

# Fiabilité

**Université de Nice - Sophia Antipolis**

Version 1.8 – 7/12/07

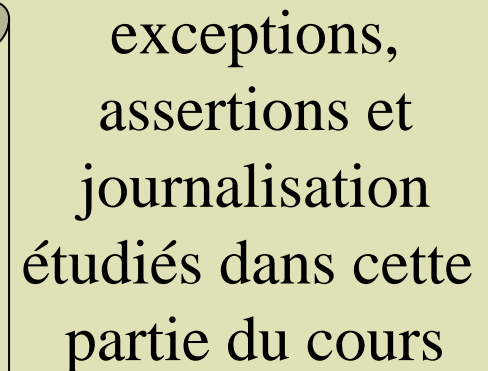
Richard Grin

# Fiabilité d'un logiciel

- La fiabilité d'un logiciel concerne
  - la **robustesse** : la capacité du logiciel à fonctionner, même en présence d'événements exceptionnels tels que la saisie d'informations erronées par l'utilisateur
  - la **correction** : le fait que le logiciel donne les résultats corrects lorsqu'il fonctionne normalement

# Fiabilité en Java

- 2 mécanismes principaux de Java facilitent une bonne fiabilité :
  - les exceptions pour la robustesse
  - les assertions pour la correction
- On peut aussi utiliser
  - l'API de journalisation (*logging*) ;  
`java.util.logging`
  - les débogueurs
  - les outils de tests d'unités (comme JUnit) ou de tests fonctionnels



exceptions,  
assertions et  
journalisation  
étudiés dans cette  
partie du cours

# Les exceptions

# Erreurs/exceptions

- On utilise les erreurs/exceptions pour traiter un fonctionnement **anormal** d'une partie d'un code (provoqué par une erreur ou un cas exceptionnel)
- En Java, une erreur ne provoque pas l'arrêt brutal du programme mais la création d'un **objet**, instance d'une classe spécifiquement créée pour être associée à des erreurs/exceptions

# Localisation du traitement des erreurs/exceptions

- Les traitements des erreurs et conditions « anormales » s'effectuent dans une zone du programme spéciale (bloc « **catch** »)
- Plutôt que de compliquer le code du traitement normal, on traite les conditions anormales à part
- Le traitement « normal » apparaît ainsi plus simple et plus lisible

# Traitement avec exception

```
try {  
    // lire la valeur de n au clavier;  
    int n = lireEntierAuClavier();  
    etagere.add(livre, n);  
}  
catch(NumberFormatException e) {  
    System.err.println("Mauvais numéro de livre");  
}  
catch {EtagerePleineException e) {  
    System.err.println("Etagere pleine");  
}
```

Traitement  
normal

# Bloc try-catch

```
try {  
    . . .  
    . . .  
}  
catch(TrucException e) {  
    // Traitement de l'exception e  
    . . .  
}  
catch(MachinException e) {  
    . . .  
}
```

Remarque : ne compile pas si aucune instruction du bloc `try` ne lance de `TrucException`

# Vocabulaire sur les erreurs/exceptions

- Une instruction, une méthode peut **lever** ou **lancer** une exception : une anomalie de fonctionnement provoque la création d'une exception

# Vocabulaire sur les exceptions (2)

- Une méthode peut attraper, saisir, traiter une exception par une clause **catch** d'un bloc **try-catch**

- Une méthode peut laisser se propager une exception :

- elle ne l'attrape pas

- l'erreur « remonte » alors vers la méthode appelante qui peut elle-même l'attraper ou la laisser remonter

si le sommet de la pile d'exécution est tournée vers le bas

# Mécanisme de traitement des exceptions

# Exception levée en dehors d'un bloc `try`

1. La méthode retourne immédiatement ;  
l'exception remonte vers la méthode appelante
2. La main est donnée à la méthode appelante
3. L'exception peut alors éventuellement être  
attrapée et traitée par cette méthode appelante  
ou par une des méthodes actuellement dans la  
pile d'exécution

# Exception levée dans un bloc `try`

- Si une des instructions du bloc `try` provoque une exception, les instructions suivantes du bloc `try` ne sont pas exécutées et,
  - si au moins une des clauses `catch` correspond au type de l'exception,
    1. la **première** clause `catch` appropriée est exécutée
    2. l'exécution se poursuit juste après le bloc `try-catch`
  - sinon,
    1. la méthode retourne immédiatement
    2. l'exception remonte vers la méthode appelante

# Exception traitée par un bloc `catch`

## Cas particuliers

- Si l'exécution d'un bloc `catch` rencontre un `return`, un `break` ou un `continue`, l'exécution se poursuit à l'endroit du programme correspondant (et pas juste après le bloc `try-catch`)

# Exécution d'un bloc `try` sans erreur ou exception

- Dans les cas où l'exécution des instructions de la clause `try` ne provoque pas d'erreur/exception,
  - le déroulement du bloc de la clause `try` se déroule comme s'il n'y avait pas de bloc `try-catch`
  - le programme se poursuit **après** le bloc `try-catch` (sauf si exécution de `return`, `break`,...)

# Cas des exceptions non traitées

- Si une exception remonte jusqu'à la méthode `main()` sans être traitée par cette méthode,
  - l'exécution du programme est stoppée
  - le message associé à l'exception est affiché, avec une description de la pile des méthodes traversées par l'exception
- A noter : en fait, seul le *thread* qui a généré l'exception non traitée meurt ; les autres *threads* continuent leur exécution

# Exemples de traitements dans un bloc `catch`

- Fixer le problème et réessayer le traitement qui a provoqué le passage au bloc `catch`
- Faire un traitement alternatif
- Retourner (`return`) une valeur particulière
- Sortir de l'application avec `System.exit()`
- Faire un traitement partiel du problème et relancer (`throw`) la même exception (ou une autre exception)

# Souplesse dans le traitement des exceptions

- La méthode dans laquelle l'erreur a eu lieu peut
  - traiter l'anomalie
    - pour reprendre ensuite le déroulement normal du programme,
    - ou pour faire un traitement spécial, différent du traitement normal
  - ne rien faire, et laisser remonter l'exception
  - faire un traitement partiel de l'anomalie, et laisser les méthodes appelantes terminer le traitement

# Pourquoi laisser remonter une exception ?

- Plus une méthode est éloignée de la méthode **main** dans la pile d'exécution, moins elle a une vision globale de l'application
- Une méthode peut donc laisser remonter une exception si elle ne sait pas comment la traiter, en espérant qu'une méthode appelante en saura assez pour la traiter

# Erreur fréquente liée au traitement des exceptions

- Si une variable est
  - déclarée et initialisée **dans** un bloc **try**,
  - et utilisée **après** le bloc **try**,

le compilateur donnera un message d'erreur du type « **undefined variable** » OU « **Variable xx may not have been initialized** »

- Pour éviter cela, déclarer (et initialiser si nécessaire) la variable avant le bloc **try**

# Déclaration incorrecte

```
try {  
    int n;  
    n = ParseInt(args[0]);  
    etagere.add(livres[n]);  
}  
catch(NumberFormatException e) { // Erreur  
    System.err.println("Mauvais numéro de livre " + n);  
    return;  
}  
catch {EtagerePleineException e} {  
    System.err.println("Etagere pleine");  
    return;  
}  
n++; // provoque une erreur à la compilation
```

# Déclaration correcte

```
int n=0; // déclaration/intialisation hors du bloc try
try {
    n = ParseInt(args[0]);
    etagere.add(livres[n]);
}
catch(NumberFormatException e) {
    System.err.println("Mauvais numéro de livre");
    return;
}
catch {EtagerePleineException e) {
    System.err.println("Etagere pleine");
    return;
}
n++; // pas d'erreur à la compilation
```

## Autre erreur à éviter

- Ne jamais attraper une exception en mettant dans le bloc `catch` un affichage de message qui ne donne que peu d'informations sur le problème
- Au moins pendant le développement, le minimum à afficher : ce qu'affiche la méthode `printStackTrace`
- Sinon, on perd des informations précieuses pour la résolution du problème

# Clause **finally**

# Clause `finally`

- La clause `finally` contient un traitement qui sera exécuté **de toute façon**, que la clause `try` ait levé ou non une exception, que cette exception ait été saisie ou non

```
try {  
    . . .  
}  
catch (...) {  
    . . .  
}  
finally {  
    . . .  
}
```

On peut, par exemple,  
fermer un fichier

# Traitement général des exceptions

```
try {  
    . . .  
}  
catch (ClasseException1 e) {  
    . . .  
}  
catch (ClasseException2 e) {  
    . . .  
}  
. . .  
finally {  
    . . .  
}
```

Il est possible d'avoir un ensemble **try - finally** sans clause **catch**

# **finally** est toujours exécuté

- Le bloc **finally** est toujours exécuté (sauf après `system.exit()`), même si le bloc **try** ou le bloc **catch** se termine par une instruction **return** ou lance une exception
- Si le bloc **try** produit une exception
  - si un bloc **catch** attrape cette exception, il est exécuté, et ensuite le bloc **finally** est exécuté
  - sinon, le bloc **finally** est exécuté, et ensuite l'exception se propage à la méthode appelante

# Prédominance du bloc `finally` sur les blocs `try` et `catch`

- Si le bloc `finally` lance une exception, la méthode lance cette exception, quelque soit ce qui s'est passé dans les blocs `try` et `catch`
- Si le bloc `finally` se termine par une instruction `return`, la méthode retourne normalement ; aucune exception n'est levée
- Il faut éviter ces 2 situations car elles nuisent à la lisibilité ; les dernières versions des compilateurs affichent un avertissement

# Prédominance du bloc **finally** sur les blocs **try** et **catch**

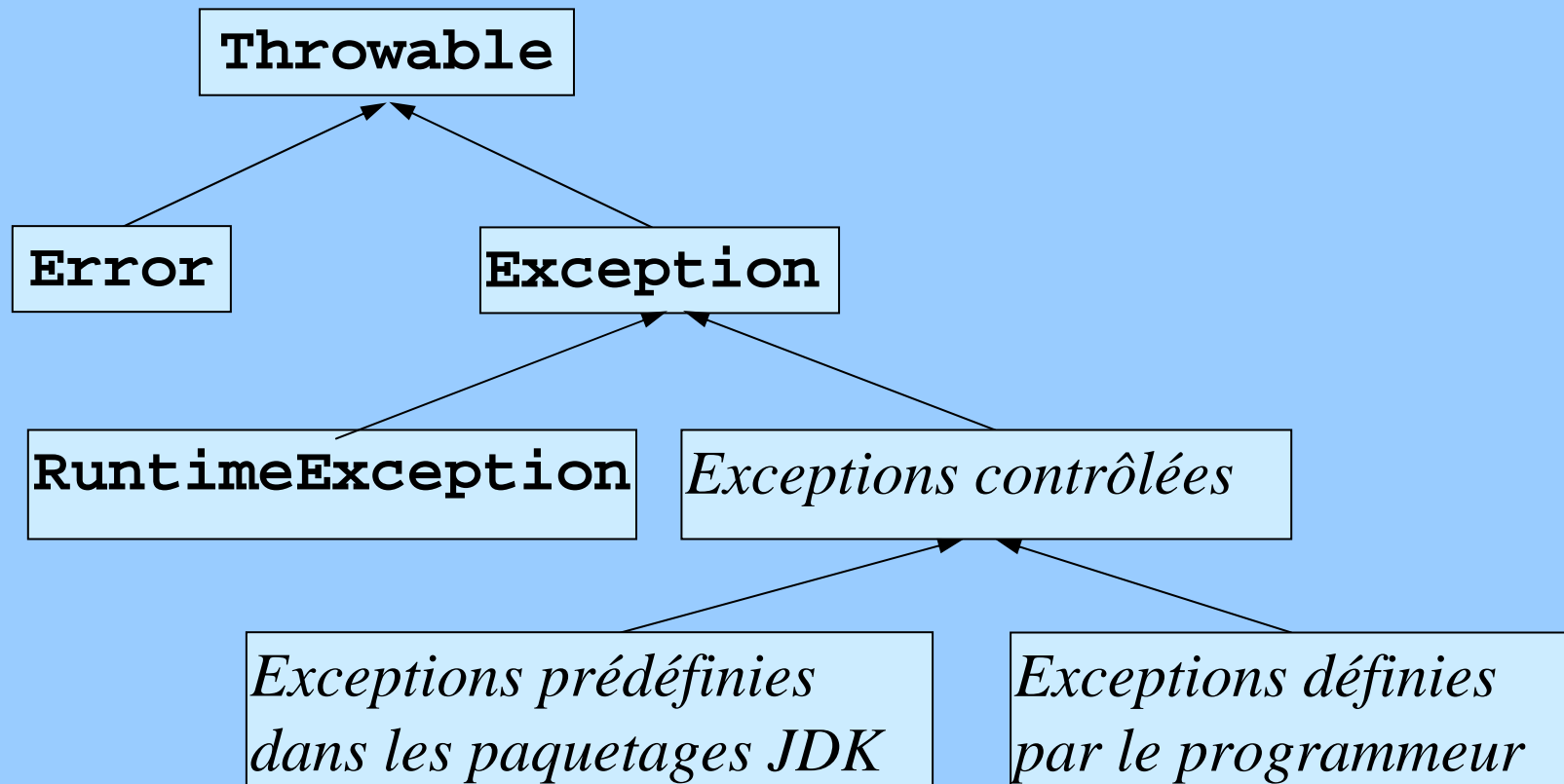
- Si le bloc **finally** se termine par une instruction ordinaire (ne lance pas d'exception et ne se termine pas par un **return**), le bloc **finally** est exécuté, et la méthode se termine (pour l'exception levée et la valeur retournée) comme si le bloc **finally** n'avait pas été exécuté

# Les différents types d'erreurs/ exceptions

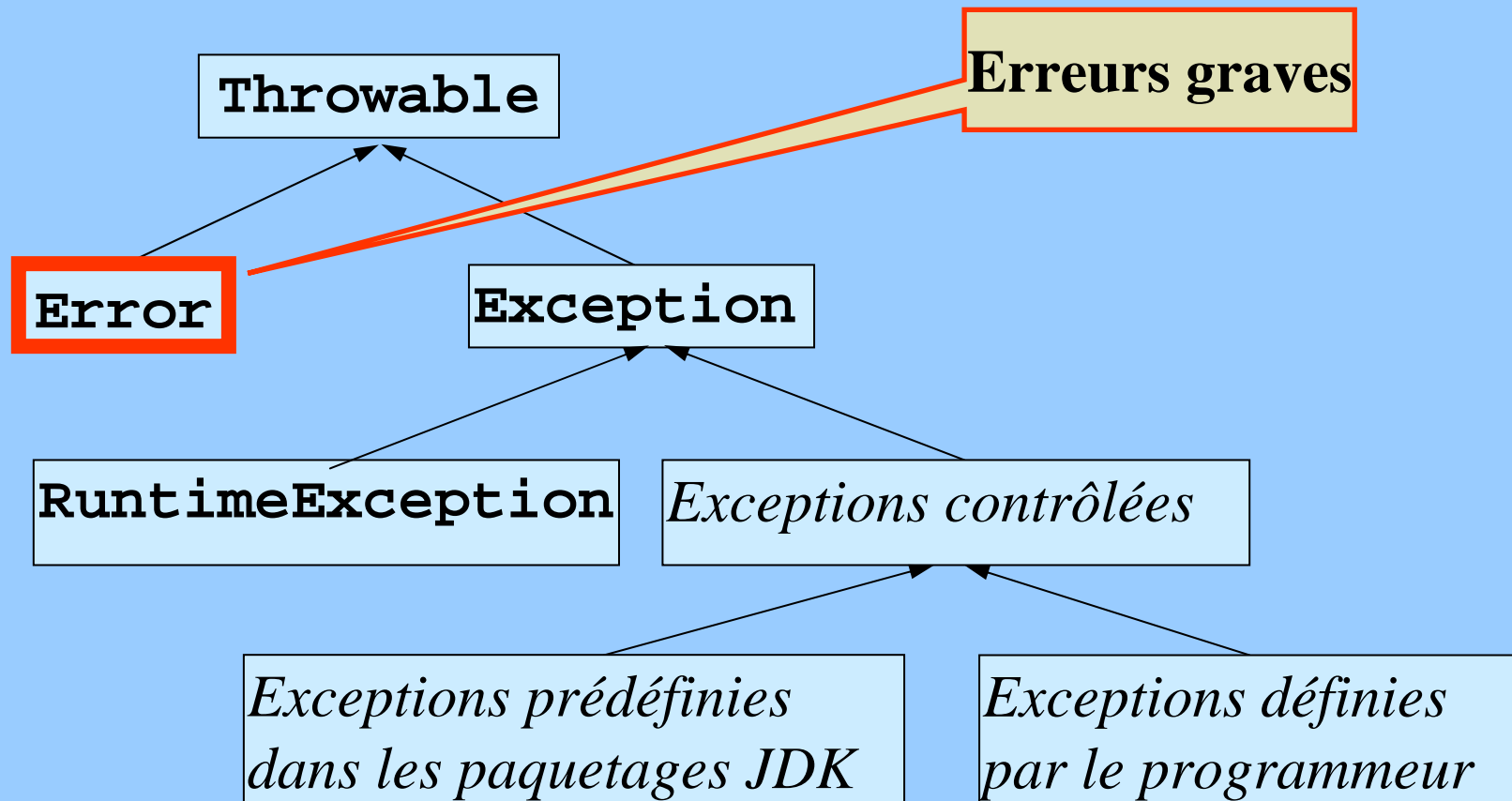
# Les classes d'erreurs/exceptions

- Les classes liées aux erreurs/exceptions sont des classes placées dans l'arborescence d'héritage de la classe `java.lang.Throwable` (classe fille de `Object`)
- Le JDK fournit un certain nombre de telles classes
- Le programmeur peut en ajouter de nouvelles

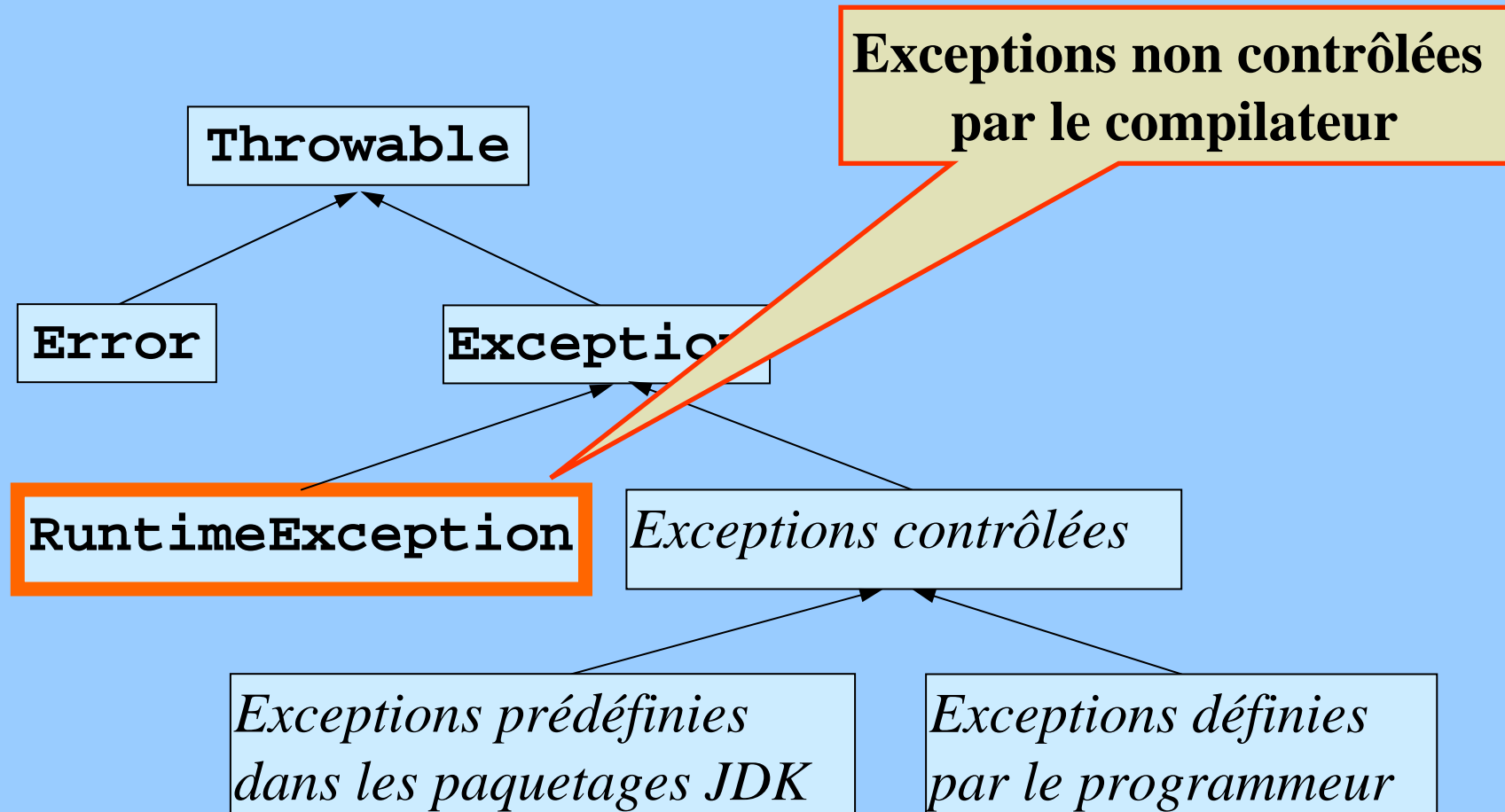
# Arbre d'héritage des exceptions



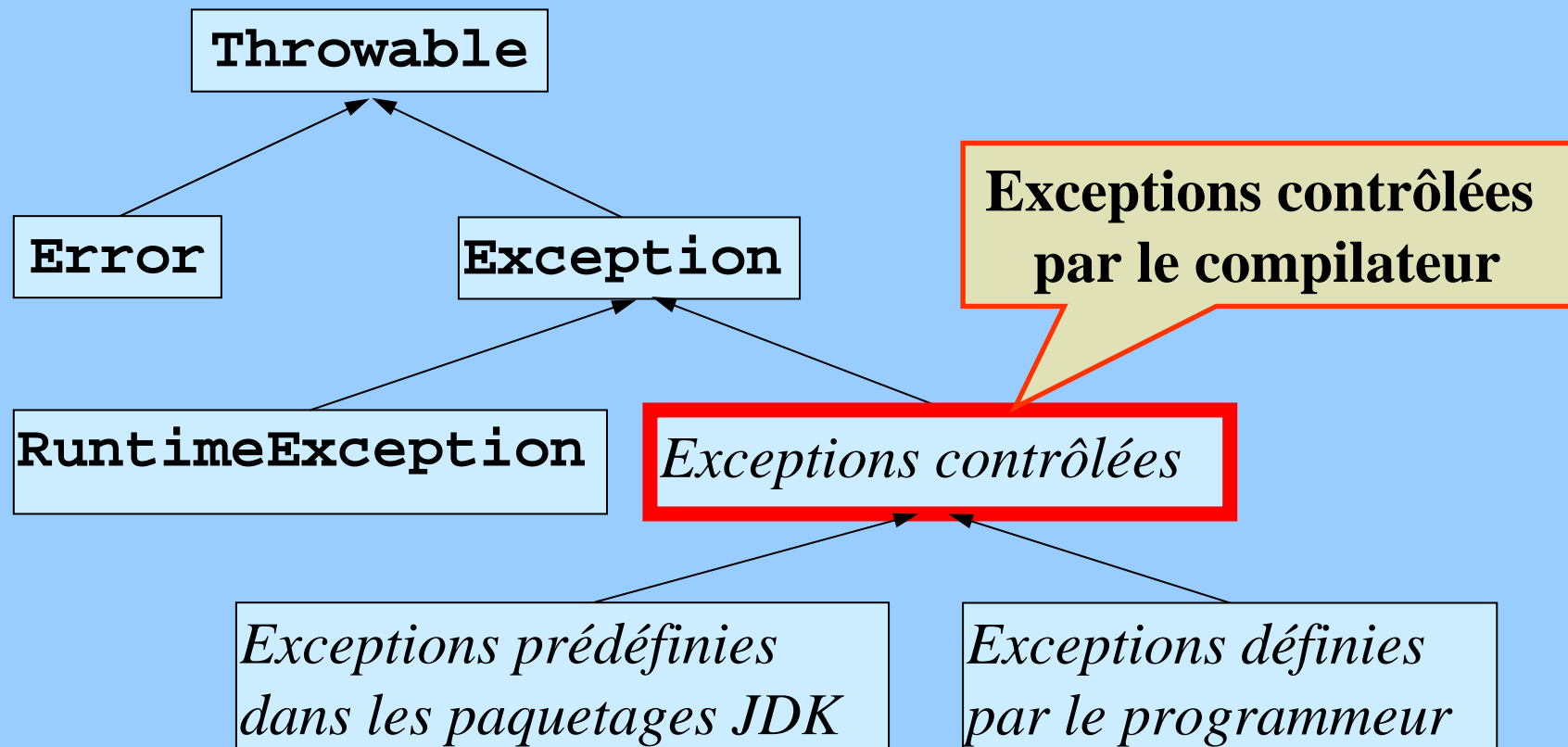
# Arbre d'héritage des exceptions



# Arbre d'héritage des exceptions



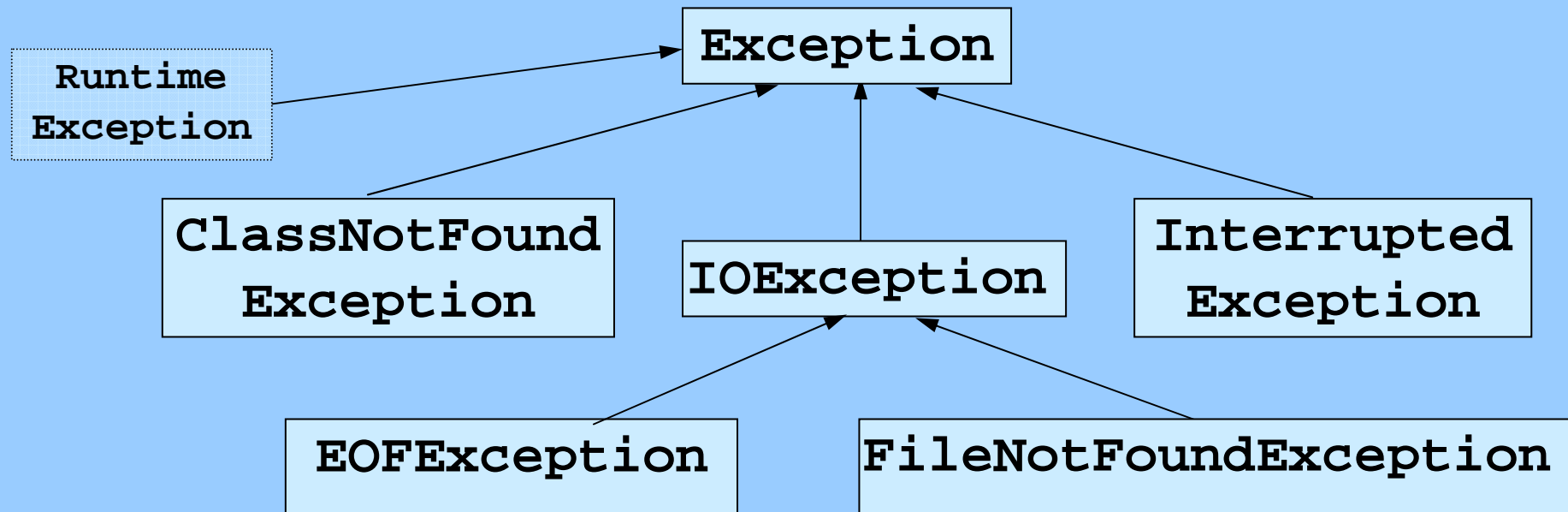
# Arbre d'héritage des exceptions



# Quelques sous-classes de `RuntimeException`

- `NullPointerException`
- `IndexOutOfBoundsException` et sa sous-classe `ArrayIndexOutOfBoundsException`
- `ArithmeticException`
- `IllegalArgumentException` et sa sous-classe `NumberFormatException`
- `ClassCastException`
- `NoSuchElementException`

# Quelques exceptions du JDK, contrôlées par le compilateur



# Exceptions « contrôlées » (1)

- Exceptions qui héritent de la classe **Exception** mais qui ne sont pas des **RuntimeException**
- Le compilateur vérifie que les méthodes utilisent correctement ces exceptions
- Toute méthode qui peut lancer une exception contrôlée, **doit** le déclarer dans sa déclaration

```
int m() throws TrucException {  
    . . .  
}
```

## Exceptions « contrôlées » (2)

- Soit une méthode `m2 ( )` avec une en-tête qui contient « `throws TrucException` »
- Si une méthode `m1 ( )` fait un appel à `m2 ( )`
  - soit `m1 ( )` entoure l'appel de `m2 ( )` avec un bloc « `try - catch(TrucException)` »
  - soit `m1 ( )` déclare qu'elle peut lancer le même type d'exception que `m2 ( )` (ou un type plus large)

## Exceptions « contrôlées » (3)

- L'en-tête d'une méthode `m()` peut très bien contenir « **throws TrucException** » sans que le corps de la méthode ne lance cette exception
- Ce procédé est rarement utilisé mais peut avoir son intérêt si on sait que la méthode risque d'être redéfinie par du code qui renverra cette exception

# Pourquoi des exceptions contrôlées ?

- Pour obliger le développeur à tenir compte des cas d'exception qui ont pu arriver durant l'exécution d'une méthode
- En langage C, les cas particuliers dans le traitement d'une fonction sont souvent indiqués par le fait que la fonction renvoie une valeur particulière (par exemple -1 est renvoyée en cas de problème, alors que les cas « normaux » renvoient une valeur positive)

# Pourquoi des exceptions contrôlées ?

- Le développeur peut oublier ce cas et récupérer la valeur -1 pour la traiter comme toutes les autres valeurs
- De plus les exceptions permettent beaucoup plus de souplesse dans le traitement des cas d'exception que le simple fait de renvoyer une valeur particulière, comme par exemple la possibilité de laisser remonter une exception

# Pourquoi des exceptions non contrôlées ?

- Les exceptions non contrôlées peuvent survenir dans toute portion de code (par exemple **NullPointerException** ou **IllegalArgumentException**)
- Si ces exceptions étaient contrôlées, toutes les méthodes auraient une clause **throws** ou seraient truffées de bloc **try-catch**

# Exceptions et en-têtes de méthodes

- Une méthode peut déclarer pouvoir renvoyer plusieurs types d'exception

- Exemple :

```
public final Object readObject() throws  
IOException, ClassNotFoundException
```

- Les exceptions peuvent être des exceptions contrôlées (le plus souvent) mais aussi des exceptions non contrôlées

# Regrouper des traitements d'erreurs

- L'arbre d'héritage de **Throwable** permet de regrouper les traitements des erreurs liées à toutes les sous-classes d'une classe d'exception
- Par exemple,  
`catch(IOException e) { . . . }`  
permet de regrouper le traitement de toutes les erreurs dues aux entrées/sorties (**EOFException**, **FileNotFoundException**, ...)

# Exception ou traitement normal ?

- Il faut réserver les exceptions aux traitements des erreurs ou des cas exceptionnels et éviter de les utiliser pour traiter un cas « normal » ; ainsi,
  - si un fichier est lu du début à la fin, **ne pas utiliser `EOFException`** pour repérer la fin du fichier ; utiliser plutôt la valeur spéciale renvoyée par la méthode de lecture quand elle rencontre la fin du fichier
  - mais si on rencontre la fin du fichier avant d'avoir lu les 10 valeurs dont on a besoin, utiliser **`EOFException`**

# Exceptions et performances

- Si aucune exception n'est levée, l'impact sur les performances d'un bloc `try-catch` est négligeable
- La levée d'une exception peut au contraire avoir un impact non négligeable sur les performances
- Une autre bonne raison de ne pas utiliser les exceptions pour les cas « normaux »

# Constructeurs des exceptions

- Par convention, toutes les exceptions doivent avoir au moins 2 constructeurs :
  - un sans paramètre
  - un autre dont le paramètre est une chaîne de caractères utilisée pour décrire le problème

# Chaînage de **Throwable**

- Le JDK 1.4 a ajouté des constructeurs à la classe **Throwable** qui prennent un **Throwable** en paramètre, ce qui permet le chaînage d'exceptions (étudié plus loin)
- Ainsi, une API peut transformer une exception de bas niveau en exception de plus haut niveau, tout en gardant la cause première de l'exception (obtenue par `getCause()`)

# Méthodes de la classe **Throwable**

- **getMessage()** retourne le message d'erreur associé à l'instance de **Throwable** (**getLocalizedMessage()** : idem en langue locale)
- **printStackTrace()** affiche sur la sortie standard des erreurs (**System.err**), le message d'erreur et la pile des appels de méthodes qui ont conduit au problème
- **getStackTrace()** récupère les éléments de la pile des appels

# Variante de `printStackTrace`

- La variante sans paramètre affiche la pile sur le fichier des erreurs standard (l'écran par défaut)
- Une variante prend un paramètre de type `PrintWriter` pour choisir un autre flot de sortie
- Si on choisit un flot `StringWriter` décoré par un `PrintWriter`, on peut ainsi enregistrer la pile dans une chaîne de caractères pour, par exemple, la passer à un logger (notion étudiée dans la suite de ce cours)

# Exemple

```
catch (Throwable e ) {  
    StringWriter sw = new StringWriter();  
    PrintWriter pw = new PrintWriter(sw);  
    e.printStackTrace(pw);  
    pw.flush();  
    logger.info(sw.toString());  
    logger.warning("Problème : ... ");  
}
```

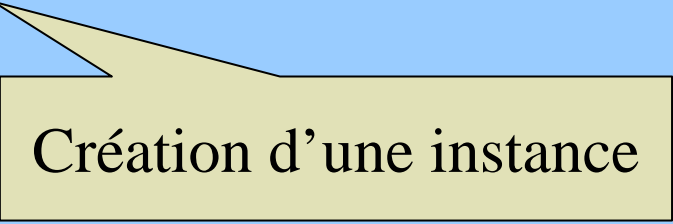
# Lancer une exception avec `throw`

# Lancer une exception

- Le programmeur peut lancer lui-même des exceptions depuis les méthodes qu'il écrit
- Il peut lancer des exceptions des classes d'exceptions fournies par le JDK
- Ou lancer des exceptions appartenant à de nouvelles classes d'exceptions, qu'il a lui-même écrites, et qui sont mieux adaptées aux classes qu'il a écrites

# Lancer une exception du JDK

```
public class Employe {  
    . . .  
    public void setSalaire(double salaire) {  
        if (salaire >= 0)  
            this.salaire = salaire;  
        else  
            throw  
                new IllegalArgumentException("Salaire négatif !");  
    }  
    . . .  
}
```



Création d'une instance

# Créer une nouvelle classe d'exception

# Créer une classe d'exception

- Si on utilise les classes d'exception existantes, l'information transportée par les exceptions peut être insuffisante pour déterminer et donner une solution satisfaisante au problème
- Le programmeur peut avoir à créer ses propres classes d'exception pour s'adapter mieux aux classes de l'application
- Par convention, les noms de classes d'exception se terminent par « Exception »

# Exemple

```
public class EtagerePleineException
    extends Exception {
    private Etagere etagere;
    public EtagerePleineException(Etagere etagere)
    {
        super("Étagère pleine");
        this.etagere = etagere;
    }
    . . .
    public Etagere getEtagere() {
        return etagere;
    }
}
```

L'exception  
contient une  
référence  
à l'étagère pleine

N'oubliez pas d'ajouter au moins  
le constructeur sans paramètre et  
celui qui prend un message en paramètre

# Utiliser la nouvelle classe d'exception

```
public class Etagere {  
    . . .  
    public void ajouteLivre(Livre livre)  
        throws EtagerePleineException {  
        if (nbLivres < livres.length)  
            livres[nbLivres++] = livre;  
        else  
            throw new EtagerePleineException(this);  
    }  
    . . .  
}
```

# Autre façon d'écrire ajouteLivre

```
public void ajouteLivre(Livre livre)
    throws EtagerePleineException {
    try {
        livres[nbLivres] = livre;
        nbLivres++;
    }
    catch(ArrayIndexOutOfBoundsException e) {
        throw new EtagerePleineException(this);
    }
}
```

Attention à ne pas  
incrémenter dans  
cette ligne

Quelle est la meilleure solution ?

# Attraper la nouvelle exception

```
// étagère de 10 livres
Etagere etagere = new Etagere(10);
...
try {
    etagere.ajouteLivre(new Livre("Java", "Eckel"));
}
catch(EtagerePleineException e) {
    System.err.println("L'étagère ne peut contenir que "
        + e.getEtagere().getContenance());
    e.printStackTrace();
}
```

# Compléments sur les exceptions

# Exceptions et redéfinition

- Soit une méthode  $m()$  redéfinie dans une sous-classe  $\mathbf{B}$  de  $\mathbf{A}$
- $m()$  de  $\mathbf{B}$  ne peut déclarer renvoyer (**throws**) que les mêmes exceptions que déclare  $m()$  de  $\mathbf{A}$ , ou des exceptions de sous-classes de ces exceptions
- En effet,  $m()$  de  $\mathbf{B}$  doit pouvoir être utilisée par une méthode qui utilise  $m()$  de  $\mathbf{A}$ , et qui n'attrape que des exceptions lancées par  $m()$  de  $\mathbf{A}$
- Elle peut aussi déclarer lancer moins (ou même pas du tout) d'exceptions

# Exceptions et constructeurs

- Aucune instance n'est créée si une exception est levée par un constructeur

# Chaînage des exceptions

- Depuis le JDK 1.4 on peut chaîner les exceptions
- On peut ainsi lancer une exception de plus haut niveau, sans perdre les informations données par l'exception de plus bas niveau que l'on a saisie
- Le chaînage se fait au moment où on construit l'exception de plus haut niveau : on passe au constructeur l'exception qui suivra dans la chaîne

# Exemple de chaînage

```
try {  
    . . .  
}  
catch(BasNiveauException e) {  
    throw new HautNiveauException(..., e);  
}
```

# Exception avec chaînage

- La classe d'exception de haut niveau doit prévoir un constructeur avec un paramètre de type **Throwable** pour le chaînage de l'exception de plus bas niveau
- Par exemple :

```
HautNiveauException(String message,  
                    Throwable cause) {  
    super(message, cause);  
}
```

Appel du constructeur  
de la classe  
**Exception**

# Récupérer la chaîne des exceptions

- Le code qui attrape l'exception **ex** de plus haut niveau peut obtenir l'exception de départ par l'appel `ex.getCause()` (méthode de la classe **Throwable**)

# Choix des exceptions à lancer

- Le développeur peut hésiter dans le choix du type d'exception que pourra lancer une des méthodes qu'il a écrites
- Voici quelques critères de choix

# Error

- Les **Error** sont réservées aux erreurs qui surviennent dans le fonctionnement de la JVM
- Par exemple, un problème de mémoire **OutOfMemoryError**, une méthode qui n'est pas trouvée **NoSuchMethodError**
- Elles ne devraient jamais arriver
- Elles ne doivent donc être choisies par le développeur que très rarement

# RuntimeException

- C'est le bon choix si le problème ne pourra être résolu (en attrapant l'exception) par une des méthodes appelantes (dans la pile d'exécution)
- Inutile d'alourdir le code des méthodes appelantes en les forçant à attraper une exception (ou à ajouter une clause **throws**) si on sait qu'elles ne pourront résoudre le problème

# RuntimeException

- Elles peuvent être dues à un bug du code (par exemple **ArrayIndexOutOfBoundsException**)

# RuntimeException

- Convient aussi si le problème est dû à une mauvaise utilisation de la méthode
- Exemple : passage d'un paramètre invalide  
**IllegalArgumentException**
- Comme une **Error**, une exception non contrôlée ne devrait jamais arriver ; elle est due à une erreur indépendante du code d'où elle a été lancée (un réseau défaillant, passage d'un mauvais paramètre,...)

# Traitement des exceptions non contrôlées

- Une bonne solution est de laisser remonter l'exception jusqu'à une classe assez proche du début de la pile d'exécution
- Cette classe attrape l'exception, l'enregistre comme événement inattendu (système de *logging*), remet éventuellement les choses en place et stoppe proprement l'action qui a provoqué le problème ; elle prévient l'utilisateur s'il le faut

# Exceptions contrôlées

- Elles sont peu fréquentes mais, au contraire des exceptions non contrôlées, pas inattendues
- Elles correspondent à des scénarios envisagés par le développeur et participent donc à la logique de l'application
- Elles sont réservées aux problèmes qui pourront être résolus (au moins partiellement) par une des méthodes de la pile d'exécution
- Par exemple, si une étagère est pleine, celui qui a voulu ajouter un livre pourra choisir une autre étagère

# Le JDK et les exceptions contrôlées

- Le JDK contient des API qui abusent des exceptions contrôlées
- Elles utilisent ces exceptions alors que la méthode appelante ne pourra pas résoudre le problème
- Par exemple, JDBC (liée au langage SQL) et les entrées-sorties
- En ce cas, une solution est d'attraper l'exception contrôlée et de renvoyer une exception non contrôlée

# Éviter le laisser aller

- Mais attention à ne pas utiliser des exceptions non contrôlées uniquement pour ne pas être gêné dans l'écriture des méthodes qui utilisent la méthode qui peut lancer l'exception (pour éviter d'avoir à écrire des **throws** ou des **try – catch – finally**)

# Exceptions du JDK ou nouveau type d'exception ?

- Si une exception du JDK convient, il vaut mieux l'utiliser car ces exceptions sont bien connues des développeurs
- Sinon, ne pas hésiter à créer un nouveau type d'exception
- Ainsi, s'il n'existe pas d'exception du JDK au bon niveau d'abstraction pour l'application, il vaut mieux créer un nouveau type d'exception

# Messages d'erreur

# Savoir lire un message d'erreur

- Les messages d'erreurs affichés par la JVM correspondent à la sortie de la méthode **`printStackTrace`**
- Lire attentivement ces messages fait gagner beaucoup de temps pour corriger du code

# Savoir lire un message d'erreur

- Le message commence par le message d'erreur associé à l'erreur qui a provoqué l'affichage
- On a ensuite la pile des méthodes traversée depuis la méthode **main** pour arriver à l'erreur
- Les méthodes les plus récemment exécutée sont en premier, la méthode **main** en dernier

# Savoir lire un message d'erreur

- A chaque étape, on a le numéro de la ligne en cause
- A la place du numéro de ligne, on peut avoir le message « **unknown source** » si la JVM Hotspot a pré-compilé le code
- Dans ce dernier cas, on peut faire afficher les numéros de ligne, en compilant la classe qui provoque l'erreur avec l'option **-g** (si on a son fichier source)

# Les assertions

# Points critiques

- Pour vérifier la correction d'un programme il est utile de vérifier certaines assertions à des endroits bien choisis du programme
- La programmation par contrat (*design by contract*) a formalisé l'utilisation des assertions
- Le langage Eiffel en implémente une partie

# Localisation des points critiques

- Les points critiques à vérifier sont
  - le début (pré-condition qui vérifie les paramètres) et la fin (post-condition) des méthodes (contrat client – utilisateur de la méthode)
  - les invariants de classe (dont héritent les classes filles) pour vérifier une propriété de l'état d'une instance
  - les invariants de boucle
  - après un traitement complexe

# Les assertions en Java

- Java n'implémente qu'une très petite partie de la programmation par contrat

# assert

- le JDK 1.4 a introduit un nouveau mot-clé **assert** qui permet d'insérer des assertions qui vont lancer une erreur **AssertionError** (classe du paquetage **java.lang**, fille de **Error**) lorsqu'elles ne seront pas vérifiées
- Ces vérifications peuvent être activées en période de test, et désactivées en production

# Syntaxe

- **`assert assertion;`**  
*assertion* est une expression booléenne
- **`assert assertion : expression;`**  
*expression* peut être n'importe quelle expression ; sa valeur sera affichée si **assert** provoque une erreur

# Exemples (1)

```
■ if (x == 0) {  
    ...  
}  
else {  
    // x doit être égal à 0 ou 1  
    assert x == 1 : x ;  
    ...  
}
```

# Exemples (2)

```
■ switch(x) {  
  case -1: . . .  
  case 0: . . .  
  case 1: . . .  
  case default: assert false : x;  
}
```

# Compilation

- Par défaut **assert** n'est pas reconnu par **javac**
- Ainsi les anciennes classes qui avaient des méthodes **assert** peuvent être compilées sans problème
- `javac -source 1.4 ...`  
reconnaît **assert** comme mot-clé

# Exécution

- Pour activer les assertions (sauf dans les classes système) :

```
java -ea
```

- Pour activer seulement pour un paquetage (et ses sous-paquetages) ou une classe :

```
java -ea<nom-paquetage>...
```

```
java -ea<nom-classe>
```

3 points à taper tels quels !

- Pour désactiver seulement pour un paquetage ou une classe ; par exemple :

```
java -ea -da<nom-paquetage>...
```

# Coût des assertions

- Pour les performances, les assertions désactivées ne coûtent que le test d'un seul drapeau qui indique qu'elles sont désactivées
- Cependant, les assertions prennent de la place dans le code compilé
- Un idiome Java permet de libérer cette place

# « Astuce » pour libérer la place des assertions

```
static final boolean assertionsActives =  
    <true ou false>;  
if (assertionsActives) {  
    assert expression;  
}
```

- Si `assertionsActives` est `false`, le compilateur va s'apercevoir que « `assert expression` » ne peut être atteint et va enlever cette instruction du code compilé

# Limitations de `assert`

- Pas assez puissant pour vérifier simplement une assertion du type «  $\forall i t[i] > 0$  »
- Ne pas utiliser pour les pré-conditions car le `contrat` ne sera pas vérifié quand les assertions seront désactivées en production ; utiliser plutôt les exceptions pour les pré-conditions
- Les assertions de type invariant de classe ne se transmettent pas aux classes filles

# Quand utiliser `assert` ?

- On peut souvent hésiter entre utiliser une assertion ou utiliser une exception
- Le principe général pour choisir est le suivant : les assertions sont faites pour **vérifier la logique interne d'un programme** mais pas pour vérifier une condition qui ne dépend pas complètement du programme
- Ne serait-ce que parce que les assertions peuvent être désactivées

# Quand utiliser `assert` ? Exemples

- Ne pas utiliser une assertion pour vérifier une condition qui ne dépend pas de l'application mais dépend de l'extérieur de l'application
- Ne pas utiliser les assertions pour vérifier une pré-condition d'une méthode `public` (mais on peut le faire pour une méthode `private`)
- Utiliser les assertions pour les post-conditions, les invariants de boucle, de classe, après les traitements complexes

# Journalisation (*logging*)

# Le logging

- On peut vouloir suivre/enregistrer certains événements qui surviennent dans une applications
- Le plus souvent on veut repérer les événements anormaux, signe d'un problème
- Au contraire de **assert**, la journalisation n'interrompt pas le déroulement de l'application

# Paquetage pour le logging

- Depuis le JDK 1.4 le paquetage `java.util.logging` fournit le code pour faire du logging

# Niveaux des messages

- La journalisation permet de fournir des messages de différents niveaux ; par exemple
  - une simple information sur ce qui s'est passé
  - un avertissement
  - un avis de problème important
- On peut choisir à l'exécution le niveau **minimum** des messages qui seront fournis, sans recompilation préalable
- Les messages de niveau inférieur ne seront pas produits

# Messages de logging

- Les messages sont créés par des « loggers », avec la méthode `log` ; par exemple `logger.log(Level.WARNING, "problème Machin", ex);`
- Ce message ne sera produit que si on a fixé à un niveau inférieur ou égal à **WARNING** le niveau minimum des messages que l'on souhaite produire
- Si le niveau a été fixé à **SEVERE** (niveau supérieur) le message ne sera pas produit

# Niveaux des messages

- **SEVERE** (le plus fort) : problème sérieux
- **WARNING** : problème potentiel
- **INFO** : une information
- **CONFIG** : pour informer d'une configuration
- **FINE** : tracer l'exécution
- **FINER** : ... avec plus de détails
- **FINEST** (le moins fort) : encore plus de détails
- **ALL** : ne filtre aucun message
- **OFF** : pas de logging

# Méthodes `log`

- Cette méthode est surchargée par de nombreuses méthodes ; en particulier
  - `log(Level niveau, String message)`
  - `log(Level niveau, String message, Object paramètre)`
  - `log(Level niveau, String message, Object[] paramètres)`
  - `log(LogRecord enregistrement)`
- Les paramètres sont des objets quelconques qui complètent le message

# Méthodes `logp`

- La méthode `log` indique l'emplacement du code qui a généré le message (la classe et la méthode)
- Des optimisations du compilateur peuvent rendre imprécises ces informations
- Les méthodes `logp` permettent de passer en paramètre le nom de la classe et de la méthode

# Méthodes `logrb`

- Les méthodes `logrb` permettent de passer en paramètre le nom d'un « *resource bundle* » pour sortir les messages dans plusieurs langues (voir le cours sur l'internationalisation des applications)

# Méthodes pour les niveaux

- Au lieu de donner le niveau d'un message, on peut utiliser des méthodes spécifiques de la classe `Logger` qui ont le même nom que le niveau auquel elle correspondent (`severe`, `warning`, `info`,...):  
`logger.warning("problème Machin");`

# Fixer le niveau minimum

- 2 façons de fixer le niveau en dessous duquel les messages ne seront pas produits :
  - par programmation, par exemple  
`logger.setLevel(Level.SEVERE);`
  - en modifiant le fichier de propriétés (étudié à la fin de cette section)
- On peut aussi filtrer les messages par un autre critère que le niveau (les filtres sont étudiés plus loin)

# Obtenir un *logger*

- On obtient un *logger* par la méthode **static getLogger** de la classe **Logger**
- On lui passe un nom (ici **fr.truc**) ; on obtient un nouveau logger, ou un logger existant si un logger de ce nom a déjà été créé :

```
Logger.getLogger( "fr.truc" );
```

# Nom d'un logger

- On peut donner n'importe quel nom mais il est recommandé de donner au logger le nom complet de la classe où le *logger* a été créé (ou qu'il suit)
- La configuration du suivi des classes en sera facilitée (étudiée à la fin de cette partie)
- On peut aussi lui donner le nom du paquetage de la classe pour une configuration un peu moins fine

# Hiérarchie des loggers

- Chaque logger a un logger parent
- La hiérarchie correspond au nom du logger
- Par exemple, le parent du logger de nom **fr.truc.machin** s'appelle **fr.truc**
- Le logger « racine » a pour nom la chaîne vide ; il s'obtient par **Logger.getLogger( "" ) ;**

# Exemple

```
package fr.truc;
import java.util.logging.*;
public class Classe {
    private static Logger logger =
        Logger.getLogger("fr.truc");
    public static void main(String[] args){
        logger.fine("Début");
        try { Machin.do(); }
        catch (Error ex){
            logger.log(Level.WARNING,
                "problème Machin", ex);
        }
        logger.fine("Fini !"); } }
```

# Handler

- On peut associer plusieurs handlers à un logger :

```
logger.addHandler(  
    new FileHandler("fichier.log"));
```

- Chaque handler a sa façon de fournir les messages à l'extérieur de l'application :
  - sur l'écran (**ConsoleHandler**)
  - dans un fichier (**FileHandler**)
  - dans un flot (**StreamHandler**)
  - dans un socket (**SocketHandler**)
  - en mémoire (**MemoryHandler**)

# MemoryHandler

- Ne fait qu'enregistrer les messages en mémoire
- On peut indiquer à ce type de handler de passer les messages (**push**) emmagasinés à un autre handler, par exemple un **FileHandler**, désigné au moment de la création du handler
- Les messages emmagasinés ne seront transmis que si une condition survient
- La condition peut être l'arrivée d'un message d'un niveau minimum (**setPushLevel**)

# FileHandler (1)

- Sans doute le plus utilisé ; enregistre les messages dans des fichiers
- Il est possible d'indiquer un, ou plusieurs fichiers qui seront utilisés circulairement
- Par défaut le niveau est **Level.ALL** ; il suffit donc de fixer le niveau du logger pour que les messages de logging soient enregistrés
- Voir la javadoc pour plus d'informations

# FileHandler – nom des fichiers

- Les noms et emplacement des fichiers peuvent être donnés sous la forme de patterns contenant par exemple « %h » pour désigner le répertoire HOME de l'utilisateur
- Par défaut, le pattern est « %h/java%u.log » ; %u représente un nombre entier (normalement 0) qui peut être augmenté si un fichier log de même nom est déjà utilisé

## FileHandler (2)

- La configuration d'un tel handler comporte de nombreuses propriétés ;

# Niveau des Handlers

- Attention, certains handlers n'affiche pas par défaut tous les niveaux de gravité de messages
- Par exemple, `ConsoleHandler` n'affiche que les messages de niveau `INFO` et supérieur
- Si on veut faire sortir un niveau inférieur, il faut non seulement fixer le niveau du logger, mais aussi fixer le niveau du handler (voir exemple du transparent suivant)

# Exemple

```
// Pour enlever la console par défaut
LogManager.getLogManager().reset();
ConsoleHandler ch = new ConsoleHandler();
ch.setLevel(Level.FINE);
Logger logger = Logger.getLogger("truc");
logger.addHandler(ch);
logger.setLevel(Level.FINE);
```

# Format de sortie

- Par défaut les messages sont sortis en format texte sur l'écran et en format XML dans un fichier
- Il est possible de construire son propre format de sortie

# Classe `Formatter`

- Chaque handler a un formateur qui met en forme le message produit ; on le modifie par `handler.setFormatter(f1)` ;
- Un formateur est une instance d'une classe qui hérite de la classe `Formatter` ; on doit redéfinir la méthode `String format(LogRecord record)`

# Configuration du *logging*

- 2 façons de configurer le *logging* :
  - par un fichier de configuration :

```
java -D  
java.util.logging.config.file=truc.conf  
...
```
  - par programmation avec des méthodes diverses de la classe `Logger` (`addHandler`, `setFilter`, `setLevel`,...)
- Le fichier de configuration par défaut se trouve sous le répertoire d'installation de jre :  
`lib/logging.properties`

# Cheminement du **LogRecord** (1)

- Lorsqu'un message est produit, il est enregistré dans une instance de **LogRecord**
- Cet enregistrement est alors passé à tous les handlers du logger

## Cheminement du `LogRecord` (2)

- Il remonte ensuite au logger parent qui le passe à ses handlers, et ainsi de suite...
- L'appel de `setUseParentHandlers(false)` de la classe `Logger` permet de stopper cette remontée si on le souhaite
- Rappel : le parent du logger de nom `fr.unice.machin` a pour nom `fr.unice` ; il est donc simple d'indiquer les classes qui seront suivies en fixant le niveau d'un logger (voir transparent suivant)

# Configuration des niveaux

- Si on ne fixe pas le niveau d'un logger, il hérite celui de son parent
- Si on a pris soin de nommer les loggers suivant le nom des classes qu'ils observent, il est facile de choisir le niveau du suivi des classes ou paquetages que l'on souhaite journaliser

# Configuration par défaut

- Le logger racine est configuré avec un **ConsoleHandler**
- Dès que l'on crée un autre logger, les messages sont donc envoyés vers l'écran (**System.err** plus exactement) à cause du cheminement des **LogRecord** vers les parents
- Le niveau par défaut est **INFO**
- Les **File** et **Socket Handler** ont un **XMLFormatter**

# Configuration par défaut (1)

```
# Par défaut configure seulement un
# ConsoleHandler avec le niveau INFO
handlers= java.util.logging.ConsoleHandler
# Pour ajouter aussi un FileHandler :
#handlers=java.util.logging.FileHandler,java
    .util.logging.ConsoleHandler
# Niveau global par défaut.
# Le ConsoleHandler a un niveau séparé.
.level= INFO
```

# Configuration par défaut (2)

```
# Fichier de sortie dans HOME.  
java.util.logging.FileHandler.pattern =  
    %h/java%u.log  
# Pour des fichiers de logging à écriture  
# circulaire (2 fichiers de 50000 octets)  
java.util.logging.FileHandler.limit = 50000  
java.util.logging.FileHandler.count = 2  
  
java.util.logging.FileHandler.formatter =  
    java.util.logging.XMLFormatter
```

# Configuration par défaut (3)

```
# Limiter les messages affichés à INFO.  
java.util.logging.ConsoleHandler.level =  
    INFO  
java.util.logging.ConsoleHandler.formatter =  
    java.util.logging.SimpleFormatter
```

# Configuration par défaut (4)

```
# On peut configurer chaque logger.  
# Par exemple, indiquer que le logger  
# fr.truc ne prend en compte que les  
# messages dont le niveau est au-dessus  
# de SEVERE :  
fr.truc.level = SEVERE
```

# Filtres

- Si on veut filtrer les messages avec d'autres attributs que le niveau, on peut filtrer les **LogRecord** avec un filtre attaché à un handler
- Le filtre doit être une instance d'une classe qui implémente l'interface **Filter**
- On associe un filtre à un logger, ou à un handler avec les 2 méthodes de même signature des classes **Logger** et **Handler**  
**setFilter(Filter filtre)**  
**throws SecurityException**

# Filtrage

- La méthode (de **Filter**) **boolean isLoggable(LogRecord enregistrement)** indique si le handler traitera l'enregistrement ou l'ignorera
- Le filtrage peut être effectué sur tout attribut du **LogRecord**
- Par exemple, sur les paramètres du message (passés en paramètre de **log**)
- On utilise pour cela les nombreux accesseurs de la classe **LogRecord**

# Chaîne complète de traitement

- Si le niveau est suffisant, une instance de **LogRecord** est créée par le logger
- Celui-ci repasse l'enregistrement à ses handlers pour fournir le message
- Si l'enregistrement passe le filtre éventuel,
  - le formatter de chaque handler met en forme l'enregistrement sous forme d'un message de journalisation
  - l'enregistrement est ensuite passé au parent du logger

# Rapidement, comment faire ?

- Pour chaque classe que l'on veut suivre,
- `import java.util.logging.*;`
- `private static Logger logger =  
    Logger.getLogger("fr.p1.Classe");`
- Dans les méthodes suivies :  
`logger.info("Message d'information");`  
`logger.warning("Avertissement");`
- Positionner le niveau de sortie des messages par programmation ou dans un fichier de propriétés

# Enregistrer une exception

```
StringWriter sw = new StringWriter();  
PrintWriter pw = new PrintWriter(sw, true);  
exception.printStackTrace(pw);  
pw.flush();  
sw.flush();  
logger.severe(sw.toString());
```