Cours 12

Les exceptions

vous les avez pratiquées sans le savoir...

ArrayIndexOfBoundsException

NullPointerException

ClassNotFoundException

....

Les exceptions, vous connaissez...

```
public class EssaiCoursException  {
  public static void main(String[] argv)  {
      Scanner sc = new Scanner(System.io) ;
     S.o.p("Donnez le numérateur de la fraction : ");
     int numerateur = sc.nextInt();
      S.o.p("Donnez son dénominateur : ") ;
     int denominateur = sc.nextInt();
      double r = numerateur  / denominateur  ;
      System.out.println(r) ;
      System.out.println("fin du programme : au revoir") ;}}
```

A l'exécution : java EssaiCoursException

```
        Donnez le numérateur de la fraction :  12
  ...   Donnez son dénominateur :  0
        Exception in thread "main" java.lang.ArithmeticException: / by zero
             at EssaiCoursException.main(EssaiCoursException.java:9)
```

Analyse de l'exemple

```
.... int numerateur = sc.nextInt();
... int denominateur = sc.nextInt() ;

double r = numerateur  / denominateur  ;

System.out.println(r) ;
System.out.println("fin du programme") ;
```

Programme non robuste :
il sera interrompu s'il y a une division par 0
(*denominateur* = = 0)

ces instructions peuvent ne jamais être exécutées

Un programme est "robuste" s'il est capable de gérer certaines erreurs d'exécution sans s'interrompre

Avec ou sans exceptions ?

- Sans mécanisme d'exception
  - c'est le programmeur qui doit réaliser de nombreux tests

    if (denominateur == 0) ....

    - et... le programmeur peut oublier de tester une condition

  - ces tests se mélangent aux instructions « utiles » du programme
    - et... le code devient vite illisible car la partie utile est masquée par les tests

Principes de gestion d'erreurs,

Une gestion saine des erreurs doit permettre :

- d'un point de vue dynamique
  - détecter, récupérer et traiter des situations exceptionnelles /anormales à l'exécution
    - sans interrompre le programme
    - en détournant le déroulement du bloc vers un bloc dédié au traitement de ces situations anormales
    - en réparant la situation erronée (si possible)
- du point de vue de la lisibilité
  - améliorer la lisibilité du code
    - En séparant les instructions gérant le comportement normal du programme et celles traitant des situations exceptionnelles

Mécanisme des exceptions

- Tous les langages « modernes » incluent un mécanisme de gestion des erreurs
- Une exception est un signal qui
  - indique que quelque chose d'exceptionnel (par exemple une erreur) s'est produit
  - arrive à l'exécution d'un programme et interrompt le flot d'exécution normal du programme
- Deux actes liées aux exceptions
  - **lancer** une exception consiste à signaler la situation exceptionnelle
  - **attraper/capturer** une exception permet d'exécuter les actions nécessaires pour traiter cette situation exceptionnelle

Partie 1

Comment **traiter/attraper/capturer** une exception ?

Grâce aux mots-clés try et catch

- Par un bloc **try/catch**
  - On met les instructions susceptibles de lancer une exception dans le bloc `try`
  - On met dans le bloc catch les instructions qui seront exécutées en cas d'erreur
- bloc **try** : **surveiller** un bloc susceptible de déclencher une exception
- bloc **catch** : **récupérer/traiter/capturer** une exception
  - a toujours un paramètre : l'objet exception créé

Exemple : Traitement de l'exception « division par 0 »

```
Scanner sc = new Scanner(System.io) ;
S.o.p("Donnez le numérateur de la fraction : ") ;
int numerateur = sc.nextInt();
S.o.p("Donnez son dénominateur : ") ;
int denominateur = sc.nextInt();

try
 {
   double r = numerateur  / denominateur  ;
   System.out.prinln(r) ;
 }
 catch (ArithmeticException e)
 {
   System.out.println("une division par zéro est survenue") ;
 }
System.out.println("fin du programme") ;
```

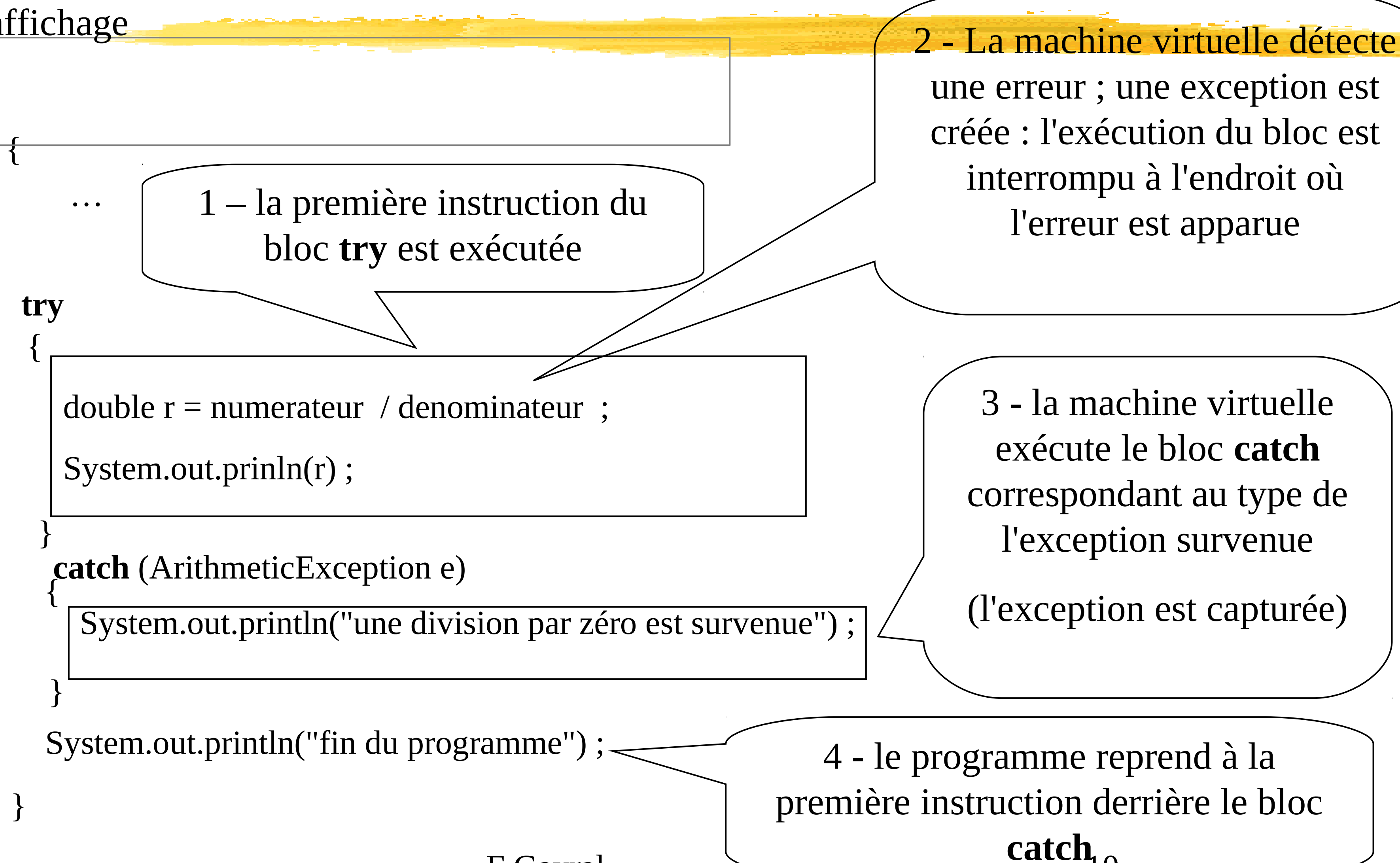
bloc d'instructions surveillées

bloc d'instructions à exécuter quand une exception de type *ArithmeticException* est créée suite à une erreur détectée dans le bloc **try**

Dynamique à l'exécution de try{...} catch (1/3)

En cas d'erreur : l'utilisateur entre 0

affichage



Dynamique à l'exécution de try{...} catch (2/3)

Sans erreur

affichage

1 - le bloc d'instructions délimité par **try** est exécuté entièrement

```
{
    …

 try
 {
   double r = numerateur  / denominateur  ;
   System.out.println(r) ;
 }
 catch (ArithmeticException e)
 {
   System.out.println("une division par zéro est survenue") ;
 }
 System.out.println("fin du programme") ;
}
```

2 - si aucune erreur n'apparaît toutes les instructions du bloc surveillé sont exécutées et le programme poursuit à la première instruction derrière le bloc **catch** (qui n'a pas été exécuté)

Dynamique à l'exécution de try{...} catch (3/3)

Dans tous les cas le programme n'est pas interrompu

```
{
      …

 try
  {
     System.out.prinln(r) ;
  }
  catch (ArithmeticException e)
  {
     System.out.println("une division par zéro est survenue") ;
  }
  System.out.println("fin du programme") ;
}
```

...le programme poursuit toujours à la première instruction derrière le bloc **catch**

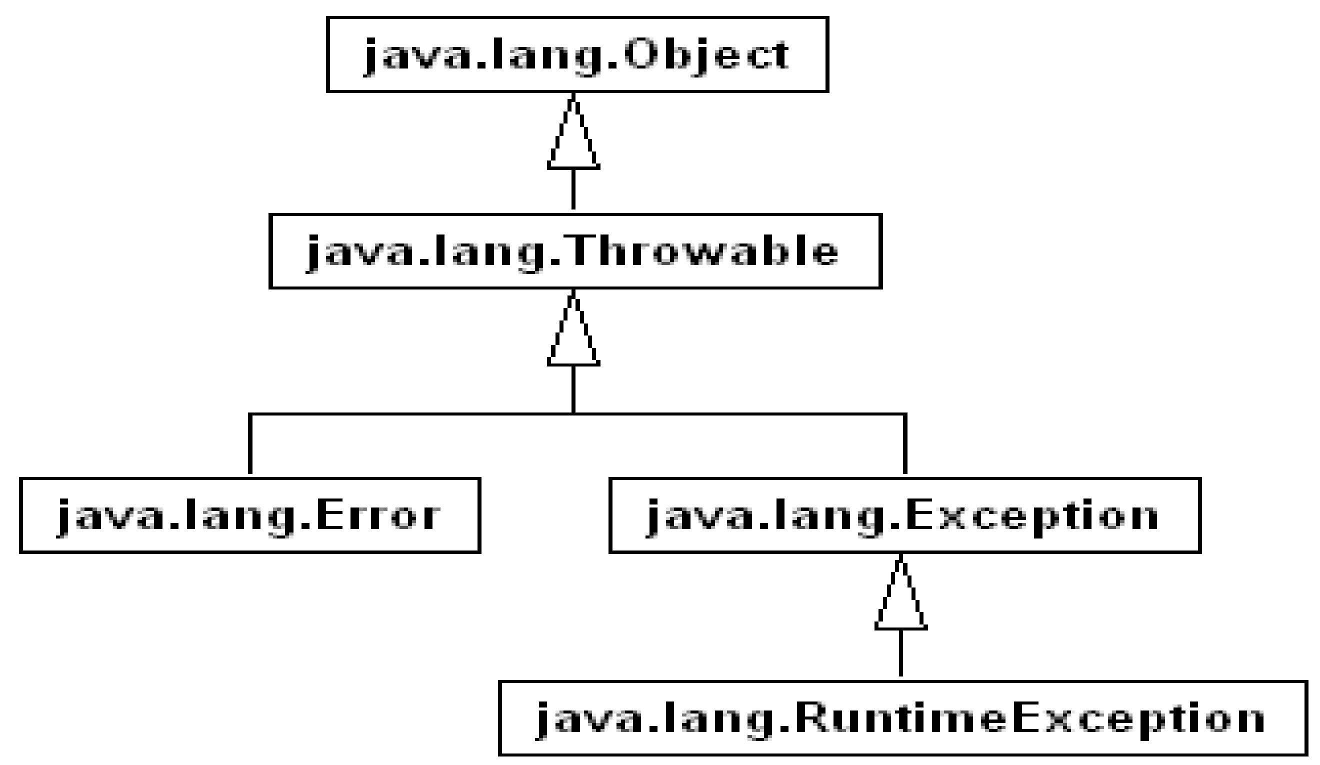
Récupérer une exception

```
{….

try
 {
   double r = numerateur  / denominateur  ;
   System.out.println(r) ;
 }
  catch (ArithmeticException e)
  {
    System.out.println("une division par zéro est survenue") ;
    denominateur = 4;
  }
  System.out.println(numerateur  / denominateur);
  System.out.println("fin du programme") ;
  }
```

Exception : classe Java

- Diagramme des classes d'exceptions



Java : 2 types d'erreurs => 2 branches dans la hiérarchie d'exceptions

- 2 types d'erreurs => 2 branches dans la hiérarchie d'exceptions
  - classe **Error** pour les erreurs graves intervenues dans la machine virtuelle Java qui conduisent à l'arrêt du programme
  - classe **Exception** représente des erreurs moins graves.

Exemple d'exceptions héritées de Error

```
public class ErreurMemoire {

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

déclenche l'exception :

java.lang.**OutOfMemoryError** (extends Error)

La machine virtuelle java ne dispose plus d'assez de mémoire pour pouvoir allouer ces chaînes

Exemple d'exceptions héritées de **RuntimeException**

```
public class TestAutreException {

    public static void main(String[] args) {

            String chaine="1234";
            CompteBancaire c =(CompteBancaire) chaine;
            System.out.println("fin du programme");     }}
```

A l'exécution : java TestAutreException

Exception in thread "main" java.lang.**ClassCastException**: java.lang.String cannot be cast to CompteBancaire

    at TestAutreException.main(TestAutreException.java:7)

Exceptions = objets

- Les exceptions sont des instances de classes données, elles contiennent
  - des variables d'instance
    - Message à afficher quand l'erreur se produit
  - des méthodes particulières
    - Constructeur avec un argument de type String (le message)
    - Méthodes
- différents packages pour les classes Exception
  - java.lang
  - java.io (IOException)

Traitement des exceptions : contraintes des blocs try/catch

- Un bloc try doit nécessairement être suivi d'un bloc catch ou d'un bloc finally sinon erreur à la compilation

  'try' without 'catch' or 'finally'

  ```
  try{
  ^
  ```

Les intructions surveillées par un bloc try peuvent lever plusieurs types d'exceptions ; le bloc try peut donc être suivi de plusieurs catch

```
try {...      }
catch (ArithmeticException e)   {...}
catch (ClassCastException exc) {}
catch (Exception ex) {...}
```

Traitement des exceptions
ordre des catch

- Lorsque une exception est levée et s'il y a plusieurs catch consécutifs, **un seul bloc catch** va être exécuté :
  - le premier susceptible d’attraper l’exception
    - les catch sont parcourus en séquence
    - Le choix se porte sur celui dont la classe d'exception est celle de la classe de l'exception levée (ou compatible)

⇒ L'ordre des blocs catch est donc important

Bloc catch traitant plusieurs exceptions (depuis la version 1.7)

```
try  {...      }
catch (ArithmeticException e)  {S.o.p (e.getMessage());}
catch (ClassCastException exc) {S.o.p (exc.getMessage());}
```

Attrape l'une ou l'autre des exceptions

```
try  {...      }
catch (ArithmeticException | ClassCastException e)     {
                              S.o.p (e.getMessage());}
```

clause finally

- Après un try/catch, on peut mettre une clause finally où le code sera exécuté systématiquement

  quelle que soit la manière dont s'est déroulée l'exécution du bloc try, exception ou pas

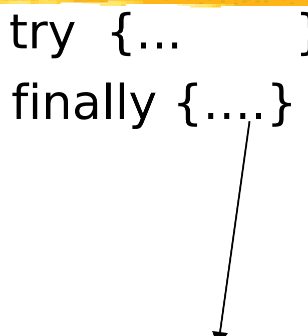
- Intérets d'une clause finally
  - libérer des ressources système, fermer des fichiers...
  - rassembler dans un seul bloc un ensemble d’instructions qui, autrement, auraient du être dupliquées

clause finally : 2 cas

```
try  {...       }
catch (ArithmeticException e)   {...}
catch (ClassCastException exc) {}
catch (Exception ex)     {...}
finally{.....}
```

Les instructions du bloc finally seront exécutées qu'on soit passé dans le catch ou pas

```
try  {...       }
finally {....}
```



Les instructions du bloc finally seront exécutées, qu'il y ait eu lancement ou pas d'une exception dans le bloc try

Le bloc finally est même exécuté avec un return dans le bloc try

Partie 2

Comment **lancer** une exception ?

Comment un programmeur peut-il lancer ses propres exceptions ?

- On a vu qu'il existait des exceptions implicites
  - implicitement créées par la machine virtuelle lorsqu'elle détecte certaines erreurs importantes (cf la sous-classe java.lang.RuntimeException)
    - ClassNotFoundException, ArithmeticException...

- Mais le programmeur peut explicitement **lancer** des exceptions lorsqu'il considère qu'une situation anormale arrive

Un exemple

```
public class Personne  {
private String nom ;
private float age ;

public Personne(String nom, float a)  {
      this.nom = nom;
      this.age = a;
}
```

Contrainte : l'âge d'une personne est strictement positif

⤇ impossible de construire une instance de Personne si a est négatif

Exception explicite : mots-clé throw/throws

throws dans la signature signale au compilateur que la méthode peut lancer une exception de type *Exception*

Un exception de type *Exception* est explicitement créée et lancée par **throw**

```
public class Personne    {

public Personne(String n, float a) throws Exception {
        if ( a < 0 ) {
            Exception e = new Exception("pas d'âge négatif") ;
            throw e ;   }
        this.age = a ;
        this.nom = n;
        }
}
```

```
// pg client
Personne p ;
try   {
        p = new Personne("toto", -8) ;
        System.out.println(p);
        }
catch (Exception e)   {
                System.out.println(e) ;
}
```

Signalement de lancement d'une exception : throws

- Le mot clé **throws** dans la signature d'une méthode signale que la méthode est susceptible de lancer une exception
- Une méthode peut lancer plusieurs exceptions

Exemple

```
public void setEncoding(String encoding) throws SecurityException,
                                    UnsupportedEncodingException
```

Tout programme qui fait appel à une méthode qui a **throws** dans sa signature doit

- soit la capturer dans un try/catch
- soit la relancer à son tour

Important

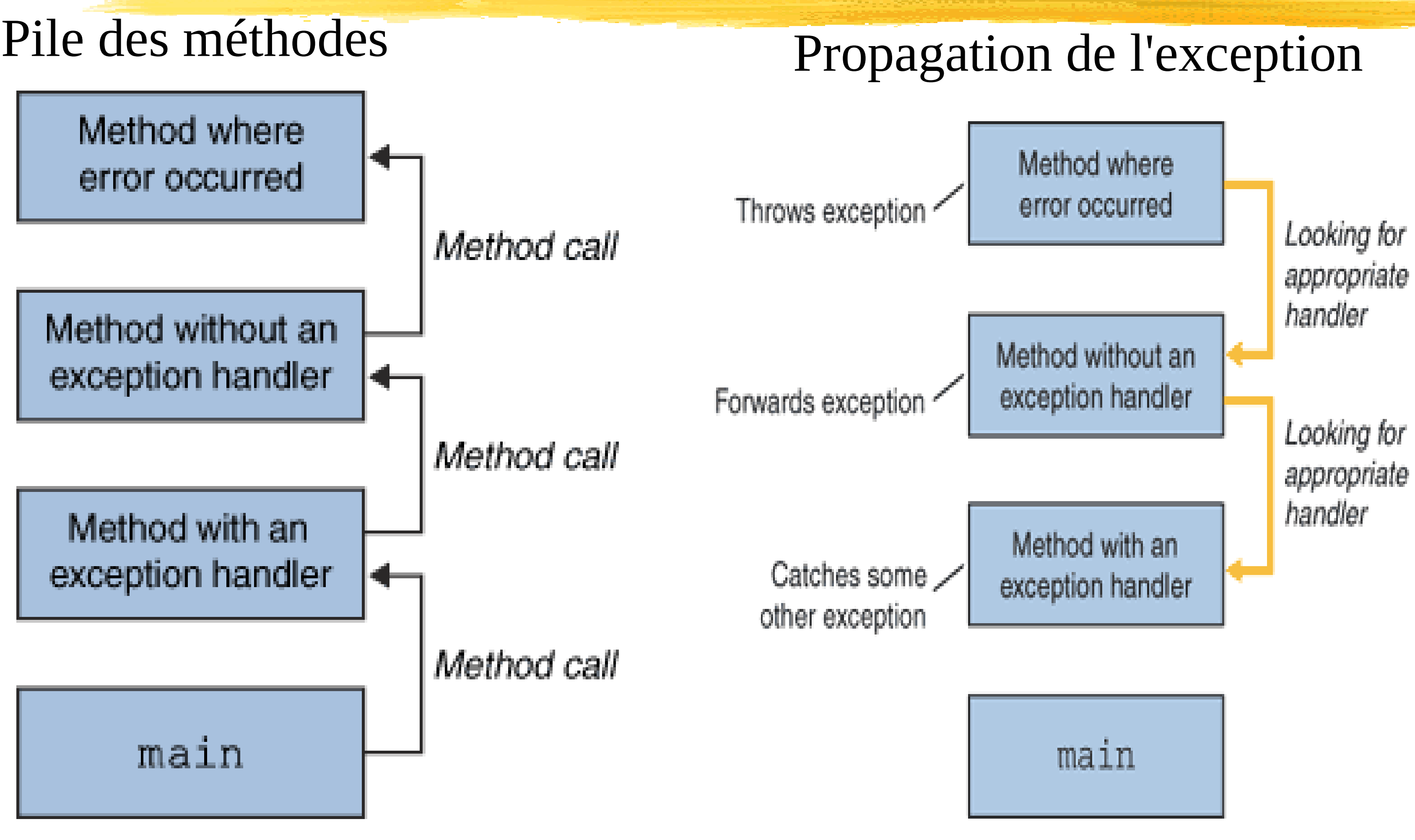
Délégation et propagation d'une exception

Une exception n'est pas forcément capturée tout de suite

Son traitement peut être déléguée à la méthode appelante

=> Mot-clé **throws**

Propagation d'une exception



```
public class PropagationPourCours {
private boolean probleme;

public PropagationPourCours (boolean pb){
this.probleme = pb;      }

public void methodeBasse() throws Exception {
        if (this.probleme) throw new Exception("problème pour demo du cours");
        System.out.println("pas de problème");      }

public void methodeMoyenne() throws Exception {
        this.methodeBasse();          }

public void methodeHaute() {
     try {
        this.methodeMoyenne();
     }
    catch(Exception e) {
        System.out.println("attrape..."+e);      }
  }

public static void main(String[] arg) {
      PropagationPourCours p= new PropagationPourCours(true);
      p.methodeHaute();   }    }
```

Principe de propagation d'une exception

- Si une exception est lancée dans un bloc try d'une méthode, le contrôle du programme est transféré au **premier catch** récupérant cette exception dans la pile des appels de méthodes

  ⇒ remontée dans la pile des appels

- La pile est alors dépilée jusque là et le code du bloc catch est exécuté puis les instructions suivant le bloc catch
- Si aucun bloc catch valable n'existe dans la pile des appels de méthodes (exception jamais attrapée)
  - 1- propagation jusqu’à la méthode main()
  - 2- affichage des messages d’erreurs et la pile d’appels
  - 3- arrêt de l’exécution du programme

Exception : celles proposées par Java, celles créées par le programmeur

- Le programmeur peut créer ses propres classes d'exceptions si :
  - celles fournies par l'API ne conviennent pas
  - il veut transmettre des paramètres qui pourront être exploités lors de la capture de l'exception

- Solution : créer une sous-classe de la classe **Exception**
  - Définir éventuellement des variables d'instance mémorisant les « problèmes »
  - Définir un constructeur
  - Redéfinir la méthode toString()

Exemple de l'âge négatif pour une Personne : classe AgeNegatifException

```
public class AgeNegatifException extends Exception {
	private float age ;

public AgeNegatifException(float a, String message)   {
	super(message) ;
	this.age=a ;
}

public String toString() {
	return super.toString()+" age négatif : " +this.age ;
}
}
```

Le constructeur revisité

```
public class Personne     {
…..
public Personne(String n, float a) throws AgeNegatifException       {
        if ( a < 0 ) {
                        Exception e = new AgeNegatifException(a, "pas d'âge négatif") ;
                        throw e ;        }
        this.age = a ;
        this.nom = n;
        }
}
```

```
// pg client            Personne p ;
                        try   { p = new Personne(“toto”, -8) ;
                                System.out.println(p);}
                        catch (AgeNegatifException e)  {      System.out.println(e) ;       }
```