

Error handling



Programmers' errors, excerpts

Incorrect implementation.

- Does not meet the specification.

Inappropriate object request.

- E.g., invalid index.

Inconsistent or inappropriate object state.

- E.g. arising through class extension.

TESTS
are part
of the
solution.

Not always programmer error

Errors often arise from the environment

- Incorrect value entered : A typical runtime error is a division by zero error
- Network interruption.
- Unexpected situations
 - Logic errors are caused due to flawed reasoning. These errors are not necessarily due to a “mistake” developer has made. They may occur because she didn’t consider a certain execution scenario.

But handling these errors is still her responsibility

Contents

- Exceptions
- Assertions
- Nullability

- Defensive programming



Dealing with exceptions

Exceptions, what is it?

Exceptions: declaration and triggering

```
void addElement(SystemElement element) throws TitleAlreadyUsedException {  
    Objects.requireNonNull(element);  
    if (indexedElements.keySet().contains(element.getTitle())) {  
        throw new TitleAlreadyUsedException(element.getTitle());  
    }  
    indexedElements.put(element.getTitle(), element);  
    super.addElement(element);  
}
```

Exception créée et « lancée »



Exception

```
public class TitleAlreadyUsedException extends Exception {  
  
    public TitleAlreadyUsedException(String title) {  
        super(title);  
    }  
}
```

Exceptions: declaration and triggering

```
Point createPoint(int x, int y, String title) throws TitleAlreadyUsedException, NonValidElementException {  
    Point newPoint = new Point(title,x,y);  
    if (checkValidy(newPoint)) {  
        elements.addElement(newPoint);  
        return newPoint;  
    }  
    else  
        throw new NonValidElementException(newPoint);  
}
```

Exception créée et « lancée »



Exception

```
public class NonValidElementException extends Exception {  
  
    SystemElement invalidElement;  
    public NonValidElementException(SystemElement newPoint)  
    {  
        this.invalidElement = newPoint;  
    }  
}
```


Handling exceptions: Let it go/Laisser passer

```
Point createPoint(int x, int y, String title) throws TitleAlreadyUsedException, NonValidElementException {  
    Point newPoint = new Point(title,x,y);  
    if (checkValidy(newPoint)) {  
        elements.addElement(newPoint);  
        return newPoint;  
    }  
    else  
        throw new NonValidElementException(newPoint);  
}
```

Si l'exception est « levée », elle n'est pas « attrapée », et continue donc à l'appel à *createPoint*

Rappel :

```
void addElement(SystemElement element) throws TitleAlreadyUsedException {  
    Objects.requireNonNull(element);  
    if (indexedElements.keySet().contains(element.getTitle())) {  
        throw new TitleAlreadyUsedException();  
    }  
    indexedElements.put(element.getTitle(),element);  
    super.addElement(element);  
}
```

Handling exceptions: catch it !

```
final String fin = "q";
String nom = readString("Creer un autre point ('q' to quit) : ");

while (!fin.equals(nom)) {
    writeln("Quelle valeur pour x ?"); int x = sc.nextInt();
    writeln("Quelle valeur pour y ?"); int y = sc.nextInt(); sc.nextLine();
    writeln("Quel nom pour le point ?"); String title = sc.nextLine();

    try {
        Point point = frame.createPoint(x, y, title);
        ...
    } catch (TitleAlreadyUsedException e) {
        writeln("Le titre a déjà été utilisé :" + title);
    } catch (NonValidElementException e) {
        writeln("Les dimensions du point ne sont pas valides :" + e.invalidElement);
    }
    nom = readString("Creer un autre point ('q' to quit) : ");
} //while
}
```

```
Creer un autre point ('q' to quit) : o
Quelle valeur pour x ? 1
Quelle valeur pour y ? 2
Quel nom pour le point ? P1
voici le point créé : Point (1,2), color: (184,158,192), Ti
.....
Quelle valeur pour x ? 35
Quelle valeur pour y ? 12
Quel nom pour le point ? P3
Les dimensions du point ne sont pas valides :Point (35,
(155,28,48), Title : P3
```

L'exception est
« attrapée »
et traitée. Ici la boucle
continue même en
cas de titre erroné ou
d'erreur de
dimensions.

Not Handling exceptions

Quelle valeur pour x ? a

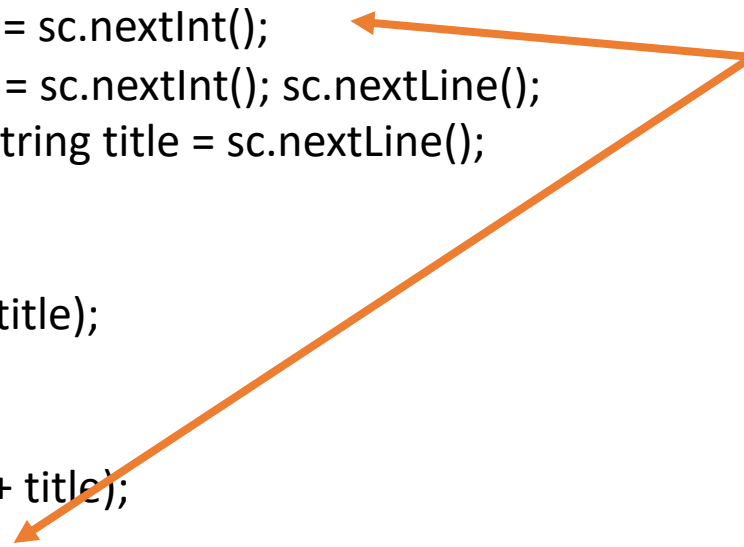
Exception in thread "main" java.util.InputMismatchException

```
final String fin = "q";
String nom = readString("Creer un autre point ('q' to quit) : ");

while (!fin.equals(nom)) {
    writeln("Quelle valeur pour x ?"); int x = sc.nextInt();
    writeln("Quelle valeur pour y ?"); int y = sc.nextInt(); sc.nextLine();
    writeln("Quel nom pour le point ?"); String title = sc.nextLine();

    try {
        Point point = frame.createPoint(x, y, title);
        ...
    } catch (TitleAlreadyUsedException e) {
        writeln("Le titre a déjà été utilisé :" + title);
    } catch (NonValidElementException e) {
        writeln("Les dimensions du point ne sont pas valide :" + e.invalidElement);
    }
    nom = readString("Creer un autre point ('q' to quit) : ");
}
```

L'exception levée en cas d'erreur de saisie n'est pas rattrapée.



Handling all Exceptions

Créer un autre point ('q' to quit) : o

Quelle valeur pour x ? a

Une erreur inattendue s'est produite, peut être une erreur de frappe : java.util.InputMismatchException

Créer un autre point ('q' to quit) :

```
while (!fin.equals(nom)) {  
    try {  
        writeln("Quelle valeur pour x ?"); int x = sc.nextInt();  
        writeln("Quelle valeur pour y ?"); int y = sc.nextInt(); sc.nextLine();  
        writeln("Quel nom pour le point ?"); String title = sc.nextLine();  
        Point point = frame.createPoint(x, y, title);  
        ...  
    } catch (TitleAlreadyUsedException e) {  
        writeln("Le titre a déjà été utilisé :" + e.getMessage());  
    } catch (NonValidElementException e) {  
        writeln("Les dimensions du point ne sont pas valides :" + e.InvalidElement);  
    } catch (Exception e) {  
        writeln("Une erreur inattendue s'est produite, peut être une erreur de frappe  
        :\" + e);  
    }  
    nom = readString("Créer un autre point ('q' to quit) : ");  
}
```

- 1) On augmente la protection.
- 2) On ajoute la possibilité d'attraper une « autre » exception.
- 3) Attention l'ordre a de l'importance ici.

Finally

```
while (!fin.equals(nom)) {
    try {
        writeln("Quelle valeur pour x ?"); int x = sc.nextInt();
        writeln("Quelle valeur pour y ?"); int y = sc.nextInt(); sc.nextLine();
        writeln("Quel nom pour le point ?"); String title = sc.nextLine();
        Point point = frame.createPoint(x, y, title);
        ...
    } catch (TitleAlreadyUsedException e) {
        writeln("Le titre a déjà été utilisé :" + e.getMessage());
    } catch (NonValidElementException e) {
        writeln("Les dimensions du point ne sont pas valides :" +
e.invalidElement);
    } catch (Exception e) {
        writeln("Une erreur inattendue s'est produite, peut être une erreur de
frappe :" + e);
    }
    finally {
        writeln("Etat du repère : " + frame);
    }
    nom = readString("Créer un autre point ('q' to quit) : ");
}
```

voici le point créé :Point (1,2), color: (184,158,192),
Title : P1
Etat du repère : Frame{xAxis=Axis size : 20, Title : X
AXIS, yAxis=Axis size : 20, Title : Y AXIS,
elements=SmartSystemElement ...

Le titre a déjà été utilisé :P1

Etat du repère : Frame{xAxis=Axis size : 20, Title : X
AXIS, yAxis=Axis size : 20, Title : Y AXIS...

Créer un autre point ('q' to quit) :

Pour exécuter une
commande dans tous
les cas, même si une
exception est levée !

Finally

```
while (!fin.equals(nom)) {  
    try {  
        writeln("Quelle valeur pour x ?"); int x = sc.nextInt();  
        writeln("Quelle valeur pour y ?"); int y = sc.nextInt(); sc.nextLine();  
        writeln("Quel nom pour le point ?"); String title = sc.nextLine();  
        Point point = frame.createPoint(x, y, title);  
        ...  
    } catch (TitleAlreadyUsedException e) {  
        writeln("Le titre a déjà été utilisé :" + e.getMessage());  
    } catch (NonValidElementException e) {  
        writeln("Les dimensions du point ne sont pas valides :" + e.invalidElement);  
    } finally {  
        writeln("Etat du repère : " + frame);  
    }  
    nom = readString("Creer un autre point ('q' to quit) : ");  
}
```

Quelle valeur pour x ?

a

Etat du repère : Frame{xAxis=Axis size : 20, Title : X
AXIS, yAxis=Axis size : 20, Title : Y AXIS...

Exception in thread "main"

java.util.InputMismatchException

Pour exécuter une
commande dans tous
les cas, même si une
exception est levée !

On résume, les effets d'une exception

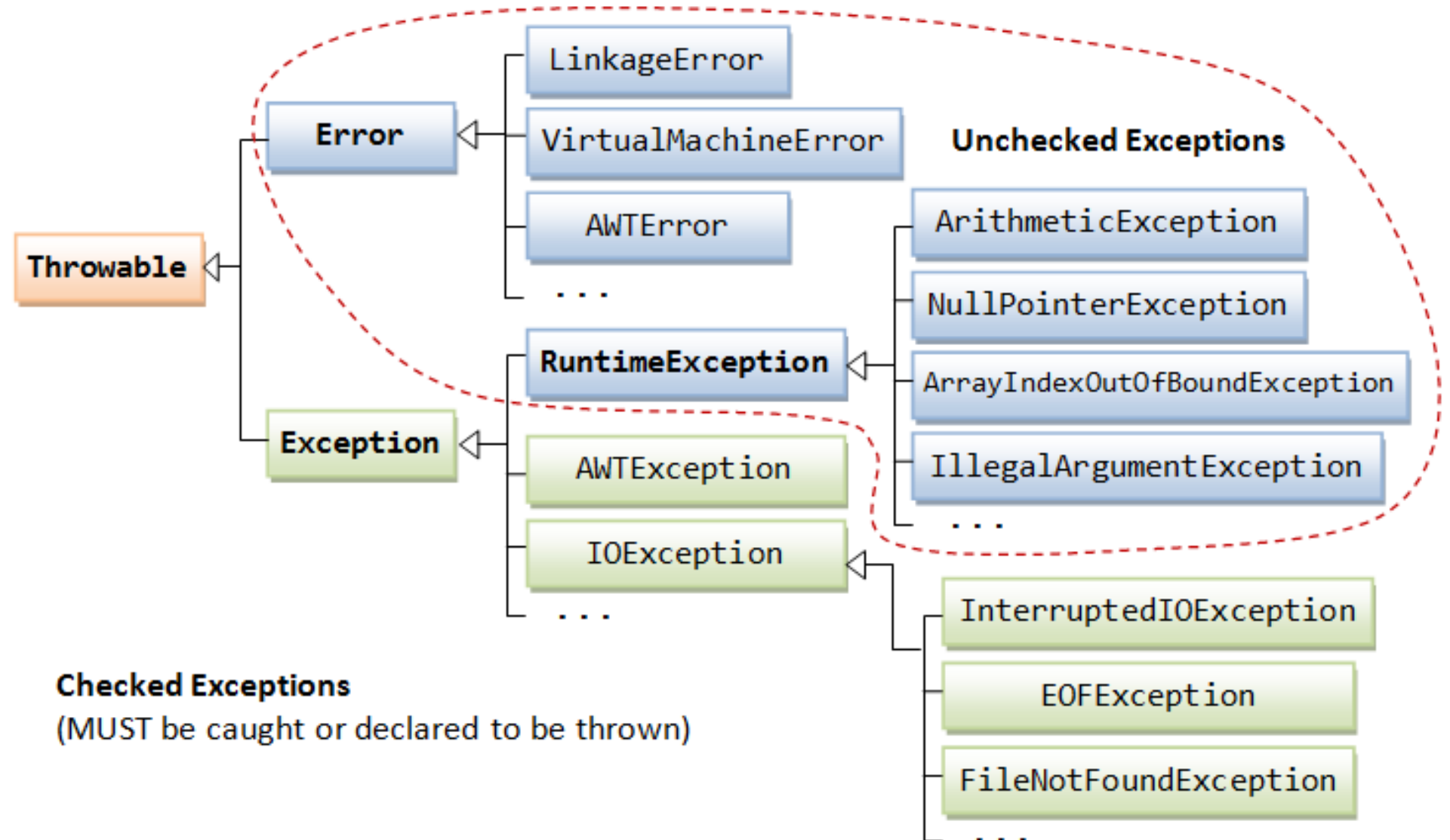
- La méthode qui lance l'exception
 - se termine prématurément.
 - ne renvoie aucune valeur
- Dans la méthode appelante
 - le contrôle ne revient pas au point d'appel de la méthode,
 - l'appelant ne peut donc pas continuer,
 - mais il peut "attraper" l'exception.
- Attraper les exceptions les plus spécifiques en premier, sinon vous ne les attraperez jamais !

To summarize

- The method that throws the exception
 - is terminated prematurely.
 - does not return any value.
- In the calling method,
 - the control does not return to the point of the method call,
 - so the caller can't continue,
 - but it can "catch" the exception.
- Catch the most specific exceptions first; otherwise, you will never catch them!

Les erreurs (**Error**) sont générées par la JVM pour indiquer des problèmes graves qui ne sont pas destinés à être traités par une application.

Exceptions



Exceptions contrôlées : sous-classes de **Exception**

Le compilateur

- signale que l'exception peut être levée (e.g., *TitleAlreadyUsedException*)
- attend de l'appelant
 - soit de la laisser passer (throws dans l'entête de la méthode), on ne sait pas comment la gérer

Segment createSegment(Point p1, Point p2, String title) **throws** TitleAlreadyUsedException

- soit de l'attraper (catch).

```
} catch (TitleAlreadyUsedException e) {  
    writeln("Le titre a déjà été utilisé :"+ e.getMessage());
```

Exceptions non contrôlées : sous-classes RuntimeException

- L'utilisation de ces exceptions est "non vérifiée" par le compilateur.
- Elles provoquent la fin du programme si elles ne sont pas attrapées.
- `IllegalArgumentException` est un exemple typique.

//ArrayIndexOutOfBoundsException: Index 3 out of bounds for length 3

```
int tab[] = new int[3];
```

```
tab[3]=1;
```

//ArithmeticException: / by zero

```
int i = 3/0;
```

//NullPointerException

```
Point p = null;
```

```
p.move(0,0);
```

Exceptions contrôlées et non contrôlées (checked)

- Les exceptions non contrôlées sont les classes RuntimeException, Error et leurs sous-classes.
 - A utiliser exclusivement pour les défaillances imprévues
 - Lorsque la récupération est improbable.
- Toutes les autres sous-classes d'exceptions sont des exceptions contrôlées :
 - Sous-classe d'exception
 - À utiliser pour les échecs anticipés.
 - Lorsque la récupération est possible.

Bien utiliser les exceptions

N'exposez pas vos implémentations par des exceptions non contrôlées !!

Par exemple, une méthode qui utilise un **tableau en interne** ne devrait jamais lever une *ArrayIndexOutOfBoundsException*!

Si, vraiment vous souhaitez lever exception alors levez *IndexOutOfBoundsException* ou une *erreur orientée métier* par exemple *NoSuchIndexedElementException*

```
SystemElement get(int index) {  
    try {  
        return elements[index];  
    } catch (ArrayIndexOutOfBoundsException a) {  
        throw new IndexOutOfBoundsException(index); //To hide implementation  
    }  
}
```

```
@Test  
void testget() {  
    SystemElementSet ses = new SystemElementSet();  
    assertThrows(IndexOutOfBoundsException.class, ()-> ses.get(ses.maxSize));  
}
```

Don't let an
exception betray
your
implementation.

Documenter les exceptions : javadoc

@throws ExceptionType reason

```
/**
 *
 * @param element to add
 * @throws TitleAlreadyUsedException all elements of the system must have different labels.
 */
public void addElement(SystemElement element) throws TitleAlreadyUsedException {
    Objects.requireNonNull(element);
    if (indexedElements.keySet().contains(element.getTitle())) {
        throw new TitleAlreadyUsedException(element.getTitle());
    }
    indexedElements.put(element.getTitle(), element);
    super.addElement(element);
}
```

Documenter (ou non) les exceptions : javadoc

Rappel : Javadoc, un contrat entre l'implémenteur et l'appelant

Le but de la balise `@throws` est d'indiquer **quelles exceptions le programmeur doit intercepter** (pour les exceptions contrôlées) ou **pourrait vouloir intercepter** (pour les exceptions non contrôlées).

Il est considéré comme une mauvaise pratique de programmation d'inclure des exceptions non contrôlées dans la clause `throws`.

Les documenter dans la balise `@throws` relève alors du jugement du concepteur d'API.

Pour en savoir plus :

<https://www.oracle.com/technical-resources/articles/java/javadoc-tool.html#throwstag>

Créer des exceptions

- S'il y a un bénéfice sinon utilisez l'une des exceptions standard, comme `UnsupportedOperationException` ou `IllegalArgumentException`. Tous les développeurs Java connaissent déjà ces exceptions.
- **Le nom doit se terminer par `Exception`**, ainsi elles sont faciles à identifier, e.g., *`TitleAlreadyUsedException`*.

The name of an exception must end with **Exception**, so they are easy to identify.

Ne cacher pas les exceptions

- Lorsque votre code attrape une exception avant de lancer votre exception personnalisée, vous ne devez pas cacher ce fait. L'exception attrapée est la « cause » qui contient généralement des informations essentielles dont vous aurez besoin pour analyser un incident de production.
 - Dans l'exemple suivant, l'exception `NumberFormatException` fournit des informations détaillées sur l'erreur. Vous perdrez ces informations si vous ne la définissez pas comme la cause de l'exception `BadFormatException`.

```
int readCoordinate(String s) throws BadFormatException {  
    try {  
        int a = Integer.parseInt(s);  
        return a;  
    } catch (NumberFormatException e) {  
        throw new BadFormatException(s, e);  
    }  
}
```

Don't hide an exception
with another exception;
record the first as the
cause of the second
exception!

Ne cacher pas les exceptions

```
public class BadFormatForCoordinateException extends Throwable {  
    public BadFormatForCoordinateException(String s, Throwable e) {  
        super("bad format for a coordinate : " +s, e);  
    }  
}
```

```
BadFormatForCoordinateException: bad format for a coordinate : 1q  
    at redThreadExample.ExceptionTests.readCoordinate(ExceptionTests.java:15)  
    at redThreadExample.ExceptionTests.testThrowableExceptions(ExceptionTests.java:24)
```

...

```
Caused by: java.lang.NumberFormatException: For input string: "1q"
```

N'ignorer jamais une exception !

Ne faites JAMAIS cela !

```
public void doNotIgnoreExceptions() {  
    try { // do something }  
    catch (NumberFormatException e) {  
        // this will never happen  
    }  
}
```

Never ignore an exception
by hiding it!



Au minimum :

```
log.error("This should never happen: " + e);
```

RuntimeExceptions ou Error

- Les RuntimeExceptions peuvent être attrapées : le code qui les a générées indique que quelque chose d'inattendu s'est produit mais la JVM elle-même n'est pas en danger.
- Au contraire, une Error exprime une exception qu'aucune "application raisonnable" ne voudrait jamais gérer : LinkageError, VirtualMachineError et ses sous-classes InternalError, OutOfMemoryError, StackOverflowError et UnknownError.

ET aucune exception n'est définie par : *extends Throwable* !

Catching errors is risky.

Never define subclasses of Throwable.

N'utilisez pas **throws Exception** !

```
public void doNotSpecifyException() throws Exception {  
    doSomething();  
}
```

- L'appelant de votre méthode sait que quelque chose pourrait mal tourner. ... une exception, c'est tout !
- Lorsque votre application change au fil du temps, la clause throws non spécifique cache toutes les modifications apportées aux exceptions auxquelles l'appelant doit s'attendre et qu'il doit gérer.

Do not use throws Exception!

Spécialiser votre gestion des exceptions

- il est facile de voir quel code lève quelles exceptions,
- il "sépare les préoccupations" plus clairement,
- il empêche la capture accidentelle d'exceptions : le programme continue à s'exécuter, produisant potentiellement le mauvais résultat.

```
public int getPlayerScoreBad(String playerFile) {  
    try (Scanner contents = new Scanner(new File(playerFile))) {  
        return Integer.parseInt(contents.nextLine());  
    } catch (Exception e) {  
        LOGGER.log(Level.WARNING, "You met a problem. ", e);  
        return 0;  
    }  
}
```

```
public int getPlayerScore(String playerFile) {  
    try (Scanner contents = new Scanner(new File(playerFile)) ) {  
        return Integer.parseInt(contents.nextLine());  
    } catch (FileNotFoundException e) {  
        logger.warn("Player file not found!", e);  
        return 0;  
    } catch (IOException e) {  
        logger.warn("Player file wouldn't load!", e);  
        return 0;  
    } catch (NumberFormatException e) {  
        logger.warn("Player file was corrupted!", e);  
        return 0;  
    }  
}
```

Eviter d'attraper des Exceptions sans discernement !

```
} catch (Exception e) {  
    System.out.println(« Exception » + e )  
}
```

Préférez des traitements spécifiques par type d'exceptions

Et surement pas Throwable !

Specialize your exception handling

Si vous attrapez une exception pour la relancer, ne la logguez pas


```
SystemElement get(int index) {  
    try {  
        return elements[index];  
    } catch (ArrayIndexOutOfBoundsException a) {  
        //Not log the exception a !  
        throw new IndexOutOfBoundsException(index); //To hide implementation  
    }  
}
```

- Vous ne savez pas comment l'appelant souhaitera la traiter.
Ne multiplier pas les logs.

Log it when you handle it !

Exceptions should not be the norm, but

- **Mixing data and control should be avoided.**
 - The alternative to throwing an exception is often returning a null value from a method.
 - This means that the return value encapsulates two meanings (success or failure and whatever the data means when present). It is good programming practice to avoid this usage where possible.
 - If null is a reasonable value for the stated purpose of a method, or if a method is expected to fail often in the normal course of operation, then it is reasonable to return null to indicate failure; otherwise it is better to throw an exception.

A close-up photograph of a mechanical assembly. In the background, the word "stop" is printed in red on a light-colored surface. In the foreground, there are dark, metallic components, including a cylindrical part with a hole and a rectangular block with a hole. The lighting is dramatic, with strong highlights and shadows.

stop

Assertive Programming

Use Assertions to Prevent the Impossible

Assertions

```
void addElement(SystemElement element) throws TitleAlreadyUsedException, IndexOutOfBoundsException{
```

```
    assert element != null : "element to add is null";
```



Condition



Texte utilisé si l'exception est levée.

Erreur levée en cas d'assertion fausse

java.lang.AssertionError: element to add is null

La classe AssertionError étend Error; c'est une exception non contrôlée.

Les AssertionErrors sont destinées à indiquer des conditions irrécupérables dans une application, donc **n'essayez jamais de les gérer ou de tenter une récupération.**

Utilisation des assertions

Les assertions peuvent être désactivées, donc elles ne sont pas toujours testées.

Quand utiliser les assertions

- Vérifiez toujours les valeurs nulles et les options vides non attendues
- Utilisez les dans les endroits du code qui ne seront jamais exécutés, comme le cas par défaut d'une instruction switch ou après une boucle qui ne se termine jamais
- **Vérifiez les arguments des méthodes privées** : Ces arguments sont fournis par le code du développeur uniquement et le développeur peut vouloir vérifier ses hypothèses sur ces arguments.

Quand ne pas utiliser les assertions

- N'utilisez pas des assertions pour vérifier les entrées dans une méthode publique et utilisez plutôt une exception non vérifiée telle que *IllegalArgumentException* ou *NullPointerException*
- Les assertions ne doivent jamais être utilisées pour remplacer les messages d'erreur

A ne pas faire

N'appellez pas de méthodes dans des conditions d'assertion et affectez plutôt le résultat de la méthode à une variable locale et utilisez cette variable avec assert



« Nullability »

Ou comment gérer la valeur Null ?

Gérer l'absence de valeur : la valeur Null (Nullability)

Nous avons de multiples occasions de donner la valeur Null à une variable

- temporairement non initialisée :
 - Par exemple, si aucun Station n'est référencée, nous retournons la valeur Null.
- mal initialisée
 - Par exemples, un constructeur qui n'initialise pas la variable; un cas d'erreur; ...
- indicateur qui représente correctement l'absence d'une valeur utile dans le cycle de vie normal d'un objet
 - Par exemple, tant qu'aucune batterie n'est associée à un vélo électrique, la valeur de la variable d'instance batterie a pour valeur Null.
- **Cette valeur est une sorte d'indicateur qui doit être interprété d'une manière spéciale.**
 - Par exemple, la méthode get de Map renvoie Null quand l'objet recherché n'existe pas;
- **Une conséquence fréquente est : NullPointerException**

Si vous devez interdire de donner la valeur null à un paramètre

```
public class Card {  
    private Rank aRank; // Should not be null  
    private Suit aSuit; // Should not be null
```

```
    public Card(Rank pRank, Suit pSuit) {  
        aRank = pRank;  
        aSuit = pSuit;  
    }  
}
```

Méthode PUBLIC, donc on doit vérifier que pRank n'a pas pour valeur Null

Pour interdire de donner la valeur null à un paramètre : lever une exception *IllegalArgumentException*

```
public class Card {  
    private Rank aRank; // Should not be null  
    private Suit aSuit; // Should not be null  
  
    /** * @pre pRank != null && pSuit != null; */  
    public Card(Rank pRank, Suit pSuit) {  
        if( pRank == null || pSuit == null){  
            throw new IllegalArgumentException();  
        }  
        aRank = pRank; aSuit = pSuit;  
    }  
}
```

Calls can introduce a null value, which would be detected very late.

Let's raise the `IllegalArgumentException` to protect our code!

Les appels peuvent introduire une valeur nulle, qui serait détectée très tard.

Levons l'exception **`IllegalArgumentException`** pour protéger notre code !

Interdire de donner la valeur null à un paramètre : Utiliser Objects et lever une exception NullPointerException

```
public class Card {  
    private Rank aRank; // Should not be null  
    private Suit aSuit; // Should not be null  
  
    /** * @pre pRank != null && pSuit != null; */  
    public Card(Rank pRank, Suit pSuit) {  
        Objects.requireNonNull(pRank);  
        Objects.requireNonNull(pSuit);  
        aRank = pRank; aSuit = pSuit;  
    }  
}
```

```
java.lang.NullPointerException  
    at java.base/java.util.Objects.requireNonNull(Objects.java:222)
```

Une autre façon de protéger
notre code !

Plus court, plus facile à lire, ...
Mais bien lire le message
d'erreur au-delà du
NullPointerException ...

Programmer oriented.

Interdire de donner la valeur null à un paramètre : utiliser une assertion

```
public class Card {  
    private Rank aRank; // Should not be null  
    private Suit aSuit; // Should not be null  
  
    private Card(Rank pRank, Suit pSuit) {  
        assert pRank != null && pSuit != null;  
        aRank = pRank; aSuit = pSuit;  
    }  
}
```

Nous avons la responsabilité de notre code.
Utilisons une assertion;
Cela ne devrait jamais arriver, mais au cas où nous identifierons plus vite l'erreur.

Si vous devez échouer, ne faites rien d'autres

Par exemple, une API accepte plusieurs paramètres qui ne sont pas autorisés à être nuls, il est préférable de vérifier chaque paramètre non nul comme condition préalable de l'API.

```
public void goodAccept(String one, String two, String three) {  
    if (one == null || two == null || three == null) {  
        throw new IllegalArgumentException();  
    }  
  
    process(one);  
    process(two);  
    process(three);  
}
```

If you must fail, do nothing else

```
public void badAccept(String one, String two, String three) {  
    if (one == null) {  
        throw new IllegalArgumentException();  
    } else {  
        process(one);  
    }  
  
    if (two == null) {  
        throw new IllegalArgumentException();  
    } else {  
        process(two);  
    }  
  
    if (three == null) {  
        throw new IllegalArgumentException();  
    } else {  
        process(three);  
    }  
}
```

Gérer un indicateur, une carte Joker n'a ni rang ni couleur

```
public class Card {  
    private Rank aRank;  
    private Suit aSuit;  
    private boolean alsJoker;  
    public boolean isJoker() { return alsJoker; }  
    public Rank getRank() { return aRank; }  
    public Suit getSuit() { return aSuit; }
```

card.getRank().ordinal() sur un joker, déclenche une
NullPointerException => (if isJoker() etc...)

Optional



Gérer un indicateur par Optional

```
public class Card {  
    private Optional<Rank> aRank;  
    private Optional<Suit> aSuit;  
    public Card() { aRank = Optional.empty(); aSuit = Optional.empty(); }  
    public boolean isJoker() { return !aRank.isPresent(); } }  
    public Rank getRank() { return aRank; }
```

L'interface est

soit modifiée `Optional<Rank> getRank()`

soit ré-écrite en `aRank.get()` qui peut lever une erreur tout de suite.

Optional : faire savoir qu'une réponse peut être vide.

- En renvoyant un Optional, l'appelant sait que la réponse peut être vide et doit être gérée au moment de la compilation.
- Cela supprime notamment le besoin de toute vérification nulle dans le code client. Une réponse vide peut alors être gérée différemment à l'aide du style déclaratif de l'API de Optional.

Optional

- Créer un Optional vide :
 - `Optional<String> empty = Optional.empty();`
- Créer un Optional qui ne contient pas null
 - `String name = "baeldung";`
 - `Optional<String> optString = Optional.of(name);`
- Créer un Optional à partir d'une valeur qui pourrait être null :
 - `Element e = myList.get(« e »);`
 - `Optional< Element > opt = Optional.ofNullable(e);`
- Tester si un Optional contient ou non une valeur
 - `empty.isPresent(); //False`
 - `empty.isEmpty(); //true;`
 - `optString.isPresent(); //True`
 - `optString.ifPresent(name -> System.out.println(name.length()));`

Optional

- Valeur si Null

```
String nullName = null;
```

```
String name = Optional.ofNullable(nullName).orElse("john");
```

```
assertEquals("john", name);
```

- Etc.

- Voir <https://www.baeldung.com/java-optional>

Retourner une collection vide est préférable à Null.

- Lorsque une méthode renvoie une collection essayer de renvoyer une collection vide au lieu de null :

```
public List<String> names() {  
    if (userExists()) {  
        return Stream.of(readName()).collect(Collectors.toList());  
    } else {  
        return Collections.emptyList();  
    }  
}
```

Ainsi il n'est plus nécessaire de vérifier si la valeur de retour est null à l'appel de cette méthode.

Gérer la valeur Null

- D'autres solutions sont associées à des frameworks dont SPRING que vous verrez en SI4.
 - @NotNulll ect.

En résumé

- Évitez d'utiliser des références nulles pour représenter des informations portées par les objets ;
- Envisagez d'utiliser des types optionnels pour représenter les valeurs absentes, si nécessaire;
- Protéger votre code
 - De vos propres erreurs par des assertions
 - Des utilisateurs par la levée d'exception.

Defensive Programming

Defensive Programming ?

- Interagir avec le code d'autres personnes
 - Puis-je vraiment lui faire confiance? Comment va-t-il évoluer ?
- Recevoir des données (entrées d'un scanner par exemple) qui peuvent ou non être valides
- Protéger son code en vérifiant les informations échangées
 - Utiliser des exceptions pour s'arrêter dans un état que nous maîtrisons
 - Définir des contrats pour préciser les conditions d'usages de nos codes
 - Réduire la visibilité sur nos codes

Les programmeurs pragmatiques vont encore plus loin.

« Ils ne se font pas confiance non plus. »

Defensive Programming

- Mettre en place de bonnes pratiques de programmation qui vous protègent de vos propres erreurs de programmation, ainsi que de celles des autres.
- La programmation défensive est une approche dans laquelle le programmeur suppose qu'il est capable d'erreurs et peut donc appliquer les bonnes pratiques pour produire un code de haute qualité.

Programmation Défensive : Comment signaler une erreur due à un argument non conforme aux attentes de notre programme ?

- public class **IllegalStateException** extends [RuntimeException](#)

Signale qu'une méthode a été invoquée à un moment illégal ou inapproprié. En d'autres termes, l'environnement Java ou l'application Java n'est pas dans un état approprié pour l'opération demandée.

- public class **IllegalArgumentException** extends [RuntimeException](#)

Indique qu'une méthode a reçu un argument illégal ou inapproprié.
(beaucoup de sous classes)

PAS PAR UN AFFICHAGE !

Defensive Programming: Using Visibilities

- Client*-developer interaction
 - Code visible uniquement dans la classe déclarante – **private**
 - Code visible uniquement par les développeurs – par défaut (package-private)
 - Code destiné à être étendu par les clients – **protected**
 - Code destiné à être utilisé par les clients – **public**
- Code à valeur constante - **final**

* l'appelant, hors du package

Check for errors first

- Celui-ci est une convention stylistique.
- Dans votre code, il est bon de vérifier votre erreur avant toute autre chose. Laissez l'exécution normale du programme pour après.
- Par exemple, dans les méthodes qui lèvent des exceptions, essayez de vérifier les erreurs et de lever l'exception le plus tôt possible.

```
private SystemElement[] shallowCopyElementsIn(SystemElement[] source, SystemElement[] target) {  
    assert target != null;  
    assert target.length >= currentSize;  
    for (int i = 0; i < currentSize; i++) {  
        target[i] = source[i];  
    }  
    return target;  
}
```

Defensive & Offensive Programming

Les erreurs qui proviennent de l'extérieur des applications, comme les entrées de l'utilisateur sont traitées avec **tolérance aux pannes**. Une situation dans laquelle le logiciel continuera à fonctionner normalement en cas de défaillance ou d'erreur survenant dans ses composants.

Les erreurs provenant du programme lui-même doivent être exemptées de la tolérance totale aux pannes.

Deux méthodologies sont appliquées dans la programmation offensive :

- Faire confiance à la validité des données internes
- Faire confiance aux composants logiciels

Defensive/Offensive programming

- Normally, defensive programming emphasizes total **Fault Tolerance**. A situation where the software will continue working normally in the event of failure or an error arising within its components.
- **Offensive Programming**, however, takes a different approach on that. According to offensive programming, the errors that should be handled defensively should come from outside the applications, such as user input. Errors arising from within the program itself should be exempted from total Fault Tolerance. There are two methodologies applied in Offensive Programming:
 - Trusting internal data validity
 - Trusting Software components

Defensive/Offensive programming : **Trusting internal data validity**

```
public String ledLight_colorname(Color c) {  
    if (c.equals(WHITE) )  
        return "white";  
    if (c.equals(GREEN) )  
        return "green";  
    if (c.equals(BLUE) )  
        return "blue";  
  
    return "black"; // handled as a dead LED light.  
}
```

**Overly
Defensive
Programming**

```
public static String ledLight_colorname(Color c) {  
    if (c.equals(WHITE) )  
        return "white";  
    if (c.equals(GREEN) )  
        return "green";  
    if (c.equals(BLUE) )  
        return "blue";  
    assert false : " a led can't have such a color"; // Assert that this  
    section is unreachable.  
    return ""; //for the compiler !!  
}
```

Offensive programming

```
@Test  
void ledLight_colornameTest(){  
    assertEquals("blue", Color.ledLight_colorname((Color.BLUE)));  
    assertEquals("black", Color.ledLight_colorname(new Color(1,2,3)));  
}
```

**java.lang.AssertionError:
a led can't have such a color**

Defensive/Offensive programming : **Trusting Software Components**

Overly Defensive Programming

```
if (is_legacy_compatible(user_config)) {  
    // Strategy: Don't trust that the new code behaves the same  
    old_code(user_config);  
} else {  
    // Fallback: Don't trust that the new code handles the same cases  
    if (new_code(user_config) != OK) {  
        old_code(user_config);  
    }  
}
```

Offensive Programming Solution of the above code.

```
// Expect that the new code has no new bugs  
if (new_code(user_config) != OK) {  
    // Loudly report and abruptly terminate program to get proper attention  
    report_error("Something went very wrong");  
    exit(-1);  
}
```

Defensive Programming is the development of computer software putting in mind all the unforeseen circumstances that could raise problematic issues. That allows the software to behave correctly despite the input provided. This development technique is mainly utilized when coding software that should be highly available, safe, and secure from malicious attempts. Defensive Programming techniques help improve Software and Source code through:

- Improving General Quality: Completely minimizes the number of bugs and problems that could arise with the code.
- Developing Comprehensible Code: Source code written with defensive coding techniques is easy to read and understand. That makes it to be easily approved in a code audit.
- Developing software that will provide a correct output despite the input given.

La programmation défensive est le développement de logiciels informatiques en tenant compte de toutes les circonstances imprévues qui pourraient poser des problèmes. Cela permet au logiciel de se comporter correctement malgré les entrées fournies. Cette technique de développement est principalement utilisée lors du codage d'un logiciel qui doit être hautement disponible, sûr et sécurisé contre les tentatives malveillantes. Les techniques de programmation défensive permettent d'améliorer le logiciel et le code source par le biais de ce qui suit :

Améliorant la qualité générale : Minimise complètement le nombre de bogues et de problèmes qui pourraient survenir avec le code.

Développer un code compréhensible : Le code source écrit avec des techniques de codage défensif est facile à lire et à comprendre. Il peut donc être facilement approuvé lors d'un audit de code.

Développer un logiciel qui fournira une sortie correcte malgré l'entrée donnée.

Conclusion

- Defensive programming can be tough to write source code, but it results in high-quality foolproof code. Without Defensive programming, your code will still run normally. However, it can easily break or give incorrect output depending on the condition or user input.
- *La programmation défensive peut être difficile à mettre en place, mais elle permet d'obtenir un code de haute qualité. Sans programmation défensive, votre code s'exécutera toujours normalement. Cependant, il peut facilement se casser ou donner une sortie incorrecte en fonction de la condition ou de l'entrée de l'utilisateur.*