STATIC FIELDS AND METHODS

Final Fields

- final fields **cannot be re-assigned** after they were initialized in constructors or field initializer

```
public class Student {
   private final String name ; // name is declared as final
   private int year = 1 ;
   private String major ;

   public Student(String name, String major) {
      this.name = name ; // name can be initialized in constructor
      this.major = major ;
   }
   void setYear(int year) { this.year = year ; }
   void setName(String name) { this.name = name ; } // Not Allowed !
   void setMajor(String major) { this.major = major ; }
   public static void main(String[] args) {
      var s1 = new Student("James", "Computer") ;
      s1.setYear(2) ;
      s1.setMajor("Mechanical") ;
      s1.setName("Brown") ; // Impossible !
   }
}
```

Static Fields

- Static fields are shared by all the objects of a class. (class variable)

```
class Rectangle7 {
    private Point leftTop, rightBottom ;
    public static int AllCount = 0 ;

    public Rectangle7(Point p1, Point p2) {
        AllCount ++ ;
        leftTop = new Point(p1.getX(), p1.getY()) ;
        rightBottom = new Point(p2.getX(), p2.getY()) ;
    }
    public Rectangle7() { AllCount ++ ; }
    public String toString() { return leftTop + "," + rightBottom ; }
}

public class StaticField {
    public static void main(String[] args) {
        var r1 = new Rectangle7() ;
        var r2 = new Rectangle7(new Point(0, 0), new Point(10, 20)) ;

        System.out.println(Rectangle7.AllCount) ;
        System.out.println(r1) ; System.out.println(r2) ;
    }
}
```

Constructors are also overloaded!

```
2
null,null
(0, 0),(10, 20)
```

Constant

- public static final is a common way to defining constants.

```
class Rectangle {
        public static final int NO_OF_SIDE = 4 ;
        ...
}
```

- More examples

java.lang.Math	
public static final double E	2.718281828459045d
public static final double PI	3.141592653589793d

java.lang.Integer	
public static final double MAX_VALUE	2147483647
public static final double MIN_VALUE	-2147483648

Static Methods

- Static methods can only access static fields and invoke static methods

```
class Rectangle8 {
   private Point leftTop, rightBottom ;
   private static int AllCount = 0 ;
   public static boolean noRectangle() { return AllCount == 0 ; }
   public static int getAllCount() { return AllCount ; }

   public Rectangle8(Point p1, Point p2) {
      AllCount ++ ;
      leftTop = new Point(p1.getX(), p1.getY()) ;
      rightBottom = new Point(p2.getX(), p2.getY()) ;
   }
   public Rectangle8() { AllCount ++ ; }
}
public class StaticMethod {
   public static void main(String[] args) {
      var r1 = new Rectangle8() ;
      var r2 = new Rectangle8(new Point(0, 0), new Point(10, 20)) ;

      System.out.println(Rectangle8.getAllCount()) ;
   }
}
```

Static Methods

- Standard mathematical methods in class Math are defined as public static methods.

```
class Math {
   public static double pow(double base, double exponent) { ... }
   public static double abs(double argument) { ... }
   public static double abs(float argument) { ... }
   public static double abs(long argument) { ... }
   public static double abs(int argument) { ... }

   public static double min(double n1, double n2) { ... }
   ...

}
```

```
   if ( Math.abs(-10) == 10 ) ...

   Math.min(10.5, 20) ;
```

Initialization of Objects

```
class Employee {
   // 5. constructors
   public Employee(String n, double s) { name = n; salary = s; }
   public Employee(double s) { this("Employee #" + nextId, s); }
   public Employee() {
      // name  = ""“, salary  =1000, id initialized in initialization block
   }
   public String getName() { return name; }
   public int getId() { return id; }
   public double getSalary() { return salary ; }

   private static int nextId; //1. static field (default value)
   private int id; // = 0; // 3. instance field (default value)
   private String name = ""; // 3.1 instance field initialization
   private double salary = 1000 ; // 3.2 instance field initialization
   // 2. static initialization block
   static {
      Random generator = new Random();
      nextId = generator.nextInt(10000);
   }
   // 4 object initialization block
   { id = nextId; nextId++; }
}
```

For the first object

1. **Static field** (0, false, or null)
2. **Static initialization block**

For each object

3. **Field** Data fields ➔ **default value** (0, false, or null)
4. **Field initializer** and **initialization block** in the order of declaration
5. **Constructor Body**

Initialization of Objects

```
public class Initialization {

   public static void main(String[] args) {

      Employee[] staff = new Employee[3];

      staff[0] = new Employee("Robert", 40000);
      staff[1] = new Employee(60000);
      staff[2] = new Employee();

      for ( var e : staff)
         System.out.printf("name=%-15s,id=%6d,salary=%-10.1f%n",
            e.getName(), e.getId(), e.getSalary() );
   }
}
```

```
name=Robert              ,id=   6072,salary=40000.0
name=Employee #6073      ,id=   6073,salary=60000.0
name=                    ,id=   6074,salary=10000.0
```

Working with *null* Reference

```
public class Employee {
   private final String name;

   public Employee(String name) {
      if ( name == null )
         throw new NullPointerException("Employee name should be given");
      this.name = name;
   }

   public String getName() { return name; }

   public static void main(String[] args) {
      Employee e1 = new Employee("Brown");
      System.out.println(e1.getName());

      Employee e2 = new Employee(null);
      System.out.println(e2.getName());
   }
}
```

Working with *null* Reference

- Objects.requireNonNull(T obj, String message)
- Objects.requireNonNullElse(T obj, T defaultObj)

```
public class Employee {
   private final String name;

   public Employee(String name) {
      // if ( name == null )
      //    throw new NullPointerException("Employee name should be given");
      // this.name = name;

      this.name = Objects.requireNonNull(name, "Employee name should be given");
      // this.name = Objects.requireNonNullElse(name, "Unknown"); // As of Java 9
   }
}
```

REFLECTION

Reflection

- With reflection, you can analyze the capabilities of classes from their byte codes.

```
Enter class name (e.g. java.util.Date):  Person

class Person
{
   public Person(java.lang.String, int, java.lang.String) { }

   public int getAge() { }
   public void increaseAge() { }
   public void moveTo(java.lang.String) { }
   public java.lang.String toString() { }
   public java.lang.String getAddress() { }
   public java.lang.String getName() { }
   public void rename(java.lang.String) { }

   private java.lang.String name;
   private int age;
   private java.lang.String address;
}
```

```
import java.util.*;
import java.lang.reflect.*;
public class ReflectionTest {
  public static void main(String[] args) {
     String name;
     if (args.length > 0) name = args[0];
     else  {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter class name (e.g. java.util.Date): ");
        name = scanner.next();
        scanner.close();
     }
     try {
        // print class name and superclass name
        final Class<?> cl = Class.forName(name); // java.lang.Class
        final Class<?> supercl = cl.getSuperclass();
        System.out.print("class " + name);
        if (supercl != null && supercl != Object.class)
            System.out.print(" extends " + supercl.getName());

        System.out.print("₩n{₩n");
        printConstructors(cl);
        System.out.println();
        printMethods(cl);
        System.out.println();
        printFields(cl);
        System.out.println("}");
     }
     catch(ClassNotFoundException e) { e.printStackTrace(); }
  }
```

Polymorphism in Generic type

<? extends T>: T and it's subclass
<? super T> : T and it's superclass
<?> : all classes

```
public static void printConstructors(final Class<?> cl) {
   // java.lang.reflect.Constructor
    final Constructor<?>[] constructors = cl.getDeclaredConstructors();

    for (final Constructor<?> constructor : constructors) {
       System.out.print("    " + Modifier.toString(constructor.getModifiers()));
       System.out.print(" " + constructor.getName() + "(");

       // print parameter types
       final Class<?>[] parameterTypes = constructor.getParameterTypes();
       for (int j = 0; j < parameterTypes.length; j++) {
          if (j > 0) System.out.print(", ");
          System.out.print(parameterTypes[j].getName());
       }
       System.out.println(") { }");
    }
}
```

```
public static void printMethods(final Class<?> cl) {
    final Method[] methods = cl.getDeclaredMethods();
    for (final Method method : methods) {
        final Class<?> returnType = method.getReturnType();

        // print modifiers, return type and method name
        System.out.print("    " + Modifier.toString(method.getModifiers()));
        System.out.print(" " + returnType.getName() + " " + method.getName() + "(");

        // print parameter types
        final Class<?>[] parameterTypes = method.getParameterTypes();
        for (int j = 0; j < parameterTypes.length; j++) {
            if (j > 0) System.out.print(", ");
            System.out.print(parameterTypes[j].getName());
        }
        System.out.println(") { }");
    }
}
```

```
    public static void printFields(final Class<?> cl) {
        final Field[] fields = cl.getDeclaredFields();

        for (final Field field : fields) {
            final Class<?> type = field.getType();
            System.out.print("    " + Modifier.toString(field.getModifiers()));
            System.out.println(" " + type.getName() + " " + field.getName() + ";");
        }
    }
}
```

Q&A