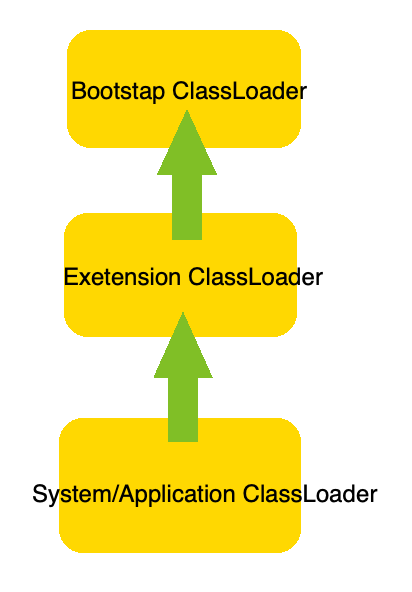
##### **ClassLoader**

The Java ClassLoader is a part of the Java Runtime Environment that dynamically loads Java classes into the Java Virtual Machine. The Java run time system does not need to know about files and file systems because of classloaders.

Java classes aren’t loaded into memory all at once, but when required by an application. At this point, the Java ClassLoader is called by the JRE and these ClassLoaders load classes into memory dynamically.

Three Types of ClassLoaders

They are parent-child relationship.

BootStrap ClassLoader: A Bootstrap Classloader is a Machine code which kickstarts the operation when the JVM calls it. It is not a java class. Its job is to load the first pure Java ClassLoader. Bootstrap ClassLoader loads classes from the location rt.jar. Bootstrap ClassLoader doesn’t have any parent ClassLoaders. It is also called as the Primodial ClassLoader.

Extension ClassLoader: The Extension ClassLoader is a child of Bootstrap ClassLoader and loads the extensions of core java classes from the respective JDK Extension library. It loads files from jre/lib/ext directory or any other directory pointed by the system property java.ext.dirs.

System ClassLoader: An Application ClassLoader is also known as a System ClassLoader. It loads the Application type classes found in the environment variable CLASSPATH, -classpath or -cp command line option. The Application ClassLoader is a child class of Extension ClassLoader.

##### **The **Class** class**

java.lang.Class is one of the most important class in java. It is used to describe the meta information inside a class. When a class is loaded from ClassLoader, one(and only one per classloader) Class object will be created.

Every class or object can call getClass() method or .class field to get the instance of the Class class.

Class c1 = String.class;Class c2 = "hello".getClass();Class c3 = "hi".getClass();

System.out.println(c1 == c2); // trueSystem.out.println(c2 == c3); // true, becuase they all get the same Class object from the classloader.

The object of Class class can perform lots of useful/powerful functionalities related to the class and objects of it.

// a demo classclass Apple{

private String color;

public String getColor() {

return color;

}

public void setColor(String color) {

this.color = color;

}}

public class ChangeDataType {

public static void main(String[] args) throws Exception {

Class appleClass = Class.forName("net.antra.reflection.Apple"); // Load the class without create object from Apple class

Apple a = (Apple) appleClass.newInstance(); // create Apple class using Class.

Constructor[] constructors = appleClass.getConstructors(); // get constructors.

Method[] methods = appleClass.getDeclaredMethods(); // get methods

Annotation[] annotations = appleClass.getDeclaredAnnotations(); // get annotations

Field[] fields = appleClass.getDeclaredFields(); // get fields// During the runtime, all the members inside a class is visible to ClassLoader.// By using Reflection API.// All the methods and field can be called. Even for those who are private.

constructors[0].newInstance();

methods[0].invoke(a);

fields[0].set(a,"newValue");

}}

##### **Reflection API**

As the code above, using reflection api, developers don't need to know the class utill at the runtime. Reflection gives us information about the class to which an object belongs and also the methods of that class which can be executed by using the object. Through reflection we can invoke methods at runtime irrespective of the access specifier used with them.

Combine with annotations, using reflection api can achieve lots of framework jobs.

Below is a small "framework" to print out company value in the annotation Antra.

// Framework code: Annotation@Target({ElementType.METHOD,ElementType.TYPE,ElementType.FIELD})@Retention(RetentionPolicy.RUNTIME)public @interface Antra {

String companyValue() default "Java is the best";}

// Scanning classpublic class ScanDemo {

public static void scanThisClass(String className) {

try {

Class clazz = Class.forName(className);

//Annotation on top of class

Annotation[] ann = clazz.getDeclaredAnnotations();

for (Annotation a : ann) {

if(a instanceof Antra) {

System.out.println(((Antra) a).companyValue());

}

}

//Annotations on Fields

Field[] fields = clazz.getDeclaredFields();

for (Field f : fields) {

Annotation[] fAnn = f.getDeclaredAnnotations();

for (Annotation a : fAnn) {

if(a instanceof Antra) {

System.out.println(((Antra) a).companyValue());

}

}

}

//Annotation on Methods

Method[] methods = clazz.getDeclaredMethods();

for (Method m : methods) {

Annotation[] mAnn = m.getDeclaredAnnotations();

for (Annotation a : mAnn) {

if(a instanceof Antra) {

System.out.println(((Antra) a).companyValue());

}

}

}

} catch (ClassNotFoundException e) {

e.printStackTrace();

}

}

}

//////////// with above code, we are able to read the annotation Antra from any class.

//////////// below is usage code.// Demo Apple @Antra(companyValue = "hqweqwei")public class Apple {

@Antra(companyValue = ".Net is OK")

private String color;

@Antra

public String getColor(){

return color;

}}

// Demo Test. Run the main method.public class TestScan {

public static void main(String[] args) {

ScanDemo.scanThisClass("net.antra.design.scan.Apple");

}}

<https://www.youtube.com/watch?v=_7KOeW05qZc>