Programmation objet

les exceptions

1. Gérer des erreurs d’exécution
2. Vue approfondie des exceptions
3. Déléguer la capture d'une exception
4. Créer de nouvelles classes d’exceptions
5. Bons et mauvais usages des exceptions

1. Gérer des erreurs d'exécution

Robustesse d'un programme

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

```
{
    int numerateur = Lire.jint("nume ?") ;
    int denominateur = Lire.jint("deno ?") ;

    double r = numerateur  / denominateur  ;

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

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

cette instruction peut ne jamais être exécutée

Principes de gestion d'erreurs

Une gestion saine d'erreurs doit permettre

d'un point de vue dynamique :

- de détecter, récupérer et traiter des erreurs sans interrompre le programme
- (si possible) de réparer la situation erronée

du point de vue de la lisibilité :

- de séparer les instructions gérant le comportement normal du programme des instructions gérant les erreurs

2. Un aperçu sur les exceptions

Exceptions

Java propose un mécanisme de gestion d'erreurs, appelé *exception*, satisfaisant tous les principes précédents

Une *exception* peut être vue comme un déroutement de la séquence normale d'instructions d'un programme

Java utilise un principe explicite de surveillance d'instructions et crée une exception quand une erreur est détectée lors de l'exécution des instructions surveillées

try{...} catch

try défini un bloc d'instructions à surveiller

catch défini un bloc d'instructions à exécuter quand une erreur est détectée

```
try
{
   int numerateur = Lire.jint("nume ?") ;
   int denominateur = Lire.jint("deno ?") ;

   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**

plusieurs clauses **catch** sont possibles lorsque plusieurs types d'erreurs peuvent apparaître

Dynamique de try{...} catch (1/3)

En cas d'erreur

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

2 - si la machine virtuelle détecte une erreur, une exception est créée : l'exécution du bloc est interrompu à l'endroit où l'erreur est apparue

```
try
{
   int numerateur = Lire.jint("nume ?") ;
   int denominateur = Lire.jint("deno ?") ;

   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") ;
```

3 - la machine virtuelle exécute le bloc **catch** correspondant au type de l'exception survenue

(l'exception est capturée)

4 - le programme reprend à la première instruction derrière le bloc **catch**

Dynamique de try{...} catch (2/3)

Sans erreur

```
try
{
    int nume = Lire.jint("nume ?") ;
    int deno = Lire.jint("deno ?") ;
    Fraction f = new Fraction(nume, deno) ;
    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") ;
```

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

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

Dynamique de try{...} catch (3/3)

Dans tous les cas le programme n'est pas interrompu

```
try
{
    int nume = Lire.jint("nume ?") ;
    int deno = Lire.jint("deno ?") ;

    Fraction f = new Fraction(nume, deno) ;

    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") ;
```

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

Principe de lisibilité

Il n'est pas utile de surveiller des instructions ne provoquant pas d'erreurs.

```
{
    int nume = Lire.jint("nume ?") ;
    int deno = Lire.jint("deno ?") ;

  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") ;
}
```

Instructions non surveillées

Contient la première instruction susceptible de provoquer une erreur (*numerateur/denominateur*) et les instructions suivantes correspondant au comportement normal.

3. Vue approfondie des exceptions

Exceptions implicites et explicites

Il y a 2 catégories d'exceptions

Les exceptions implicites :

Elles sont implicitement créées par la machine virtuelle lorsqu'elle détecte une erreur.

Elles correspondent à des erreurs générales (division par zéro, invocation d'une méthode avec une référence **null**, ...)

Les exceptions explicites :

Elles sont explicitement créées par le programmeur lorsqu'il estime qu'une situation contrevient au déroulement normal du programme

Exception implicite

Exemple

```
{

    int numerateur = Lire.jint("nume ?") ;

    int denominateur = Lire.jint("deno ?") ;


    try
     {
        double r = numerateur  / denominateur  ;

        System.out.prinln(r) ;
      }
     catch (ArithmeticException e)
    {
       System.out.println(e) ;
     }

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

  }
```

la machine virtuelle créée et lève (propage) automatiquement une exception de type *ArithmeticException* s'il y a une division par zéro

Exception explicite

Exemple

```
{
      Personne p  ;

           String nom  = Lire.jString("nom ?") ;

           int age = Lire.int("age ?") ;

      try
       {
          if (age < 0)

               throw new Exception("pas d'age négatif") ;

          p = new Personne(nom, age) ;
       }
       catch (Exception e)
       {
            System.out.println(e) ;
       }

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

Un exception de type *Exception* est explicitement créée par ***new*** *Exception*

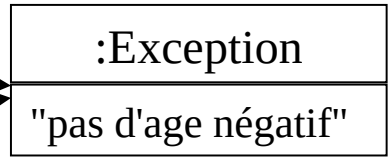
...et levée (propagée vers la clause **catch**) par **throw**

Une exception est un objet (1/2)

Exemple avec une exception explicite

```
{
    Personne p  ;
        String nom  = Lire.jString("nom ?") ;
        int age = Lire.int("age ?") ;
    try
     {
       if (age < 0)
           throw new Exception("pas d'age negatif") ;
       p = new Personne(nom, age) ;
     }
     catch (Exception e)
    {
          System.out.println(e) ;
     }
    System.out.println("fin du programme") ;
}
```

Un objet de la classe *Exception* est créée



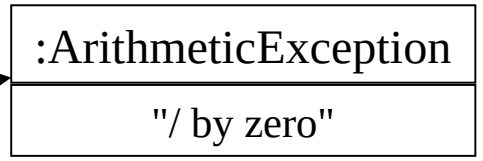
e est une variable-référence sur un objet de la classe *Exception*.
Si un tel objet a été créé au sein du bloc **try**, alors *e* contient sa référence.
e est strictement locale à la clause **catch( )** {…}.

Une exception est une objet (2/2)

Exemple avec une exception implicite

```
{
        int nume = Lire.jnt("nume ?") ;
        int deno = Lire.jint("deno ?") ;

   try
    {
       double r = numerateur / denominateur ;
       System.out.prinln(r) ;
    }
    catch (ArithmeticException e)
   {
       System.out.println(e) ;
    }

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

Si une division par zéro survient un objet de la classe *ArithmeticException* est implicitement créée



e est une une référence sur un objet de la classe *ArithmeticException*.
Si un tel objet a été créé dans le bloc *try*, alors *e* contient sa référence

Choix de la clause *catch* capturant une exception

Le type de l'exception créée détermine la clause **catch** qui capture (traite) l'exception

```
{
    Personne p  ;

        String nom  = Lire.jString("nom ?") ;

        int age = Lire.int("age ?") ;

    try
     {
        if (age < 0)  throw new Exception("pas d'age negatif") ;

        p = new Personne(nom, age) ;

       System.out.println(1.0/age) ;
      }

     catch (ArithmeticException e)
            { System.out.println(e) ; }

     catch (Exception e)
            { System.out.println(" problème d'age : " + e) ; }

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

si age < 0

:Exception

"pas d'age négatif"

une seule des 2 exceptions possibles peut avoir été créée et levée

:ArithmeticException

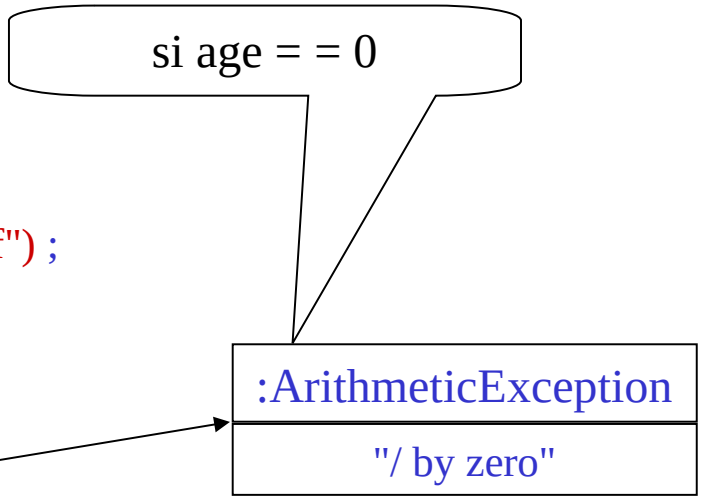
"/ by zero"

si age = = 0

L'ordre des clauses catch est important

Les clauses **catch** doivent être ordonnées de l'exception la plus spécifique à la plus générale

```
{
    Personne p  ;

        String nom  = Lire.jString("nom ?") ;

        int age = Lire.int("age ?") ;

    try
     {
        if (age < 0)  throw new Exception("pas d'age negatif") ;

        p = new Personne(nom, age) ;

       System.out.println(1.0/age) ;
      }

     catch (Exception e)
            { System.out.println("probleme d'age " + e) ; }

     catch (ArithmeticException e)
            { System.out.println(e) ; }

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



toute instance de *ArithmeticException* étant aussi instance de *Exception*...

...ce bloc **catch** ne sera jamais exécuté

finally

La clause *finally* défini un bloc d'instructions qui sera exécuté dans tous les cas de figure (exception levée ou pas)

```
public static double inverse(double x)
{
    double inv ;
    try
    {
      inv = 1 /x ;
      return inv ;
    }
    catch (ArithmeticException e)
    { System.out.println(e) ;
      return 0 ;
    }
    finally
    {
      System.out.print("on continue") ;
    }
}
```

affichage effectué dans tous les cas de figure (exception levée ou pas).
Note : une instruction **return** n'est exécutée qu'après que le bloc **finally** soit exécuté !

4. Déléguer la capture d'une exception

Principe de délégation

Une méthode (ou un constructeur) peut lever une exception et déléguer sa capture (traitement) à la méthode appelante

Toute méthode qui délègue un ou plusieurs types d'exceptions doit le signaler dans sa signature avec le mot clé **throws**.

Toute méthode appelant une méthode (ou un constructeur) déléguant des exceptions doit les capturer ou les déléguer aussi

Exemple

signale au compilateur que le constructeur peut déléguer une exception de type *Exception*

la méthode appelante (*main*) utilise le constructeur *Animal*. Elle doit donc obligatoirement capturer (ou déléguer à son tour) toute exception déléguée par ce constructeur

```
public class Animal
{
   private float poids ;

    public Animal(float p)  throws Exception
    {
         if (p < 0)
            throw new Exception("poids negatif") ;
        this.poids = p ;
    }

}   // fin classe Animal
```

```
public class TestAnimal
{
    public static void main(String[] args)
    {
        try
        {
            Animal a = new Animal(-15) ;
        }
         catch (Exception e)
                { System.out.println(e) ; }

      } // fin main
} // fin classe TestAnimal
```

Un exemple de délégations en cascade

```
public class Animal
{
    private float poids ;

    public Animal( ) {this.poids = 0f ;}

    public Animal(float p)  throws Exception
    {
        if (p < 0)
            throw new Exception("poids negatif") ;
        this.poids = p ;
    }

    public Animal autreAnimal ( ) throws Exception
    {
        Animal a = new Animal(this.poids - 10) ;
        return a ;
    }
}  // fin classe Animal
```

```
public class TestAnimal
{
    public static void main(String[] args)
    {
        Animal a1 =  new Animal( ) ;
        try
        {
            Animal a2 = a1.autreAnimal( ) ;
        }
        catch (Exception e)
        { System.out.println(e) ; }
    }  // fin main
}  // fin classe TestAnimal
```

1 – créé et lève une exception qui est déléguée vers l'appelant (méthode *autreAnimal*)

2 - l'exception reçue du constructeur *Animal(float p)* est à son tour déléguée vers l'appelant (*main*)

3 - l'exception est capturée

Délégation des exceptions non contrôlées

Toute exception non contrôlée est automatiquement créée par la machine virtuelle. Si elle n'est pas capturée par un **catch** dans la méthode courante elle est automatiquement déléguée à la méthode appelante.

Si la méthode appelante (*X*) ne la capture pas elle est de nouveau automatiquement déléguée à la méthode appelante de *X*.

Le processus se poursuit jusqu'à ce qu'une méthode capture l'exception. Si aucune ne le fait le programme est interrompu, et la machine virtuelle affiche l'exception

Exemple

si le dénominateur vaut 0 une exception non contrôlée de la classe *ArithmeticException* est automatiquement créée (levée). N'étant pas capturée dans la méthode elle est automatiquement déléguée à la méthode appelante (*main*)

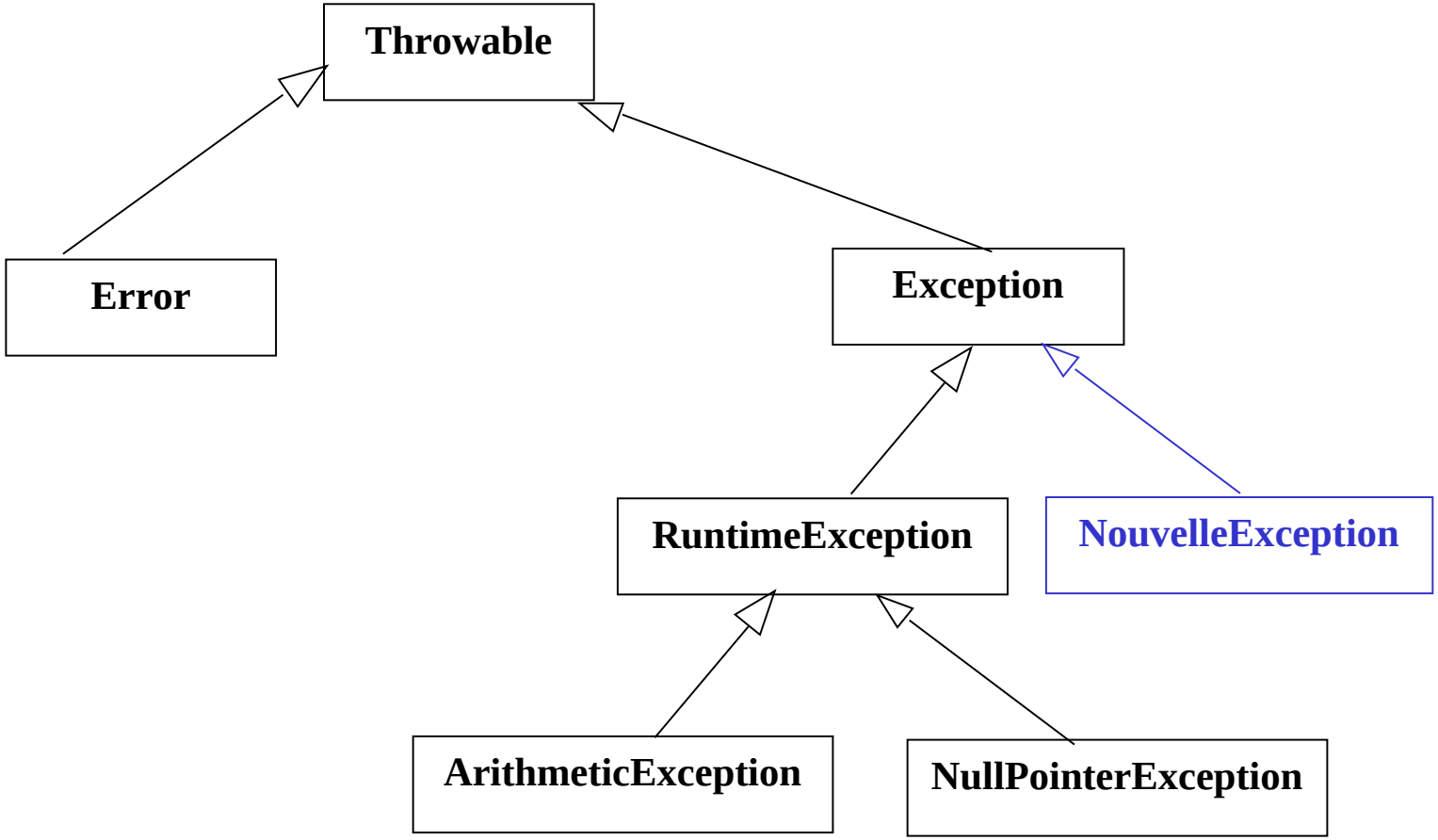
la méthode appelante (*main*) capture l'exception. Si elle ne le faisait pas, le programme serait interrompu et l'exception affichée (le *main* n'ayant pas d'appelant on ne peut continuer la délégation)

```
public class Fraction
{
    private int numerateur ;
    private int denominateur ;
     …

    public double valeurFraction( )
    {
       return (this.numerateur / this.denominateur) ;
    }

}  // fin classe fraction
```

```
public class TestFraction
{
    public static void main(String[] args)
    {
        Fraction f = new Fraction(12, 0) ;

        try
        {
            double r = f.valeurFraction( )  ;
        }
         catch (ArithmeticException e)
                { System.out.println(e) ; }

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

      }  // fin main
} // fin classe TestAnimal
```

5. Créer/définir de nouvelles classes d'exceptions contrôlées(sous-classes de Exception)

Intérêt : transmettre des paramètres via l'exception qui pourront être exploités lors de la capture de l'exception



Exemple

Transmettre la valeur ayant provoquée la levée de l'exception

```
public class PasAnimal extends Exception
{
    private float poidsFourni ;

    public PasAnimal(String mes,
    {                          float poids )
       super(mes) ;
       this.poidsFourni = poids
    }

    public float getPoidsFourni( )
    {
       return this.poidsFourni ;
    }

}   // fin classe PasAnimal
```

```
public class Animal
{
    private float poids ;

    public Animal(float p)  throws PasAnimal
    {   if (p < 0)
           throw new PasAnimal("poids negatif", p) ;

        this.poids = p ;
    }
}   // fin classe Animal
```

```
try
  {
      float p = Clavier.readFloat("poids ?") ;
       Animal a = new Animal(p) ;
   }
    catch (PasAnimal e)
    {float lePoids= e.getPoidsFourni( ) ; }
}
```

on récupère la valeur de poids ayant provoqué l'exception

6. Bons et mauvais usages des exceptions

1 - Une exception doit être utilisée dans les situations exceptionnelles. Elle ne doit pas se substituer aux contrôles usuels

2 - En général une méthode (autre que *main*) qui crée une exception la délègue vers la méthode appelante

(ainsi le traitement de l'exception est de la responsabilité de l'appelant qui peut choisir la façon de sortir de la situation erronée)

Une mauvaise utilisation d'une exception

```
public static boolean dansTableau(int[] t, int x)
{
    int i = 0 ;
    try
    {
        while (t[i] != x)
        { i++ ; }
    }
    catch (IndexOutOfBoundsException e )
            { return false ;}
    return true ;
} // dansTableau
```

contrôle indispensable

```
int i = 0 ;
while (i < t.length && t[i] != x)
{ i++ ; }

return (i < t.lengh) ;
```

on est hors tableau