Séance 1 Java les base de java :

Le bloc d'initialisation en Java

Le bloc d'initialisation contient le code qui est toujours exécuté chaque fois qu'une instance est créée. Il est utilisé pour déclarer / initialiser la partie commune des différents constructeurs d'une classe. Par exemple,

```
import java.io.*;
public class GFG
{
    // Initializer block starts..
    {
        // This code is executed before every constructor.
        System.out.println("Common part of constructors invoked !!");
    }
    // Initializer block ends

    public GFG()
    {
        System.out.println("Default Constructor invoked");
    }
    public GFG(int x)
    {
        System.out.println("Parametrized constructor invoked");
    }
    public static void main(String arr[])
    {
        GFG obj1, obj2;
        obj1 = new GFG();
        obj2 = new GFG(0);
    }
}
```

Production:

```
Partie commune des constructeurs invoquée !!

Constructeur par défaut appelé

Partie commune des constructeurs invoquée !!

Constructeur paramétré appelé
```

Nous pouvons noter que le contenu du bloc d'initialisation est exécuté chaque fois qu'un constructeur est invoqué (avant le contenu du constructeur)

Exceptions Java

Lors de l'exécution de code Java, différentes erreurs peuvent survenir: erreurs de codage faites par le programmeur, erreurs dues à une mauvaise saisie ou autres choses imprévisibles.

Lorsqu'une erreur se produit, Java s'arrête normalement et génère un message d'erreur. Le terme technique pour cela est: Java lancera une **exception** (lancera une erreur).

Java essaie et attrape

L' `try`instruction vous permet de définir un bloc de code à tester pour les erreurs lors de son exécution.

L' `catch`instruction vous permet de définir un bloc de code à exécuter, si une erreur se produit dans le bloc try.

Les mots `try`- `catch`clés et viennent par paires:

Syntaxe

```
try {
  //  Block of code to try
}
catch(Exception e) {
  //  Block of code to handle errors
}
```

Prenons l'exemple suivant:

Cela générera une erreur, car **myNumbers [10]** n'existe pas.

```
public class MyClass {
  public static void main(String[ ] args) {
    int[] myNumbers = {1, 2, 3};
```

```
    System.out.println(myNumbers[10]); // error!
  }
}
```

La sortie sera quelque chose comme ceci:

```
Exception in thread "main"
java.lang.ArrayIndexOutOfBoundsException: 10
        at MyClass.main(MyClass.java:4)
```

Exemple

```
public class MyClass {
  public static void main(String[ ] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    }
  }
}
```

finally

L' `finally`instruction vous permet d'exécuter du code, après `try...catch`, quel que soit le résultat:

Exemple

```
public class MyClass {
  public static void main(String[] args) {
    try {
      int[] myNumbers = {1, 2, 3};
      System.out.println(myNumbers[10]);
```

```
    } catch (Exception e) {
      System.out.println("Something went wrong.");
    } finally {
      System.out.println("The 'try catch' is finished.");
    }
  }
}
```

La sortie sera:

```
Something went wrong.
The 'try catch' is finished.
```

Classes et méthodes abstraites

L' **abstraction des** données consiste à masquer certains détails et à ne montrer que les informations essentielles à l'utilisateur.
L'abstraction peut être réalisée avec **des classes abstraites** ou des **interfaces** (sur lesquelles vous en apprendrez plus dans le chapitre suivant).

Le `abstract`mot-clé est un modificateur sans accès, utilisé pour les classes et les méthodes:

- **Classe abstraite:** est une classe restreinte qui ne peut pas être utilisée pour créer des objets (pour y accéder, elle doit être héritée d'une autre classe).

- **Méthode abstraite:** ne peut être utilisée que dans une classe abstraite, et elle n'a pas de corps. Le corps est fourni par la sous-classe (héritée de).

Une classe abstraite peut avoir à la fois des méthodes abstraites et régulières:

```
abstract class Animal {

  public abstract void animalSound();

  public void sleep() {

    System.out.println("Zzz");

  }

}
```

À partir de l'exemple ci-dessus, il n'est pas possible de créer un objet de la classe Animal:

```
Animal myObj = new Animal(); // will generate an error
```

Pour accéder à la classe abstraite, elle doit être héritée d'une autre classe.

Exemple

```
// Abstract class

abstract class Animal {

  // Abstract method (does not have a body)

  public abstract void animalSound();

  // Regular method

  public void sleep() {

    System.out.println("Zzz");

  }

}


// Subclass (inherit from Animal)

class Pig extends Animal {

  public void animalSound() {

    // The body of animalSound() is provided here
```

```
    System.out.println("The pig says: wee wee");
  }
}

class MyMainClass {
  public static void main(String[] args) {
    Pig myPig = new Pig(); // Create a Pig object
    myPig.animalSound();
    myPig.sleep();
  }
}
```

Introduction aux chargeurs de classe ClassLoader

Les chargeurs de classes sont responsables du **chargement dynamique des classes Java pendant l'exécution sur la JVM** (Java Virtual Machine). En outre, ils font partie du JRE (Java Runtime Environment). Par conséquent, la JVM n'a pas besoin de connaître les fichiers ou les systèmes de fichiers sous-jacents pour exécuter des programmes Java grâce aux chargeurs de classe.

De plus, ces classes Java ne sont pas chargées en mémoire en même temps, mais lorsqu'elles sont requises par une application. C'est là que les chargeurs de classe entrent en scène. Ils sont responsables du chargement des classes en mémoire.

Dans ce didacticiel, nous allons parler de différents types de chargeurs de classe intégrés, de leur fonctionnement et d'une introduction à notre propre implémentation personnalisée.

Types de chargeurs de classe intégrés

Commençons par apprendre comment différentes classes sont chargées à l'aide de divers chargeurs de classes à l'aide d'un exemple simple:

```
public void printClassLoaders() throws ClassNotFoundException {

System.out.println("Classloader of this class:"

+ PrintClassLoader.class.getClassLoader());

System.out.println("Classloader of Logging:"

+ Logging.class.getClassLoader());

System.out.println("Classloader of ArrayList:"

+ ArrayList.class.getClassLoader());

}
```

Lorsqu'elle est exécutée, la méthode ci-dessus imprime:

```
Class loader of this class:sun.misc.Launcher$AppClassLoader@18b4aac2

Class loader of Logging:sun.misc.Launcher$ExtClassLoader@3caeaf62

Class loader of ArrayList:null
```

Comme nous pouvons le voir, il existe trois chargeurs de classe différents ici; application, extension et bootstrap (affichés comme *null*).

Le chargeur de classe d'application charge la classe dans laquelle la méthode d'exemple est contenue. **Une application ou un chargeur de classe système charge nos propres fichiers dans le chemin de classe.**

Ensuite, l'extension charge la classe *Logging* . **Les chargeurs de classes d'extension chargent des classes qui sont une extension des classes Java de base standard.**

Enfin, le bootstrap charge la classe *ArrayList* . **Un bootstrap ou chargeur de classe primordial est le parent de tous les autres.**

Cependant, nous pouvons voir que la dernière sortie, pour la *ArrayList* , affiche *null* dans la sortie. **En effet, le chargeur de classe d'amorçage est écrit en code natif, pas en Java - il n'apparaît donc pas comme une classe Java.** Pour cette raison, le comportement du chargeur de classe d'amorçage diffère selon les JVM.

Parlons maintenant plus en détail de chacun de ces chargeurs de classe.

Chargeur de classe Bootstrap

Les classes Java sont chargées par une instance de *java.lang.ClassLoader* . Cependant, les chargeurs de classes sont eux-mêmes des classes. Par conséquent, la question est de savoir qui charge le *java.lang.ClassLoader* lui - même *?*

C'est là que le bootstrap ou le chargeur de classe primordial entre en scène.

Il est principalement responsable du chargement des classes internes JDK, généralement *rt.jar* et d'autres bibliothèques de base situées dans le *répertoire $ JAVA_HOME / jre / lib* . En outre, le **chargeur de classe Bootstrap sert de parent de toutes les autres instances de *ClassLoader*** .

Ce chargeur de classe bootstrap fait partie de la JVM principale et est écrit en code natif comme indiqué dans l'exemple ci-dessus. Différentes plates-formes peuvent avoir différentes implémentations de ce chargeur de classe particulier.

2.2. Chargeur de classe d'extension

Le **chargeur de classe d'extension est un enfant du chargeur de classe bootstrap et prend en charge le chargement des extensions des classes Java standard de base** afin qu'il soit disponible pour toutes les applications exécutées sur la plate-forme.

Le chargeur de classe d'extension se charge à partir du répertoire des extensions JDK, généralement le répertoire *$ JAVA_HOME / lib / ext* ou tout autre répertoire mentionné dans la propriété système *java.ext.dirs* .

2.3. Chargeur de classe système

Le chargeur de classe système ou d'application, quant à lui, se charge de charger toutes les classes de niveau application dans la JVM. **Il charge les fichiers trouvés dans la variable d'environnement CLASSPATH, *-classpath* ou *-cp* option de ligne de commande** . En outre, c'est un enfant du chargeur de classe Extensions.

3. Comment fonctionnent les chargeurs de classe?

Les chargeurs de classes font partie de l'environnement d'exécution Java. Lorsque la machine virtuelle Java demande une classe, le chargeur de classe essaie de localiser la classe et de charger la définition de classe dans l'environnement d'exécution à l'aide du nom de classe qualifié complet.

La **méthode *java.lang.ClassLoader.loadClass ()* est responsable du chargement de la définition de classe dans le runtime** . Il essaie de charger la classe en fonction d'un nom complet.

Si la classe n'est pas déjà chargée, elle délègue la demande au chargeur de classe parent. Ce processus se produit de manière récursive.

Finalement, si le chargeur de classe parent ne trouve pas la classe, la classe enfant appellera la méthode *java.net.URLClassLoader.findClass ()* pour rechercher des classes dans le système de fichiers lui-même.

Si le dernier chargeur de classe enfant ne parvient pas non plus à charger la classe, il lance *java.lang.NoClassDefFoundError* ou *java.lang.ClassNotFoundException.*

Examinons un exemple de sortie lorsque ClassNotFoundException est levée.

```
java.lang.ClassNotFoundException: com.baeldung.classloader.SampleClassLoader

at java.net.URLClassLoader.findClass(URLClassLoader.java:381)

at java.lang.ClassLoader.loadClass(ClassLoader.java:424)

at java.lang.ClassLoader.loadClass(ClassLoader.java:357)

at java.lang.Class.forName0(Native Method)

at java.lang.Class.forName(Class.java:348)
```

Si nous parcourons la séquence d'événements dès l'appel de *java.lang.Class.forName ()* , nous pouvons comprendre qu'il essaie d'abord de charger la classe via le chargeur de classe parent, puis *java.net.URLClassLoader.findClass ()* pour rechercher la classe elle-même.

Lorsqu'il ne trouve toujours pas la classe, il lève une *exception ClassNotFoundException.*

Il existe trois caractéristiques importantes des chargeurs de classe.

3.1. Modèle de délégation

Les chargeurs de classe suivent le modèle de délégation où, **sur demande pour trouver une classe ou une ressource, une instance *ClassLoader* délègue la recherche de la classe ou de la ressource au chargeur de classe parent** .

Disons que nous avons une requête pour charger une classe d'application dans la JVM. Le chargeur de classe système délègue d'abord le chargement de cette classe à son chargeur de classe d'extension parent qui à son tour la délègue au chargeur de classe d'amorçage.

Ce n'est que si le bootstrap puis le chargeur de classe d'extension ne parviennent pas à charger la classe, le chargeur de classe système essaie de charger la classe elle-même.

3.2. Classes uniques

En raison du modèle de délégation, il est facile de garantir **des classes uniques car nous essayons toujours de déléguer vers le haut** .

Si le chargeur de classe parent ne parvient pas à trouver la classe, alors seulement l'instance actuelle tentera de le faire elle-même.

3.3. Visibilité

De plus, **les chargeurs de classes enfants sont visibles pour les classes chargées par ses chargeurs de classes parents** .

Par exemple, les classes chargées par le chargeur de classe système ont une visibilité sur les classes chargées par l'extension et les chargeurs de classe Bootstrap, mais pas l'inverse.

Pour illustrer cela, si la classe A est chargée par un chargeur de classe d'application et que la classe B est chargée par le chargeur de classe d'extensions, les classes A et B sont alors visibles en ce qui concerne les autres classes chargées par le chargeur de classe d'application.

La classe B, néanmoins, est la seule classe visible en ce qui concerne les autres classes chargées par le chargeur de classe d'extension.

4. ClassLoader personnalisé

Le chargeur de classe intégré suffirait dans la plupart des cas où les fichiers sont déjà dans le système de fichiers.

Cependant, dans les scénarios où nous devons charger des classes à partir du disque dur local ou d'un réseau, nous pouvons avoir besoin d'utiliser des chargeurs de classe personnalisés.

Dans cette section, nous aborderons d'autres cas d'utilisation des chargeurs de classe personnalisés et nous montrerons comment en créer un.