Agenda

- Reflection
- Annotation
- Nested Classes
- Local Class
- enum

Reflection

- It is a technique to read the metadata and work with that data.
- .class = Byte-code + Meta-data + Constant pool + ...
- When class is loaded into JVM all the metadata is stored in the object of java.lang.Class (heap area).
- This metadata includes class name, super class, super interfaces, fields (field name, field type, access modifier, flags), methods (method name, method return type, access modifier, flags, method arguments, ...), constructors (access modifier, flags, ctor arguments, ...), annotations (on class, fields, methods, ...).

Reflection applications

- Inspect the metadata (like javap)
- Build IDE/tools (Intellisense)
- Dynamically creating objects and invoking methods
- Access the private members of the class

Get the java.lang.Class object

- way 1: When you have class-name as a String (taken from user or in properties file)

```
Class<?> c = Class.forName(className);
```

- way 2: When the class is in project/classpath.

```
Class<?> c = ClassName.class;
```

- way 3: When you have object of the class.

```
Class<?> c = obj.getClass();
```

Access metadata in java.lang.Class

- Name of the class

```
String name = c.getName();
```

- Super class of the class

```
Class<?> supcls = c.getSuperclass();
```

- Super interfaces of the class

```
Class<?> supintf[] = c.getInterfaces();
```

- Fields of the class

```
Field[] fields = c.getFields(); // all fields accessible (of class & its
super class)
```

```
Field[] fields = c.getDeclaredFields(); // all fields in the class
```

- Methods of the class

```
Method[] methods = c.getMethods(); // all methods accessible (of class & its
super class)
```

```
Method[] methods = c.getDeclaredMethods(); // all methods in the class
```

- Constructors of the class

```
Constructor[] ctors = c.getConstructors(); // all ctors accessible (of class
& its super class)
```

```
Constructor[] ctors = c.getDeclaredConstructor(); // all ctors in the class
```

Invoking method dyanmically

```

```

```
    public class Middleware {
        public static Object invoke(String className, String methodName, Class[]
methodParamTypes, Object[] methodArgs) throws Exception {
            // load the given class
            Class c = Class.forName(className);
            // create object of that class
            Object obj = c.newInstance(); // also invokes param-less constructor
            // find the desired method
            Method method = c.getDeclaredMethod(methodName, methodParamTypes);
            // allow to access the method (irrespective of its access specifier)
            method.setAccessible(true);
            // invoke the method on the created object with given args & collect
the result
            Object result = method.invoke(obj, methodArgs);
            // return the results
            return result;
        }
    }
```

```
    // invoking method statically
    Date d = new Date();
    String result = d.toString();
```

```
    // invoking method dyanmically
    String result = Middleware.invoke("java.util.Date", "toString", null, null);
```

```
class Test {
  void sayHello() {
    System.out.println("Hello from text class");
  }

  void sayHelloTo(String name) {
    System.out.println("Hello " + name);
  }
}

public static void main(String[] args) {
        invoke("com.sunbeam.p1.Test", "sayHello", null, null);
        invoke("com.sunbeam.p1.Test", "sayHelloTo", new Class[]{String.class}, new
Object[] { "Rohan" });
}
```

Annotations

- Added in Java 5.0.

- Annotation is a way to associate metadata with the class and/or its members.
- Annotation applications
  - Information to the compiler
  - Compile-time/Deploy-time processing
  - Runtime processing
- Annotation Types
  - Marker Annotation: Annotation is not having any attributes.
    - @Override, @Deprecated, @FunctionalInterface ...
  - Single value Annotation: Annotation is having single attribute -- usually it is "value".
    - @SuppressWarnings("deprecation"), ...
  - Multi value Annotation: Annotation is having multiple attribute
    - @RequestMapping(method = "GET", value = "/books"), ...

Pre-defined Annotations

- @Override
  - Ask compiler to check if corresponding method (with same signature) is present in super class.
  - If not present, raise compiler error.
- @FunctionalInterface
  - Ask compiler to check if interface contains single abstract method.
  - If zero or multiple abstract methods, raise compiler error.
- @Deprecated
  - Inform compiler to give a warning when the deprecated type/member is used.
- @SuppressWarnings
  - Inform compiler not to give certain warnings: e.g. deprecation, rawtypes, unchecked, serial, unused
  - @SuppressWarnings("deprecation")
  - @SuppressWarnings({"rawtypes", "unchecked"})
  - @SuppressWarnings("serial")
  - @SuppressWarnings("unused")

Meta-Annotations

- Annotations that apply to other annotations are called meta-annotations.
- Meta-annotation types defined in java.lang.annotation package.

@Retention

- RetentionPolicy.SOURCE
  - Annotation is available only in source code and discarded by the compiler (like comments).
  - Not added into .class file.
  - Used to give information to the compiler.
  - e.g. @Override, ...
- RetentionPolicy.CLASS
  - Annotation is compiled and added into .class file.
  - Discared while class loading and not loaded into JVM memory.
  - Used for utilities that process .class files.

    - e.g. Obfuscation utilities can be informed not to change the name of certain class/member using @SerializedName, ...
- RetentionPolicy.RUNTIME
    - Annotation is compiled and added into .class file. Also loaded into JVM at runtime and available for reflective access.
    - Used by many Java frameworks.
    - e.g. @RequestMapping, @Id, @Table, @Controller, ...

@Target

- Where this annotation can be used.
- ANNOTATION_TYPE, CONSTRUCTOR, FIELD, LOCAL_VARIABLE, METHOD, PACKAGE, PARAMETER, TYPE, TYPE_PARAMETER, TYPE_USE
- If annotation is used on the other places than mentioned in @Target, then compiler raise error.

@Documented

- This annotation should be documented by javadoc or similar utilities.

@Repeatable

- The annotation can be repeated multiple times on the same class/target.

@Inherited

- The annotation gets inherited to the sub-class and accessible using c.getAnnotation() method.

Custom Annotation

- Annotation to associate developer information with the class and its members.

```
//@Repetable(Information.class) // to make it repetable
@Retention(RetentionPolicy.RUNTIME) // the def attribute is considered as
"value" = @Retention(value = RetentionPolicy.RUNTIME )
@Taget({TYPE, CONSTRUCTOR, FIELD, METHOD}) // { } represents array
@interface Developer {
    String firstName();
    String lastName();
    String company() default "Sunbeam";
    String value() default "Software Engg";
}

@Inherited
@Documented
@Retention(RetentionPolicy.RUNTIME)
@Taget({TYPE})
@interface CodeType {
    String[] value();
}
```

```
@Taget({TYPE, CONSTRUCTOR, FIELD, METHOD})
@Retention(RetentionPolicy.RUNTIME)
@interface Information {
  Developer[] value();
}
```

```
//@Developer(firstName="Nilesh", lastName="Ghule", value="Technical
Director") // compiler error -- @Developer is not @Repeatable
@CodeType({"businessLogic", "algorithm"})
@Developer(firstName="Nilesh", lastName="Ghule", value="Technical Director")
class MyClass {
    // ...
    @Developer(firstName="Shubham", lastName="Borle", company="Sunbeam Karad
")
    private int myField;
    @Developer(firstName="Rahul", lastName="Sansuddi")
    public MyClass() {

    }
    @Developer(firstName="Shubham", lastName="Borle", company="Sunbeam Karad
")
    public void myMethod() {
        @Developer(firstName="James", lastName="Bond") // compiler error
        int localVar = 1;
    }
}
```

```
// @Developer is inherited
@CodeType("frontEnd")
@CodeType("businessLogic") // allowed because @CodeType is @Repeatable
class YourClass extends MyClass {
    // ...
}
```

Annotation processing (using Reflection)

```
Annotation[] anns = MyClass.class.getDeclaredAnnotations();
for (Annotation ann : anns) {
    System.out.println(ann.toString());
    if(ann instanceof Developer) {
        Developer devAnn = (Developer) ann;
        System.out.println(" - Name: " + devAnn.firstName() + " " + devAnn.
lastName());
        System.out.println(" - Company: " + devAnn.company());
        System.out.println(" - Role: " + devAnn.value());
    }
}
```

```
System.out.println();

Field field = MyClass.class.getDeclaredField("myField");
anns = field.getAnnotations() ;
for (Annotation ann : anns)
    System.out.println(ann.toString());
System.out.println();

//anns = YourClass.class.getDeclaredAnnotations();
anns = YourClass.class.getAnnotations();
for (Annotation ann : anns)
    System.out.println(ann.toString());
System.out.println();
```

Member/Nested classes

- By default all Java classes are top-level.
- In Java, classes can be written inside another class/method. They are Member classes.
- Four types of member/nested classes
    - Static member classes --
    - Non-static member class --
    - Local class --
    - Annoymous Inner class --
- When .java file is compiled, separate .class file created for outer class as well as inner class.

Static member classes

- Like other static members of the class (belong to the class, not the object).
- Accessed using outer class (Doesn't need the object of outer class).
- Can access static (private/public) members of the outer class directly.
- Static member class cannot access non-static members of outer class directly.
- The outer class can access all members (including private) of inner class directly (no need of getter/setter).
- The static member classes can be private, public, protected, or default.

```
class Outer {
    private int nonStaticField = 10;
    private static int staticField = 20;

    public static class Inner {
        public void display() {
            System.out.println("Outer.nonStaticField = " + nonStaticField);
// error
            System.out.println("Outer.staticField = " + staticField); // ok
- 20
```

```
        }
    }
}
public class Main {
    public static void main(String[] args) {
        Outer.Inner obj = new Outer.Inner();
        obj.display();
    }
}
```

Non-static member classes/Inner classes

- Like other non-static members of the class (belong to the object/instance of Outer class).
- Accessed using outer class object (Object of outer class is MUST).
- Can access static & non-static (private) members of the outer class directly.
- The outer class can access all members (including private) of inner class directly (no need of getter/setter).
- The non-static member classes can be private, public, protected, or default.

```
class Outer {
    private int nonStaticField = 10;
    private static int staticField = 20;
    public class Inner {
        public void display() {
            System.out.println("Outer.nonStaticField = " + nonStaticField);
// ok-10
            System.out.println("Outer.staticField = " + staticField); // ok-
20
        }
    }
}
public class Main {
    public static void main(String[] args) {
        //Outer.Inner obj = new Outer.Inner(); // compiler error
        // create object of inner class
            //Outer outObj = new Outer();
            //Outer.Inner obj = outObj.new Inner();
        Outer.Inner obj = new Outer().new Inner();
        obj.display();
    }
}
```


- If Inner class member has same name as of outer class member, it shadows (hides) the outer class member. Such Outer class members can be accessed explicitly using `Outer.this`.

Static member class and Non-static member class -- Application

```
// top-level class
class LinkedList {
    // static member class
    static class Node {
        private int data;
        private Node next;
        // ...
    }
    private Node head;
    // non-static member class
    class Iterator {
        private Node trav;
        // ...
    }
    // ...
    public void display() {
        Node trav = head;
        while(trav != null) {
            System.out.println(trav.data);
            trav = trav.next;
        }
    }
}
```

Local class

- Like local variables of a method.
- The class scope is limited to the enclosing method.
- If enclosed in static method, behaves like static member class. If enclosed in non-static method, behaves like non-static member class.
- Along with Outer class members, it can also access (effectively) final local variables of the enclosing method.
- We can create any number of objects of local classes within the enclosing method.

```
public class Main {
    private int nonStaticField = 10;
    private static int staticField = 20;
    public static void main(String[] args) {
        final int localVar1 = 1;
        int localVar2 = 2;
        int localVar3 = 3;
        localVar3++;
        // local class (in static method) -- behave like static member class
        class Inner {
            public void display() {
                System.out.println("Outer.nonStaticField = " +
nonStaticField); // error
                System.out.println("Outer.staticField = " + staticField); //
ok 20
```

```
                System.out.println("Main.localVar1 = " + localVar1); // ok 1
                System.out.println("Main.localVar2 = " + localVar2); // ok 2
                System.out.println("Main.localVar3 = " + localVar3); //
error
            }
        }
        Inner obj = new Inner();
        obj.display();
        //new Inner().display();
    }
}
```

Annoymous Inner class

- Creates a new class inherited from the given class/interface and its object is created.
- If in static context, behaves like static member class. If in non-static context, behaves like non-static member class.
- Along with Outer class members, it can also access (effectively) final local variables of the enclosing method.

```
// (named) local class
class EmpnoComparator implements Comparator<Employee> {
    public int compare(Employee e1, Employee e2) {
        return e1.getEmpno() - e2.getEmpno();
    }
}
Arrays.sort(arr, new EmpnoComparator());    // anonymous obj of local class
```

```
// Anonymous inner class
Comparator<Employee> cmp = new Comparator<Employee>() {
    public int compare(Employee e1, Employee e2) {
        return e1.getEmpno() - e2.getEmpno();
    }
};
Arrays.sort(arr, cmp);
```


Enum

- In C enums were internally integers
- In java, It is a keyword added in java 5 and enums are object in java.
- used to make constants for code readability
- mostly used for switch cases
- In java, enums cannot be declared locally (within a method).
- The declared enum is converted into enum class.

- The enum type declared is implicitly inherited from java.lang.Enum class. So it cannot be extended from another class, but enum may implement interfaces.
- The enum constants declared in enum are public static final fields of generated class.
- Enum objects cannot be created explicitly (as generated constructor is private).
- The enums constants can be used in switch-case and can also be compared using == operator.
- The enum may have fields and methods.

```
public abstract class Enum<E> implements java.lang.Comparable<E>,
java.io.Serializable {
    private final String name;
    private final int ordinal;

    protected Enum(String,int); // sole constructor - can be called from user-
defined enum class only

    public final String name(); // name of enum const
    public final int ordinal(); // position of enum const (0-based)
    public String toString(); // returns name of const
    public final int compareTo(E);// compares with another enum of same type on
basis of ordinal number
    public static <T> T valueOf(Class<T>, String);
    // ...
}
```

```
// user-defined enum
enum ArithmeticOperations {
    ADDITION, SUBTRACTION, MULIPLICATION, DIVISION
}

// generated enum code
final class ArithmeticOperations extends Enum {

private ArithmeticOperations(String name, int ordinal) {
    super(name, ordinal); // invoke sole constructor Enum(String,int);
}

public static ArithmeticOperations[] values() {
    return (ArithmeticOperations[])$VALUES.clone();
}

public static ArithmeticOperations valueOf(String s) {
    return (ArithmeticOperations)Enum.valueOf(ArithmeticOperations,s);
}


  public static final ArithmeticOperations ADDITION;
  public static final ArithmeticOperations SUBTRACTION;
  public static final ArithmeticOperations MULIPLICATION;
  public static final ArithmeticOperations DIVISION;
```

```
    private static final ArithmeticOperations $VALUES[];

    static {
      ADDITION = new ArithmeticOperations("ADDITION", 0);
      SUBTRACTION = new ArithmeticOperations("SUBTRACTION", 1);
      MULIPLICATION = new ArithmeticOperations("MULIPLICATION", 2);
      DIVISION = new ArithmeticOperations("DIVISION", 3);
      $VALUES = (new ArithmeticOperations[] {
        ADDITION, SUBTRACTION, MULIPLICATION, DIVISION
        });
    }
  }
```