Core JDK APIs

Agenda

- Reflection
- Regular expressions

Overview

Java SE

- Apart from the Core language Java SE (standard edition) comes with a number of useful utilities …

collections	Io/nio	date & time
object comparison	object cloning	XML/JSON parsing
concurrency	object serialization	security
networking	logging	GUI APIs
JDBC	process API	JShell
JMX	reflection	
lambdas	regex	
streams	locales	

In blue are the ones already covered and in green are the ones for which there is a dedicated session

Reflection

What is reflection ?

- Type introspection is the ability of a running Java application to retrieve information about itself such as classes, methods in a class etc.

- Reflection is the ability of a running Java application to manipulate itself and uses type introspection capabilities of Java SE

Key mechanism to work with code that hasn’t been written yet

What is reflection ?

- The implementation of Java reflection utilities is provided by the **java.lang.reflect** package

- A typical example of using a reflection is to dynamically invoke a method of an unknown object

```
// invoking the someMethod method (that does not have
// parameters) on object
Method method = object.getClass().
       getMethod("someMethod", null);
method.invoke(object, null);
```

Reflection should be used only by purpose ! Overusing reflection results in degraded performance, security issues and overcomplicated source code.

Reflection uses

- In practice many frameworks and libraries make use of reflection for different purposes:
  - Spring framework uses reflection to dynamically load beans from XML definitions
  - JAXP (Java API for XML Parsing) to dynamically load the XML parser to use
  - Persistence frameworks like Hibernate uses reflection to process Hibernate configuration and determine how to link entities
  - Unit testing frameworks like Junit use reflection to determine test methods in a class that need to be called

Class metadata

- Every type whether it is object or primitive has a **java.lang.Class** instance associated
- It is the entrypoint to the Reflection API
- There are different ways to retrieve the Class instance for a particular primitive type or object

Retrieving the class

- For an object the class can be retrieved with the getClass() method

```
Table table = new Table();
Class<? extends Table> tableClass = table.getClass();
```

- The class instance can be retrieving by using .class on the primitive or object type

```
Class<Integer> intClass = int.class;
Class<Table> tableClass = Table.class;
Class<double[][]> doubleMatrixClass = double[][].class;
```

Retrieving the class

- That static Class.forName() method can also be used to retrieve the class instance from the fully qualified class name

```
Class tableClass =
       Class.forName("com.martin-toshev.Table");
Class doubleMatrixClass = Class.forName("[[D");
```

- If a class is not found than a **java.lang.ClassNotFoundException** exception is thrown

Class.forName cannot be used for primitive types

In practice one can encounter also a **java.lang.NoClassDefFoundError** whereby the class has been successfully compiled but for some reason its definition cannot be found at runtime (i.e. when creating a new object)

Retrieving the class

- For primitive types the corresponding class can also be retrieved through the wrapper type using the TYPE static field:

```
Class intClass = Integer.TYPE
Class doubleClass = Double.TYPE
```

Retrieving the class

- A class can also be retrieved using any of the methods depending on the particular scenario:

Class.getSuperclass()	Returns the superclass Class instance
Class.getClasses()	Returns the public member classes and interface Class instances (including the inherited)
Class.getDeclaredClasses()	The same as getClasses() but returns only the classes declared in the class (with any package access modified)
Class.getDeclaringClass()	Returns the wrapper class where the specified is declared
Class.getEnclosingClass()	Returns the enclosing class (i.e. for anonymous type)
Field.getDeclaringClass()	Returns the class of an instance field
Method.getDeclaringClass()	Returns the class of an instance method
Constructor.getDeclaringClass()	Returns the class of a constructor

Class introspection

- The following methods from the Class API can be used to retrieve information about the class:

getModifiers()	Returns the class modifers
getField() getDeclaredField()	Returns a class field by name
getFields() getDeclaredFields()	Returns the class fields
getMethod() getDeclaredMethod()	Returns a class method by name and parameter types
getMethods() getDeclaredMethods()	Returns the class methods
getConstructor() getDeclaredConstructor()	Returns a constructor by parameter types
getConstructors() getDeclaredConstructors()	Returns the class constructors

Field manipulation

- The following methods can be used on a Field instance:

getType() getGenericType()	Returns the field type
getModifiers()	Returns the field modifiers
get() getInt()/getLong() …	Returns the current value of the field. Getters values are available for fields or primitive type
set() setInt()/setLong() …	Sets a value for the field on a specified instance. Setter methods are available for fields of primitive type
setAccessible()	It **true** allows a value of the field to be set even if private or final
getName()	Returns the field name
getDeclaredAnnotations()	Returns the annotations declared on the field

Since modifiers are returned as an integer representing the modifiers as flags the static **Modifier.toString()** method can be used to return the modifiers as a string

Method manipulation

- The following methods can be used on a Field instance:

getReturnType() getGenericReturnType()	Returns the return type of the method
getParameters()	Returns the method parameters
getExceptionTypes()	Returns the exceptions that the method declares using **throws**
getModifiers()	Returns the method modifiers
getName()	Returns the method name
invoke()	Returns the field name
getDeclaredAnnotations()	Returns the annotations declared on the method

Since JDK 8 parameter names can be retrieved if the **–parameters** option is passed to the Java compiler

Constructor manipulation

- The following methods can be used on a Constructor instance:

getParameters()	Returns the constructor parameters
getExceptionTypes()	Returns the exceptions that the constructor declares using **throws**
getModifiers()	Returns the constructor modifiers
newInstance()	Creates a new object instance using the constructor and the expected parameters
getDeclaredAnnotations()	Returns the annotations declared on the method

A class with a default constructor can also be instantiated with the **Class.newInstance()** method but since JDK 9 it is deprecated and should not be used

Array creation

- The Reflection API provides an Array class that can be used to create an array:

```
Table[] tables = (Table[])
       Array.newInstance(Table.class, 10);
```

- The class also provides getter/setter methods that can be used to modify or retrieve a value from the array

```
Array.get(tables, 5);
Array.set(tables, 2, new Table());
```

Other Class operations

- Additional useful operations from the Class API include:

isArray()	Checks if the given class is an array
isEnum()	Checks if the given class is an enum
getEnumConstants()	Returns the enum constants (if the class is an enum)
isInstance()	Checks if a given object is instance of the class
isAssignableFrom()	Determines if the class is a superclass/superinterface of the class/interface provides as a parameter
cast()	Casts an object to the class or interface represented by the this Class instance
getPackage()	Returns the package of the class
getModule()	Returns the module of the class
getInterfaces()	Returns the Class instances of the interfaces implemented by the class

Other Class operations

- Additional useful operations from the Class API include:

getClassLoader()	Returns the ClassLoader instance used to load the class
getClasses()	Returns the member classes of the class
getAnnotations()	Returns the annotations declared on the class
getResource()	Returns a URL to a resource (file) relative to the class
getResourceAsStream()	Returns an InputStream to a resource (file) relative to the class
getProtectionDomain	Returns the protection domain of the class

The protection domain of a class conveys the security-related information for the class as per the Java security sandbox model discussed later in the session

Dynamic proxy

- Dynamic proxies provide the possibility to tunnel method invocations of a class through a common method of a proxy class
- In the Java reflection class a dynamic proxy is created by the **java.lang.reflect.Proxy** class
- It provides the possibility to return a proxy that implements dynamically and interface
- The methods of the interface are executed by the invoke() method of a handler class

Dynamic proxy

```
public class DynamicInvocationHandler
        implements InvocationHandler {

        @Override
        public Object invoke(Object proxy,
                Method method, Object[] args)
                        throws Throwable {
                // do some logic and call method …
        }
}
```

```
List proxy = (List) Proxy.newProxyInstance(
        DynamicProxyTest.class.getClassLoader(),
        new Class[] { List.class },
        new DynamicInvocationHandler());
```

Dynamic proxy

- Dynamic proxies have a wide range of applications:
  - creation of database transactions (query is proxied to a method that commits or rolls-back the transaction)
  - unit testing (for invoking mock (fake) objects instead of concrete implementations)
  - for AOP (aspect-oriented-programming) whereby a method invocation can be intercepted and amended with additional logic or modified

Limitations of reflections

- Although the Reflection API is quite powerful it is still limited in terms of modifying the actual class (adding, modifying or deleting methods or fields etc.)

- This capability can be providing by:

  - custom classloaders where everytime a new class is required a new classloader instances that loads it is created

  - Provide a Java instrumentation agent using the Java instrumentation API

Limitations of reflections

- The reflection API also does not provide the capability to generate a Java class at runtime and load it in the JVM
- This capability can be achieved by third-party libraries such as:
  - cglib
  - asm
  - Javassist
  - BCEL
- The JShell API introduced in JDK 9 can also be used to dynamically create and load a Java class

Regular expressions

Regular expressions

- Regular expressions are sequence of characters that define a search pattern

- They are a concept formalized theoretically using formal language theory

- Example (basic expression for an email)

```
^(.+)@(.+)$
```

Regular expressions

- Regular expressions capabilities in Java are provided by the **java.util.regex** package

- The main class used to represent a regular expression is **java.util.regex.Pattern**

- A **java.util.regex.Matcher** is used to match text against a pattern

```
Pattern pattern = Pattern.compile("^(.+)@(.+)$");
Matcher matcher = pattern.matcher("mail@test.com");
System.out.println(matcher.matches()); // true
matcher = pattern.matcher("test.com");
System.out.println(matcher.matches()); // false
```

Regular expression syntax

<string_literal> (i.e. abc)	Matched directly
[a-z] [a-zA-Z]	Any of the characters in a range (between **a** and **z** in the example)
[^abc]	Any characters except the specified (abc in the exmaple)
<pattern>|<pattern>	Logical OR (matches first or second pattern)
\d	A digit [0-9]
\D	A non-digit [^0-9]
\s	A whitespace characters
\S	A non-whitespace character
\w	A word character [a-zA-Z_0-9]
\W	[^\w]

A good summary of regular expression constructs is also provided by the Pattern Javadoc

Regular expression syntax

- Quantifiers provide the possibility to specify number of occurrences of a match (by default they are greedy)

<pattern>?	Either 0 or 1 matches
<pattern>*	Zero or more matches
<pattern>+	One or more matches
<pattern>{n}	Exactly **n** matches
<pattern>{n, }	At least **n** matches
<pattern>{n, m}	At least **n** but maximum **m** matches

- Two additional characters can be appended at the end of a quantifier
  - ?: makes the quantifier match as little as possible (reluctant)
  - +: makes the quantifier try maximum match (does not fallback as the default greedy quantifier to try a smaller match)

Regular expression syntax

- Parts of the matched text can be retrieved with capturing groups using the () brackets
- Group 0 is special and represents the entire match
- Matched groups can also be referenced in the regular expression with \<group_number>

Regular expression syntax

```
Pattern pattern = Pattern.compile("^(.+)@(.+)$");
Matcher matcher = pattern.matcher("mail@test.com");
while(matcher.find()) {
        // mail@test.com
        System.out.println(matcher.group(0));
        // mail
        System.out.println(matcher.group(1));
        // test.com
        System.out.println(matcher.group(2));
        // IndexOutOfBoundsException exception
        System.out.println(matcher.group(3));
}
```

Regular expression syntax

- Boundary matches can be used to specify where to match:
  - ^: matching should start at start of line
  - $: matching should end at end of line
- Example (matching a phone number):

```
"^\\+(?:[0-9] ?){6,14}[0-9]$"
```

The String.matches() method can be used a shorthand to check if a String matches a regular expression

Pattern flags

- The **Pattern.compile()** method may accept different flags (combined with | into a single integer) such as:
  - Pattern.CASE_INSENSITIVE: ignores case while matching
  - Pattern.MULTILINE: matches the entire text (and not line by line)
  - Pattern.ALL_FLAGS: includes all flags
  - others flags in the Patterns class

```
int flags = Pattern.CASE_INSENSITIVE |
                    Pattern.MULTILINE;
Pattern pattern = Pattern.compile("^(.+)@(.+)$",
              flags);
```

Matcher class

- Methods of the matcher class include:

matches()	Checks if a string matches a pattern
find()	Finds the next occurrence of the match (returns false in case no more matches occur)
start(n)	Returns the start index of group **n**
end(n)	Returns the end index of group **n**
pattern()	Returns the Pattern for which this matcher applies
group(n)	Returns the group with index **n**
replaceFirst()	Replaces the first matched occurrence with a string
replaceAll()	Replaces all matches with a string

Regular expression tools

- Working with regular expression requires testing against sample input
- It is not always convenient to run multiple times a sample code and debug how a string is processed by a regular expression
- For that reason regular expression testing tools such as this, this and this come in handy

Questions ?