Flow Control

1. Overview

The learning objectives of this laboratory session are:

- How to use control statements in Java
- How to declare classes and methods
- Hands on experience with classes and methods
-

2 Control statements in Java

The following is a reminder of flow control syntax. Each control statement is one logical statement, which often encloses a *block* of statements in curly braces {}. The examples assume the block contains more than one statement.

Indenting is essential. Four spaces are most common.

2.1 Selection (`if, switch`)

2.1.1 if Statement

```
//----- if statement with a true clause
   if (expression) {
       statements // do these if expression is true
   }

//----- if statement with true and false clause
   if (expression) {
       statements // do these if expression is true
   } else {
       statements // do these if expression is false
   }

//----- if statements with many parallel tests
   if (expression1) {
       statements // do these if expression1 is true
   } else if (expression2) {
       statements // do these if expression2 is true
   } else if (expression3) {
       statements // do these if expression3 is true
   . . .
   } else {
       statements // do these no expression was true
   }
```

2.1.2 switch Statement

The effect of the **switch** statement is to choose some statements to execute depending on the integer value of an expression. The same effect can be achieved with a series of cascading if statements, but in some cases the switch statement is easier to read, and in some compilers it can produce more efficient code. The **break** statement exits from the **switch** statement. If there is no **break** at the end of a **case**, execution continues in the next **case**, and this is almost always an error.

```
   switch (expr) {
      case c1:
            statements // do these if expr == c1
            break;
```

```
    case c2:
          statements // do these if expr == c2
          break;
    case c2:
    case c3:
    case c4:         //  Cases can simply fall through.
          statements // do these if expr ==  any of c's
          break;
    . . .
    default:
          statements // do these if expr != any above
}
```

2.2 Loop Statements

2.2.1 while

The **while** statement tests the expression. If the expression evaluates to true, it executes the body of the while. If it is false, execution continues with the statement after the while body. Each time after the body is executed, execution starts with the test again. This continues until the expression is false or some other statement (break or return) stops the loop.

```
while (testExpression) {
    statements
}
```

2.2.2 for

Many loops have an initialization before the loop, and some "increment" before the next loop. The **for** loop is the standard way of combining these parts.

```
for (initialStmt; testExpr; incrementStmt) {
    statements
}
```

This is the same as (except continue will increment):

```
initialStmt;
while (testExpr) {
    statements
    incrementStmt
}
```

2.2.3 do

This is the least used of the loop statements, but sometimes a loop that executes one time before testing is used.

```
do
{
   statements
}
while (testExpression);
```

2.2.4 Other loop controls

All loop statements can be labeled, so that `break` and `continue` can be used from any nesting depth. Labels must precede their use.

```
break;       //exit innermost loop or switch
break label; //exit from loop label
```

```
   continue;    //start next loop iteration
   continue label; //start next loop label
```

Put label followed by colon at front of loop.

```
outer: for (. . .) {
         . . .
         continue outer;
```

2.3 `try...catch` and `throw`

Sketches provided here only for reference. Will be dealt with in detail in a later session.

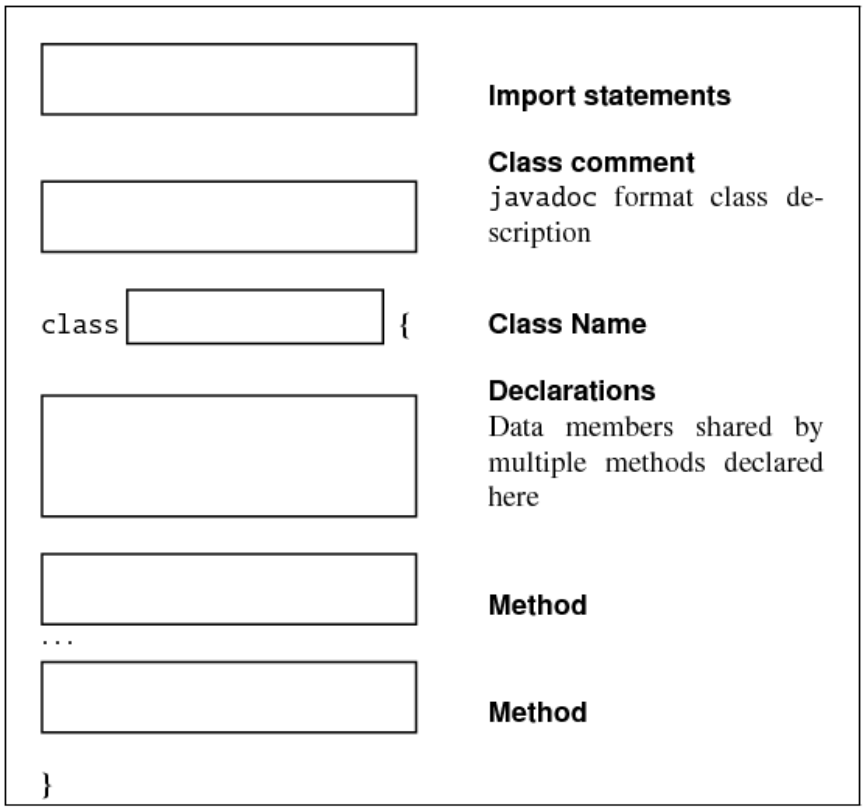
2.3.1 Simple `try...catch` for exceptions

```
try {
    . . . // statements that might cause exceptions
} catch (exception-type x) {
    . . . // statements to handle exception
}
```

2.3.2 `throw`

```
throw exception-object;
```

3 Simple classes



3.1 Class declarations

Classes are the basic building blocks of Java programs. Classes can be compared to the blueprints of buildings. Instead of specifying the structure of buildings, though, classes describe the structure of "things" in a program. These things are then created as physical software objects in the program. Things worth representing as classes are usually important nouns in the problem domain. A Web-based shopping cart application, for example, would likely have classes that represent customers, products, orders, order lines, credit cards, shipping addresses and shipping providers.
Use the following syntax to declare a class in Java:

```
[ public ] [ ( abstract | final ) ] class SomeClassName [ extends SomeParentClass ] [ implements SomeInterfaces ]
   {
      // variables and methods are declared within the curly braces
   }
```

- A class can have **public** or *default* (no modifier) visibility.
- It can be either **abstract**, **final** or *concrete* (no modifier).
- It must have the class keyword, and class must be followed by a legal identifier.
- It may optionally extend one parent class. By default, it will extend **java.lang.Object**.
- It may optionally implement any number of comma-separated interfaces.
- The class's variables and methods are declared within a set of curly braces '{}'.
- Each .java source file may contain only one public class. A source file may contain any number of default visible classes.
- Finally, the source file name must match the public class name and it must have a .java suffix.

3.2 Method declarations

A general syntax for a method declaration is:

```
[modifiers] return_type method_name (parameter_list) [throws_clause] {

   [statement_list]
}
```

Everything within square brackets [] is optional. Of course, you don't include the square brackets in your code; they are included here only to indicate optional items. You can see that the minimal method declaration includes:

- **Return Type**: The return type is either a valid Java type (primitive or class) or void if no value is returned. If the method declares a return type, every exit path out of the method must have a return statement.
- **Method Name**: The method name must be a valid Java identifier. See Java Variables for the rules for Java identifiers.
- **Parameter List**: The parentheses following the method name contain zero or more type/identifier pairs that make up the parameter list. Each parameter is separated by a comma. Also, there can be zero parameters.
- **Curly Braces**: The method body is contained in a set of curly braces. Normally, the method body contains a sequence of semicolon delimited Java statements that are executed sequentially. Technically, though, the method body can be empty.

The combined name and parameter list for each method in a class must be unique. The uniqueness of a parameter list takes the order of the parameters into account. So `int myMethod (int x, String y)` is unique from `int myMethod (String y, int x )`.

A method's visibility (also know as its access scope) defines what objects can invoke it and whether subclasses can override it. The four possible visibility modifiers are: `public`, `protected`, `private`, and no modifier. Keeping an object's methods as hidden as possible helps simplify the object's published API (Application Programming Interface: the specification that defines how the programmer can access the methods and variables of a set of classes). Make a method only as visible as it needs to be. For instance, if a method is intended to be overridden by a subclass but never invoked by client code, make the visibility `protected` instead of `public`. If a method should never be invoked by another class and is not meant to be overridden, make it `private`.
Below is a list of visibility modifiers

Modifier	Can be Accessed By
`public`	any class
`protected`	owning class, any subclass, any class in the same package
No Modifier	owning class, any class in the same package
`private`	owning class

3.2.1 Parameters are Passed by Value

In Java, when a **value** is passed into a **method invocation** as an argument, it is passed **by value**. This is unlike some other programming languages that allow pointers to memory addresses to be passed into methods. When a primitive value is passed into a method, a copy of the primitive is made. The **copy is what is actually manipulated in the method**. So, the value of the copy can be changed within the method, but the original value remains unchanged.

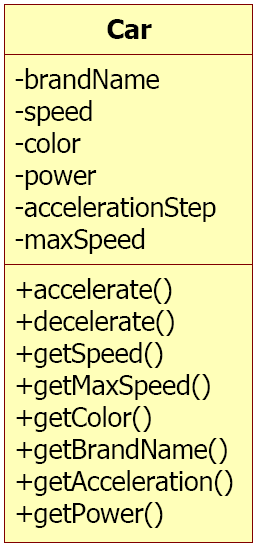
When an **object reference** or **array reference** is passed into a method, a **copy of the reference is actually manipulated by the method**. So, the **method can change the attributes of the object**. But, if it reassigns a new object or array to the reference, the reassignment only affects the copy, not the original reference.

The table below provides a summary of method modifiers:

Modifier	Description
Visibility	Can be one of the values: `public`, `protected`, or `private`. Determines what classes can invoke the method.
`static`	The method can be invoked on the class instead of an instance of the class. For example, `String.valueOf(35)` is calling `valueOf` on the `String` class instead of on any specific String object. Of course, static methods can still be called on object instances: `myString.valueOf(35)`.

`abstract`	The method is not implemented. The class must be extended and the method must be implemented in the subclass.
`final`	The method cannot be overridden in a subclass.
`native`	The method is implemented in another language.
`synchronized`	The method requires that a monitor (lock) be obtained by calling code before the method is executed.
`throws`	A list of exceptions thrown from this method.

3.3 A Simple Class Example



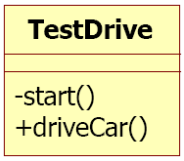
The following is an example of simple class called Car.

```
import java.awt.Color;
  /**
    *  Represents a car.
    *  The attributes are speed, engine power and color.
    *  The methods are accelerate,
    *  decelerate, getSpeed, getColor, getPower,
    *  getAcceleration, and getMaxSpeed.
    *
    * @author Laboratory Team
    */
   public class Car
   {
      private String brandName;
      private int speed = 0;  // current car speed
      private Color color;
      private int power;
      private int accelerationStep=0; // speed increase when gas pedal pushed
      private int maxSpeed;           // maximum speed for this car
      private static final int MIN_SPEED = 0; // minimum speed for all cars; km/h
      private static final int MAX_SPEED = 300; // speed limit on all cars; km/h
      private static final int MIN_POWER = 4; // minimum power for all cars; no less
than 4 bhp
      private static final int MAX_POWER = 500; // maximum power for all cars.no more
than 500 bhp
      private static final int ACCELERATION_MIN_STEP = 1; // minimum speed increase
per second when gas pedal pushed
private static final int ACCELERATION_MAX_STEP = 30; // maximum speed increase per
second
//  when gas pedal pushed
      /**
       * Constructor to create a new Car object
       * @param brand name or manufacturer.
       * @param color color of the Car object - one of Java color constants
       * @param power engine power
       * @param accelerationStep increase in speed when gas pedal pushed
       */
      public Car(String brand, Color color, int power, int maxSpeed, int
accelerationStep)
      {
         //creates a new Car object with specified brand name, color, engine power,
and maximum speed
         this.brandName = brand;
         this.color = color;
         this.power = (power > 4)? power: 4;
         if (maxSpeed < 0) this.maxSpeed = MIN_SPEED; // this car wil never move
         else
```

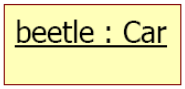
```
        if (maxSpeed < MAX_SPEED) this.maxSpeed = maxSpeed;
        else maxSpeed = MAX_SPEED;
        if (power < MIN_POWER) this.power = MIN_POWER;
        else
        if (power > MAX_POWER) this.power=MAX_POWER;
        if (accelerationStep < ACCELERATION_MIN_STEP)
           this.accelerationStep = ACCELERATION_MIN_STEP;
        else
        if (accelerationStep > ACCELERATION_MAX_STEP)
           this.accelerationStep = ACCELERATION_MIN_STEP;
        else
           this.accelerationStep = accelerationStep;
    }

    /**
     * Simulates pressing the accelerator.
     * @return the new speed
     */
    public int accelerate()
    {
        int newSpeed = speed + getAcceleration();
        if(newSpeed <= getMaxSpeed())
        {
           speed = newSpeed;
        }
        else
        {
           speed = getMaxSpeed();
        }
        return speed;
    }
    /**
     * Simulates releasing the accelerator.
     * @return the new speed
     */
    public int decelerate()
    {
        if(speed > MIN_SPEED)
        {
           speed--;
        }
        return speed;
    }

    /**
     * @return the current speed
     */
    public int getSpeed()
    {
        return speed;
    }
    /**
     * @return the max speed
     */
    public int getMaxSpeed()
    {
        return MAX_SPEED;
    }
    /**
     * @return the color of the car
     */
```

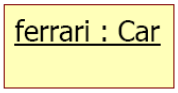
```
    public Color getColor()
    {
       return color;
    }
     /**
      * @return the brand name of the car
      */
    public String getBrandName()
    {
       return brandName;
    }

    /**
     * @return the car's acceleration step
     */
    public int getAcceleration()
    {
       return accelerationStep;
    }
    /**
     * @return the power of the car
     */
    public int getPower()
    {
       return power;
    }
 }
```

The following class implements a simple test drive:




```
import java.awt.Color;
  /**
  * TestDrive demonstrates creating and calling
  * methods on Car  object.
  *
  * @author Laboratory Team
  */
   public class TestDrive
   {
      //The Java virtual machine (JVM) always starts
      //execution with the 'main' method of the class passed
      //as a argument to the java command
      public static void main(String []args)
      {
         TestDrive td = new TestDrive();
         td.start();
         //exit TestDrive
      }
      private void start()
      {
         //Create a Volkswagen beetle Car
         Car beetle = new Car("Volskwagen Beetle", Color.orange, 80, 160, 10);
         //Take it for a drive
         System.out.println("Starting beetle test drive!");
         driveCar(beetle);

         //Create a Ferrari
         Car ferrari = new Car("Ferrari Testarosa", Color.red, 300, 280, 30);
         //Take it for a drive
         System.out.println("Starting ferrari test drive!");
         driveCar(ferrari);
```





```
    }

    public static void driveCar(Car c)
    {
       System.out.print("Car is a " + c.getBrandName());
       System.out.println(" colored  " + c.getColor());
       System.out.print("\t engine power is " + c.getPower());
       System.out.println(" speeding step is " +c.getAcceleration());
       //press the accelerator 15 "times"
       for(int i = 0; i < 15; i++) {
          System.out.println("accelerating: " + c.accelerate());
       }
       //release the accelerator 5 "times"
       for(int i = 0; i < 5; i++) {
          ;
          System.out.println("decelerating: " + c.decelerate());
       }
       System.out.println("final cruising speed: " + c.getSpeed());
    }
}
```

4 Lab Tasks

4.1. Study and understand the given classes. Then test drive two cars. To do this, create a BlueJ project with the two classes, compile and run it. Note what is printed for a car's color. Why?

4.2. Add new cars to the test drive, and play with tem too.

4.3. Change the Car class by adding extra features like breaking and changing gears (gears must be changed at certain speed limits to keep engine working properly).

4.4. Add a fuel tank and capacity to the Car class and simulate fuel consumption. Consider fuel consumption varies according to preset rules depending on the speed.