NumberFormatException e ) {
              System.out.println( "Not a number!" );
            } catch ( IOException e ) {
              System.out.println( "Keyboard connected?" );
            } finally {
              System.out.println( "Iteration: " + i );
            }
        }
    }
}
```

Créer de nouveaux types d'exceptions

Les exceptions sont des objets on peut donc créer ses propres types d'exceptions.

Les exceptions dites ≪**checked exceptions**≫ doivent être traitées ou déclarées.

Pour les exceptions dites ≪**unchecked exceptions**≫ (**Error**, **RuntimeException**, une sous-classe de l'une ou l'autre) il n'est pas nécessaire de les déclarer ou de les traiter.

Le compilateur s'assure que toutes les exceptions ≪**checked**≫ sont déclarées ou traitées.

```
public class MyException extends Exception {
}
```

```
public class MyException extends Exception {
    public MyException() {
        super();
    }
    public MyException( String message ) {
        super( message );
    }
}
```

Classe Time

Dans l'exemple qui suit, la méthode **parseTime** attrape les exceptions de type **NumberFormatException** ou **NoSuchElementException** afin de lancer une exception dont le type est plus informatif, **TimeFormatException**.

```
public class TimeFormatException extends IllegalArgumentException {

    public TimeFormatException() {
        super();
    }

    public TimeFormatException( String msg ) {
        super( msg );
    }
}
```

```
public class Time {

    // ...

    public static Time parseTime( String timeString ) {

        StringTokenizer st = new StringTokenizer( timeString, ":", true

        int h, m, s;

        try {
            h = Integer.parseInt( st.nextToken () );
        } catch ( NumberFormatException e1 ) {
            throw new TimeFormatException( "first field is not a number
        } catch ( NoSuchElementException e2 ) {
            throw new TimeFormatException( "first separator not found:
        }

        try {
            st.nextToken();
```

```
} catch ( NoSuchElementException e2 ) {
    throw new TimeFormatException( "first separator not found:
}

try {
    m = Integer.parseInt( st.nextToken() );
} catch ( NumberFormatException e1 ) {
    throw new TimeFormatException( "second field is not a numbe
} catch ( NoSuchElementException e2 ) {
    throw new TimeFormatException( "second separator not found:
}

try {
    st.nextToken();
} catch ( NoSuchElementException e2 ) {
    throw new TimeFormatException( "second separator not found:
}

try {
    s = Integer.parseInt( st.nextToken() );
```

```
        } catch ( NumberFormatException e1 ) {
            throw new TimeFormatException( "third field is not a number
        } catch ( NoSuchElementException e2 ) {
            throw new TimeFormatException( "third field not found: " +
        }

        if ( st.hasMoreTokens() )
            throw new TimeFormatException( "invalid suffix:" + timeStri

        if ( (h<0) || (h>23) || (m<0) || (m>59) || (s<0) || (s>59) )
            throw new TimeFormatException( "values out of range:" + tim

        return new Time( h,m,s );
    }
}
```

Pourquoi créer de nouveaux types d'exceptions ?

Afin de les catégoriser et de les traiter à l'aide de blocs **catch** distincts/spécifiques.

```
try {
 ...
} catch ( Exception e ) {
    // trop général, attrape tout!
}

try {
 ...
} catch ( NumberFormatException e ) {
    var = DEFAULT_VALUE;
}

try {
 ...
} catch ( TimeFormatException e ) {
    var = DEFAULT_VALUE;
}
```

Message d'erreur

```
throw new Exception( "information concise mais précise" );
```

Remarques

Utilisez les types existants si possible.

Ayez des messages clairs.

Lorsqu'on traite (**catch**) une erreur, il faut s'assurer que l'état de l'objet demeure consistant !