Runtime

09 December 2021 15:27

Reflection: It is a mechanism which enables a program to examine its own structure at runtime. Using reflection a function can access the meta-data of an object passed to it at runtime whose type is not known at compile time.
In Java every type is an instance of java.lang.Class<T> and this instance contains the meta-data (type information) of that type. A reference to a java.lang.Class instance can be obtained using following methods

1. From literal name of the type T which is available at compile time
   Class<T> c = T.class;
2. From an instance obj of the class
   Class<?> c = obj.getClass();
3. From fully qualified name N of type which is discovered at runtime
   Class<?> c = Class.forName(N);

Class.forName obtains the instance of java.lang.Class for the specified type name using the built-in class loader. This class loader reads binary representation of type p.T from path p/T.class which it looks for in each location (directories and archives) specified in the *classpath* property of the JVM which defaults to the current directory.

Instance method binding

Binding type	Class name	Method name	Usage
static	compile time	compile time	implicit for final methods
dynamic	runtime	compile time	implicit for non-final methods
late	runtime	runtime	explicit with reflection

Annotation: It is a meta (descriptive) interface which can be applied as a custom modifier to a declaration target such as package, type (class of interface), field, method, parameter and return value.
An annotation is defined using one of the following *retention policies* which indicate how long this annotation will be retained by its target

1. **SOURCE** - The annotation appears in the source-code of its declaration target but it is discarded by the compiler. An annotation with this level of retention can only be consumed through its compile-time annotation processor.
2. **CLASS** - The annotation is included in the class file along with the binary representation of its declaration target but it is not loaded into the memory at runtime. An annotation with this level of retention can only be consumed through its compile-time annotation processor.

3. **RUNTIME** - The annotation is loaded into the memory along with the meta-data of the declaration target at runtime. An annotation with this level of retention can be directly consumed by applications using reflection.

Functional Programming: It is a declarative style of programming in which data is processed by passing it through a series of functions which do not have side effects and may accept other functions as their arguments.
Java provides syntactical support (from version 8.0) for functional programming using

1. **Method Reference** - It is an identifier of a method with a particular return type and a particular list of parameter types. A method reference is produced by applying double colon (::) operator to a class or an object along with the name of (static or instance) method or from an anonymous (auto-generated) method known as *lambda expression*.
2. **Functional Interface** - It is an interface which contains exactly one abstract method which may be checked at compile time by applying java.lang.*@FunctionalInterface* annotation to the definition of that interface. A functional interface is used as a type for a method reference which is compatible with its abstract method and it is automatically implemented at runtime to invoke the referenced method.

Stream API - It is a part of Java runtime library (from version 8.0) which provides support for processing a sequence of elements known as a *stream* in a functional manner. It includes support for

1. **Operation Pipeline** - The elements of a stream are passed through a chain of methods each performing an *intermediate operation* which consumes a stream and produces another stream except the last method which performs the *terminal operation* which consumes a stream but does not produce any stream.
2. **Lazy Evaluation** - Each method in the operation pipeline processes the incoming stream using its own internal iteration which is only executed when the method which performs the terminal operation is invoked.

Native Method: It is a method defined in a Java class which on invocation calls a global function containing native machine instructions from a platform specific dynamically linkable library (shared object on UNIX). The global function called by a native method is generally implemented in C/C++ and it consumes services offered by the Java runtime using the *Java Native Interface* (JNI) exposed by the JVM.
Purpose

1. To increase performance of Java application by eliminating excessive verification performed during execution of bytecodes.

2. To consume platform specific services which are not exposed by the Java runtime library.
3. To reuse native libraries implemented for legacy (non-Java) applications.

Disadvantages

1. It can compromise the runtime safety generally ensured by the JVM.
2. It can limit the portability of the application to different platforms supported by Java.