Class loaders in Java

- Class loaders in Java
  - The classloader subsystem
    - Delegation hierarchy principle
    - Visibility principle
    - Uniqueness principle
  - `java.lang.ClassLoader` class
    - Source code
    - `loadClass(String name)`
    - `findClass(String name)`
    - `getResource(String name)`
    - `setDefaultAssertionStatus(boolean enabled)`
    - ClassLoader code example
  - The class loader hierarchy
    - System/Application class loader

The classloader subsystem

The classloader subsystem is an abstract class (`java.lang.class`) and is used for **loading, linking, and initialization** of the `.class` files(byte codes) into the *JVM Memory(Method area)* subsystem at **run-time**.

Java ClassLoader is followed by three basic principles:

Delegation hierarchy principle

Class loaders follow the delegation model where, a **ClassLoader instance will delegate the search of the class or resource to the parent class loader, before trying to find the class itself**.

Let's say we have a request to load an application class into the JVM. The system *application* class loader will first delegates the loading of that class to its parent (*extension*) which, in turn, will delegate to the *bootstrap* class loader.

For instance, in an application server, different applications may need different versions of the same class. The class loader delegation model makes it possible to meet these requirements without causing conflicts.

Visibility principle

Class loaders in Java can have varying levels of visibility, which determines their ability to find and load classes from other class loaders. Java uses the following.

Parent-first visibility: The parent class loader is used first to load a class. If it cannot find the class, the child class loader is consulted.

Classes from the Application Classloader - those defined by the user - could use classes from the Extension or Bootstrap Classloader. But not the other way around.

Similarly, classes from the Extension Classloader use classes from the Bootstrap Classloader. Classes from Bootstrap can't use classes from the extension - if they did, they would be included in the Bootstrap. Classes from the Application Classloader didn't even exist at the time the classes in the Bootstrap were written.

The level of visibility depends on the class loader hierarchy and the classpath, and it can have significant implications for application behavior. It's important to consider the visibility model used in an application to ensure that classes are loaded correctly and that classloading conflicts are avoided.

Uniqueness principle

Java class loaders **keep different versions of the same class in separate namespaces**, which allows for creating multiple instances of a class with different versions. This is **useful for web applications that need to load shared libraries without conflicts**.

However, this feature can cause issues if not used carefully. If a class is loaded by two different class loaders, the **JVM will treat them as separate classes, and objects created from them will not be interchangeable**. This can lead to unexpected behavior if these objects are passed between methods expecting objects created by different class loaders.

To avoid these issues, **it is recommended to use a single class loader to load classes whenever possible. When multiple class loaders are used, take extra care to ensure that objects are not passed between classes with different namespaces**.

`java.lang.ClassLoader` class

Source code

```
private final ClassLoader parent;

protected Class<?> loadClass(String name)
        throws ClassNotFoundException
    {
        synchronized (getClassLoadingLock(name)) {
```

```
            // First, check if the class has already been loaded
            Class<?> c = findLoadedClass(name);
            if (c == null) {
                long t0 = System.nanoTime();
                try {
                    if (parent != null) {
                        c = parent.loadClass(name, false); //the final attribute
                    } else {
                        c = findBootstrapClassOrNull(name);
                    }
                } catch (ClassNotFoundException e) {
                    // ClassNotFoundException thrown if class not found
                    // from the non-null parent class loader
                }

                if (c == null) {
                    // If still not found, then invoke findClass in order
                    // to find the class.
                    c = findClass(name);
                }
            }
            return c;
        }
    }
```

`loadClass(String name)`

Loads a class with the specified name. It first checks if the class has already been loaded, and if not, it delegates the loading of the class to the parent class loader.

`findClass(String name)`

Finds the class with the name you've specified. It is called by the `loadClass()` method if the parent class loader cannot find the class.

`getParent()`

Returns a class loader's parent class loader.

`getResource(String name)`

Finds the resource with the name you have specified. It searches the classpath for the resource and returns a URL object that can be used to access the resource.

setDefaultAssertionStatus(boolean enabled)

Enables or disables assertions for this class loader and all classes loaded by it.

ClassLoader code example

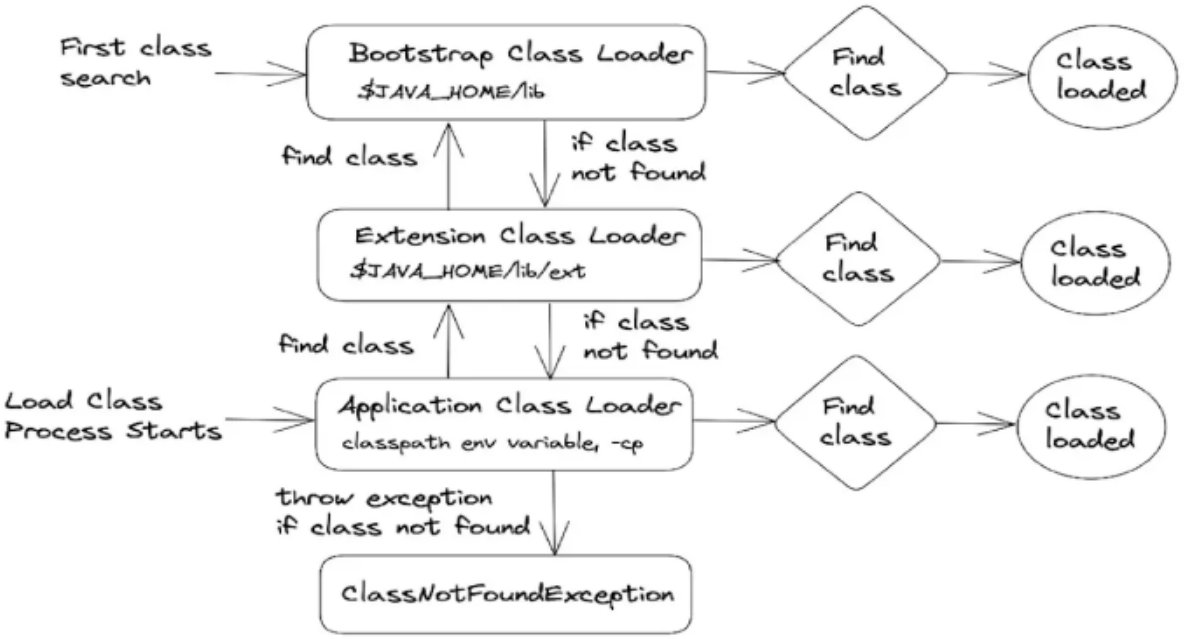
This test shows the different class loaders being used for different classes

```
@Test
public void testClassLoaders() throws ClassNotFoundException {
    ClassLoaderPrinter classLoaderPrinter = new ClassLoaderPrinter();
    classLoaderPrinter.printClassLoaders();
}

private class ClassLoaderPrinter {

    private Logger LOGGER = LoggerFactory.getLogger(ClassLoaderPrinter.class);

    public void printClassLoaders() throws ClassNotFoundException {

        LOGGER.info("Classloader of this class:"
                + ClassLoaderPrinter.class.getClassLoader());

        LOGGER.info("Classloader of Logging:"
                + Logging.class.getClassLoader());

        LOGGER.info("Classloader of ArrayList:"
                + ArrayList.class.getClassLoader());
    }
}
```

This will produce the output

```
Classloader of this class:sun.misc.Launcher$AppClassLoader@18b4aac2
Classloader of Logging:sun.misc.Launcher$ExtClassLoader@7adf9f5f
Classloader of ArrayList:null (Bootstrap classloader - null parent)
```

The class loader hierarchy



System/Application class loader

Loads classes from the application's classpath (JAR, WAR etc...).

The application class loader is a standard Java class that loads classes from the directories and JAR files listed in the `CLASSPATH` environment variable or the -classpath command-line option. It loads the first class it finds if there are multiple versions.

The application class loader is the last class loader to search for a class. **If it can't find it, the JVM throws a `ClassNotFoundException`**. This class loader can also delegate class loading to its parent class loader, the **extension class loader**.

Aside from loading classes from the classpath, the application class loader also loads **classes generated at runtime**, like those created by the Java Reflection API or third-party libraries that use bytecode generation.

The `ClassNotFoundExcpetion` class

By looking at the stacktrace of the exception we can see what class loaders have been involved in the class search

```
java.lang.ClassNotFoundException: com.myapp.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)
```

Extension class loader

Instead of the extension class loader, Java 9 and later versions use the `java.lang.ModuleLayer` class to **load modules from the extension directory**. The extension directory is now treated as a separate layer in the module system, and modules in the extension directory are loaded by the extension layer's class loader. Note that in Java 9 and later versions, it is recommended to use modules instead of the extension mechanism to share code between applications. The extension mechanism is still available for backward compatibility with older applications, but it is not recommended for new code.

Bootstrap class loader

Also known as the *primordial* class loader, this is **the class loader where the search starts**. The bootstrap class loader is responsible for loading **core Java classes** such as `java.lang.Object` and `java.lang.String`. It is **implemented in native code** (C/C++) and classes are located in the `$JAVA_HOME/lib` directory.

Create a custom class loader

In some special needs we would want to create a custom class loader. For example:

- if we need to **load class definitions over a network channel** (as internet browsers do)
- Creating classes dynamically suited to the user's needs, e.g. in JDBC, **switching between different driver implementations is done through dynamic class loading**.
- Implementing a **class versioning mechanism while loading different bytecodes for classes with the same names and packages**. This can be done either through a URL class loader (load jars via URLs) or custom class loaders

Implementation

The custom class loader shouldn't override the `loadClass` method because that would alter the delegation principle implemented in the abstract `ClassLoader` class (the

source code above). The custom logic should be implemented in the `findClass` method (since it is called only when the parent doesn't find the class definition).

```
public class CustomClassLoader extends ClassLoader {

    @Override
    public Class findClass(String name) throws ClassNotFoundException {
        byte[] classBytes = loadClassFromFile(name);
        return defineClass(name, classBytes, 0, classBytes.length);
    }

    private byte[] loadClassFromFile(String fileName) throws ClassNotFoundException
{
        InputStream inputStream = getClass().getClassLoader().getResourceAsStream(
                fileName.replace('.', File.separatorChar) + ".class");
        byte[] buffer;
        ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
        int nextValue = 0;
        try {
            while ( (nextValue = inputStream.read()) != -1 ) {
                byteStream.write(nextValue);
            }
        } catch (IOException e) {
            throw new ClassNotFoundException("Error loading the class file", e);
        }
        buffer = byteStream.toByteArray();
        return buffer;
    }
}
```