

Exceptions et Flots de Contrôle

Tolérance aux fautes et Threads

Les exceptions

- ◆ Le modèle d'objets simplifie la tolérance aux erreurs
 - Une erreur est contenue au sein d'un objet
 - P.ex: un débordement de valeur dans la méthode *calculerSurface()* de Carré n'influence pas les données dans un autre objet
- ◆ Mais ... il faut un moyen pour récupérer les erreurs, sinon
 - l'exécution s'arrête avec un message d'erreur
 - Un objet client qui envoie le message *calculerSurface()* à un objet (provoquant un débordement de valeur) reste bloqué
 - P.ex: on envoie un message à un objet qui se trouve sur une autre machine, et le réseau est stoppé

Les exceptions

- ◆ Une exception est un signal qui indique qu'un événement anormal est survenu dans un programme
- ◆ La récupération (le traitement) de l'exception permet au programme de continuer son exécution.
- ◆ Une exception est un signal **déclenché** par une instruction et **traité** par une autre
 - 1. Il faut qu'un objet soit capable de signaler ou lever (**throw**) une exception à un autre objet
 - 2. Il faut que l'autre objet puisse saisir (**catch**) une exception afin de la traiter
 - P.ex: en Java, FileNotFoundException, NullPointerException, ArrayIndexOutOfBoundsException, UnknownHostException

Les exceptions

- ◆ Lorsque l'exception se produit le contrôle est transféré à un *gestionnaire d'exceptions*
- ◆ Séparation de l'exécution normale de l'exécution en cas de condition anormale
- ◆ Erreurs et Exceptions
 - Erreur: indications de problèmes irrécupérables dus au matériel ou au système d'exploitation
 - Exception: erreurs résiduelles dans le programme qui peuvent être traitées par une sortie propre ou une récupération (arithmétique, pointeurs, index, i/o, etc.)
- ◆ Attention: ne pas programmer par les exceptions !!!
 - Prévoir un maximum de cas possibles, et ne se fier au mécanismes d'exceptions que pour les cas d'erreurs résiduelles

Lever une exception

```
/* Dans Cercle */  
public float calculerSurface()  
    throws OverflowException  
{  
if (rayon>100000)  
    /* pour éviter un débordement */  
    throw new OverflowException();  
    else return rayon*rayon*3.14;  
}
```

Pour signaler une exception:

- ◆ Une méthode doit spécifier dans son entête quelles sont les exceptions possibles
 - L'exception fait partie du contrat entre le programmeur et le client
- ◆ Une exception est signalée avec l'opérateur *throw*

Traiter les exceptions

```
main(){
    Cercle c1 = new
        Cercle(2,3,4,ecran);

    try
        { float f =
          c1.calculerSurface();}
    catch (OverflowException e)
        { System.out.println("Caught
          exception" + e.toString());}
    finally
        { System.out.println("finally
          s'exécute toujours"); }
}
```

Pour traiter une exception:

- ◆ Dans le client, il faut vérifier si la méthode invoquée génère une exception ou pas
 - *try - catch - finally*
- ◆ **try**: essayer le code
- ◆ **catch** : ce code s'exécute si l'objet appelé génère une exception
- ◆ **finally**: s'exécute toujours

Traiter les exceptions

- ◆ L'instruction try/catch/finally
- ◆ **try**: définit un bloc de code dont les exceptions sont traitées
- ◆ **catch(UneException e)**: définit un bloc d'instructions permettant de prendre des mesures pour sortir de la condition exceptionnelle
 - une clause catch par exception possible
- ◆ **finally**: clause facultative qui permet de définir des opérations de nettoyage qui doivent toujours s'exécuter, indépendamment des exceptions et de leur traitement

Définir les exceptions

- ◆ Dans Java, une exception est aussi un objet !
 - On le crée par sa classe
 - `new OverflowException()`
 - On peut lui envoyer des messages
 - `e.toString()`
 - On peut passer l'exception en paramètre à un autre objet
 - La classe *Throwable* hérite de la classe *Object*

```
class OverflowException extends
    java.lang.Throwable{

    public OverflowException(){ }

    public String toString(){
        return new String("Débordement de
            valeur");
    }
}
```


Interpréteur Java et Exceptions

- ◆ Une instruction **throw e** est exécutée
 - l'interpréteur *stoppe* immédiatement l'exécution et *cherche* un gestionnaire d'exception (bloc catch) capable de saisir l'exception **e**
 - si oui, il *quitte* le bloc try, et va dans le bloc catch correspondant, *exécute* le code du gestionnaire, et continue ensuite l'exécution avec l'instruction qui suit *immédiatement* le gestionnaire
 - si non, la recherche continue sur le bloc englobant l'instruction
 - s'il ne trouve toujours rien, il cherche un gestionnaire au sein des blocs de code de la méthode *appelante* (propagation de l'exception **e**)
 - s'il ne trouve toujours rien, il remonte jusqu'au main() s'il le faut.
 - s'il ne trouve rien, le programme s'arrête.

Quand doit-on lever des exceptions?

- ◆ Le programmeur prévoit des cas d'exceptions
 - utiliser throws pour déclarer que la méthode peut lever une exception (éventuellement la définir)
- ◆ Si une méthode appelle une autre méthode qui peut lever une exception
 - soit, inclure du code de gestion d'exception
 - soit, utiliser throws pour déclarer que la méthode peut lever cette exception

Flot de Contrôle

```
class Cercle {...}  
class Carre {...}  
class Test {  
    public void maMethode {  
        c = new Cercle(3,4,5);  
        c1 = new Carre(5,3,2);  
        c2 = new Carre(1,2,2);  
        c.calculerSurface();  
    }  
}
```

```
class Cercle { ... }  
class Carre {...}  
class Test {  
    public static void main(...){  
        c = new Cercle(3,4,5);  
        c1 = new Carre(5,3,2);  
        c2 = new Carre(1,2,2);  
        c.calculerSurface();  
    }  
}
```

Quelle est la différence ?

Flot de Contrôle

- ◆ Rappel: les méthodes représentent un ensemble de primitives qui attendent d'être choisies par un client, sans contrainte d'ordre
- ◆ Sans `main()` il n'y a « rien » qui se passe
- ◆ La méthode `main()` représente le point d'entrée du programme.

Flot de Contrôle

- ◆ L'interpréteur cherche la méthode `main()` pour commencer l'exécution du programme
- ◆ Le programme s'arrête lorsque la méthode `main()` est terminée, mais
- ◆ ... le flot de contrôle peut parcourir beaucoup de méthodes par l'intermédiaire d'objets de différentes classes avant la fin du programme

Flot de Contrôle

```
class Cercle { ..  
    public Cercle(int a,int b,int r){ coordX=a;coordY=b; rayon=r;}  
    public float calculerSurface(){ return 3.14 * rayon*rayon; }  
...}
```

```
class Carre { ...  
    public Carre(int a,int b,int r){ coordX=a;coordY=b; cote =r;}  
    public float calculerSurface(){ return cote*cote; }  
...}
```

```
class Test {  
    public static void main(...){  
        c = new Cercle(3,4,5);  
        c1 = new Carre(5,3,2);  
        c2 = new Carre(1,2,2);  
        c.calculerSurface();  
    }  
}
```

2a

2b

5

3a

4a

3b

4b

1

Début de l'exécution

2

3

4

5

6

Appels bloquants

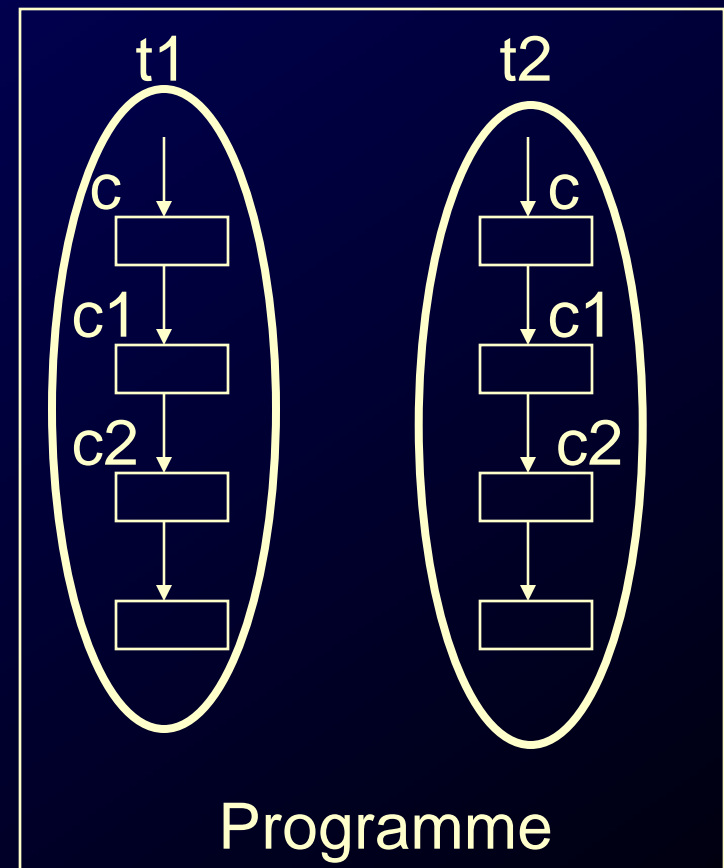
Fin de l'exécution

Threads

- ◆ Pour avoir **plusieurs** flots de contrôle au sein de la même application
- ◆ En démarrant plusieurs threads, vous démarrez plusieurs « activités » qui vont travailler en concurrence sur des objets partagés
- ◆ Un thread est un flot de contrôle séquentiel à l'intérieur d'un programme
 - Synonymes: contexte d'exécution, processus léger
- ◆ En Java: un thread est un objet

Threads

```
class ThreadGeometrique extends Thread{ ..  
    public ThreadGeometrique () {super();}  
    public void run(){  
        c = new Cercle(3,4,5);  
        c1 = new Carre(5,3,2);  
        c2 = new Carre(1,2,2);  
        c.calculerSurface();  
    }  
}  
class Test {  
    public static void main(String args[]){  
        t1 = new ThreadGeometrique ;  
        t2 = new ThreadGeometrique ;  
        t1.start(); /* non bloquant*/  
        t2.start(); /* non bloquant*/  
    }  
}
```



Threads

◆ Méthodes

- **run()** : permet de définir ce que va faire le thread
- **start()** : permet d'effectuer des initialisations et déclenche implicitement le run()
- **sleep()** : permet d'interrompre le run() pendant un certain temps
- **yield()** : donne une chance à un thread de même priorité de s'exécuter

Définir des Threads

- ◆ Sous-classer la classe Thread
 - redéfinir la méthode run()
- ◆ Invoquer le constructeur Thread(Runnable t)
 - t est un objet qui implémente l'interface Runnable et qui définit une méthode run()

Exécution Concurrente de Threads

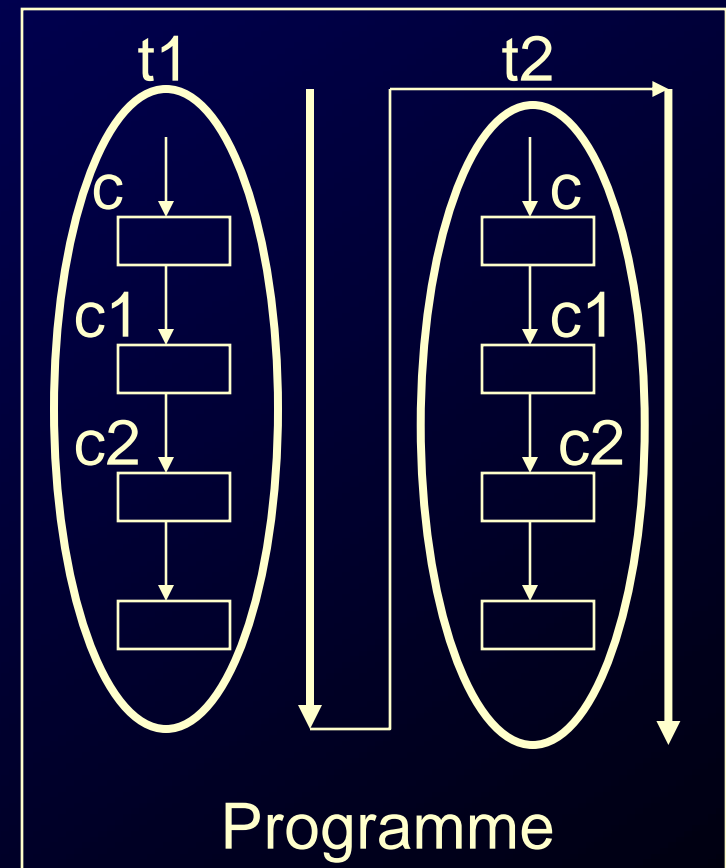
- ◆ Sur des machines à un seul CPU, il est nécessaire de partager le temps CPU entre les différents threads.
- ◆ L'algorithme de partage de temps est basé sur les *priorités*. Chaque thread reçoit une priorité.
- ◆ C'est le thread de priorité la plus haute qui commence son exécution
- ◆ S'il s'arrête ou suspend son exécution, alors un thread de priorité inférieure peut s'exécuter

Exécution Concurrente de Threads

- ◆ Pré-emption: dès qu'un thread de priorité supérieure est prêt à s'exécuter, le thread courant est stoppé
- ◆ Un thread peut laisser la main à un thread de même priorité en invoquant la méthode `yield()`.
- ◆ Si plusieurs threads de même priorité sont prêts à s'exécuter, un choix est fait entre eux (round-robin)
- ◆ **Si** le système permet le time-slice, un thread peut être interrompu en faveur d'un autre thread de même priorité.

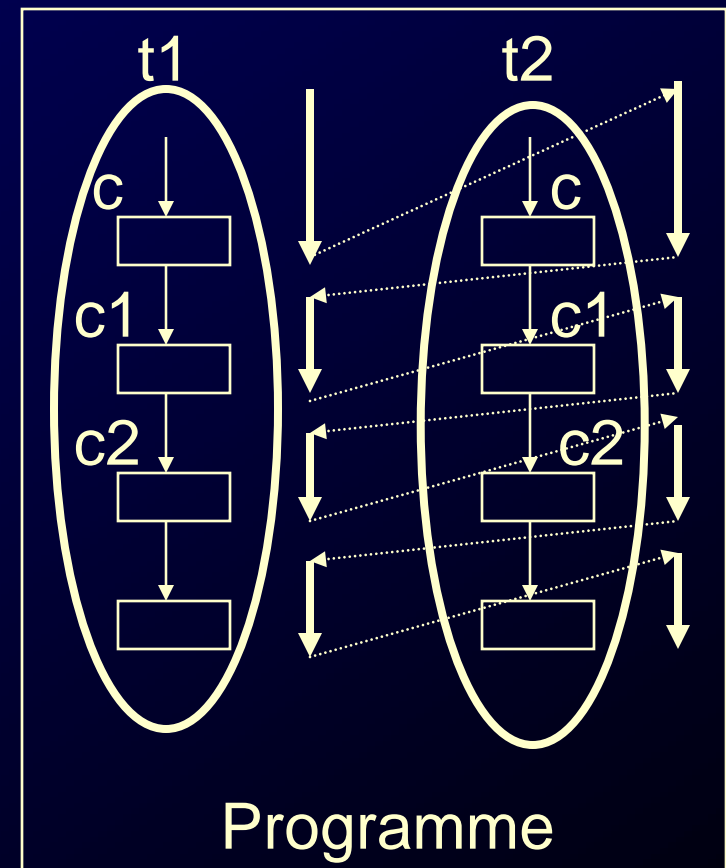
Exécution Concurrente de Threads

```
class ThreadGeometrique extends Thread{ ..  
    public ThreadGeometrique () {super();}  
    public void run(){  
        c = new Cercle(3,4,5);  
        c1 = new Carre(5,3,2);  
        c2 = new Carre(1,2,2);  
        c.calculerSurface();  
    }  
}  
class Test {  
    public static void main(String args[]){  
        t1 = new ThreadGeometrique ;  
        t2 = new ThreadGeometrique ;  
        t1.start(); /* non bloquant*/  
        t2.start(); /* non bloquant*/  
    }  
}
```



Exécution Concurrente de Threads

```
class ThreadGeometrique extends Thread{ ..  
    public ThreadGeometrique () {super();}  
    public void run(){  
        c = new Cercle(3,4,5); Thread.yield();  
        c1 = new Carre(5,3,2); Thread.yield();  
        c2 = new Carre(1,2,2); Thread.yield();  
        c.calculerSurface();  
    }  
}
```

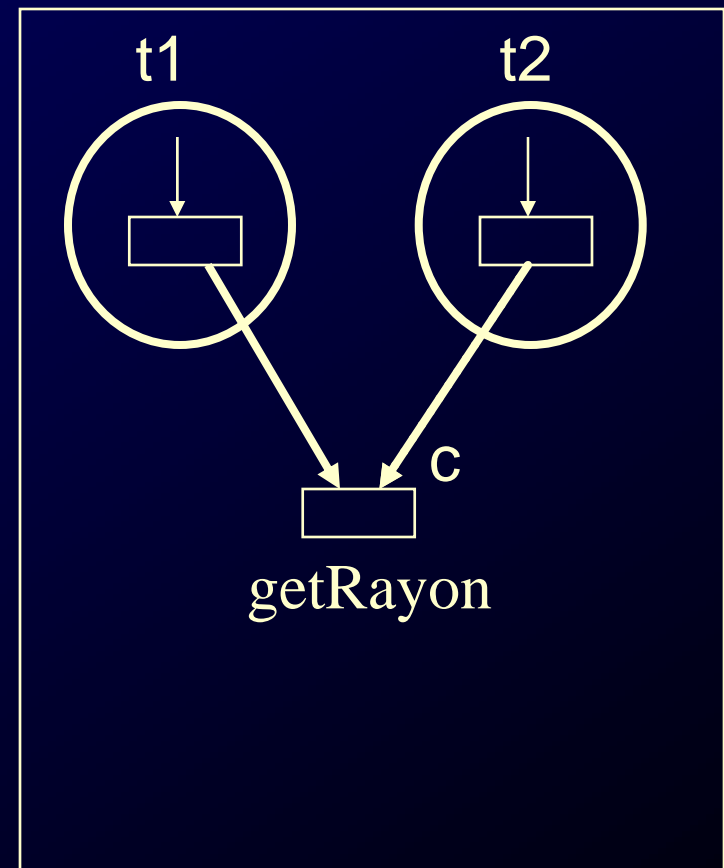


Synchronisation des Threads

- ◆ **Besoin de partager des données**
 - un thread qui lit dans un fichier, un thread qui écrit dans un fichier
 - le fichier ne doit pas être accédé en même temps en lecture et en écriture
 - utiliser des verrous (lock), **synchronized**
- ◆ **Besoin d'organiser / de coordonner les activités**
 - un thread attend sur un autre thread qu'une certaine information soit disponible, ou qu'une certaine activité soit terminée
 - un thread doit avoir un moyen pour informer un autre thread qu'une certaine activité a été réalisée
 - **wait(), notify(), notifyAll()**

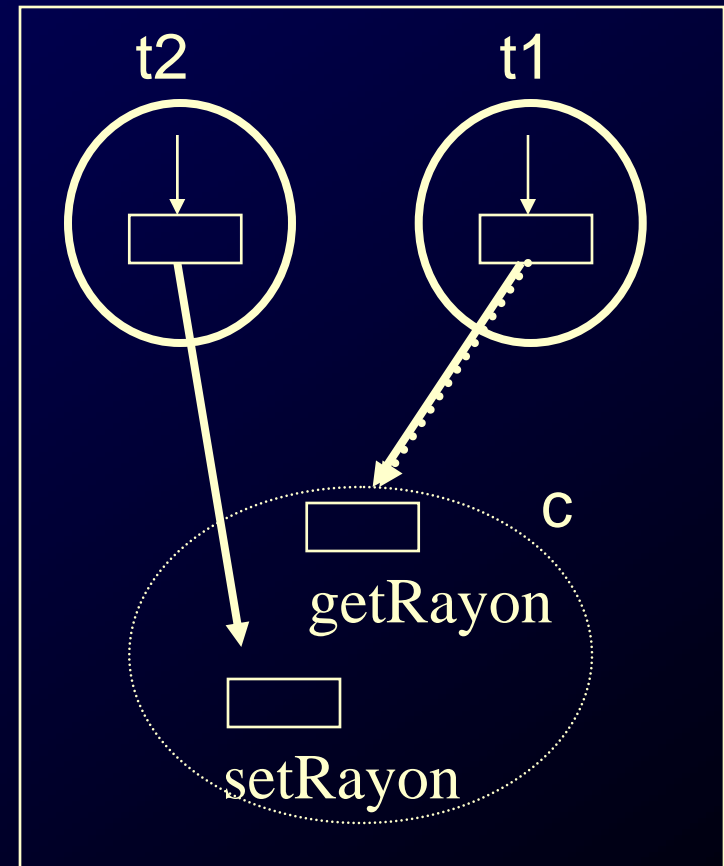
Exécution Concurrente de Threads

```
class Cercle { ...  
    public float synchronized getRayon(){  
        return rayon ;  
    }  
}  
class ThreadGeometrique extends Thread{  
    public ThreadGeometrique (Cercle c){  
        super(); Cercle cerclePartage = c;  
    public void run() {  
        float r=cerclePartage. getRayon(); }  
    }  
class Test {  
    public static void main(String args[]){  
        Cercle c = new Cercle(1,2,3);  
        t1 = new ThreadGeometrique(c) ;  
        t2 = new ThreadGeometrique(c) ;  
        t1.start(); t2.start(); }  
}
```



Exécution Concurrente de Threads

```
class Cercle { ...  
    public void setRayon(r){  
        rayon=r ;}  
    public float getRayon(){  
        return rayon ;}  
class ThreadGeometrique extends Thread{ ...  
    public void run(){  
        synchronized(cerclePartage){  
            float r=cerclePartage.getRayon(); }}}  
class ThreadModificateur extends Thread{ ...  
    public void run(){  
        synchronized(cerclePartage){  
            cerclePartage.setRayon(2); }}}  
class Test { /* ... main ... */  
    t1 = new ThreadGeometrique(c) ;  
    t2 = new ThreadModificateur (c) ;  
    t2.start(); t1.start(); } }
```



Synchronisation des Threads

◆ Synchronized

- verrou sur un objet
- 2 techniques:
 - *synchronized(cerclePartage) { /* section critique */ }*
 - *public void synchronized getRayon{ /* section critique */ }*
- le verrou est enlevé lorsque le thread sort de la section critique, ou

◆ wait(); notify(), notifyAll();

- pour coordonner des activités (nécessite un objet)
- Pour débloquer le verrou avant la fin du synchronized

Autres processus légers

- ◆ Applets: chargées à travers un browser
- ◆ Servlets: chargés à travers un serveur

