Classes and methods

Defining classes

- A class is a template from which objects are created. It is defined using the **class** keyword.

  Example:

```
class Ticker {
    // body of the class
}
```

- All classes inherit from the **Object** class unless a superclass is specified.

  Example:

```
class SportsTicker extends Ticker {
    // body of the class
}
```

Instance variables

- Instance variables (fields) are defined outside a method definition and are not modified using the static keyword.

  Example:

```
public class VolcanoRobot {
    String status;
    int speed;
    float temperature;
}
```

Class variables

- Class variables related to the class as a whole, not to a particular object.

  Examples:

  - `static int sum;`
  - `static final int maxObjects = 10;`

Methods

- Methods define the behavior of the objects created from the class.
- A method has four mandatory parts:
  - The name of the method
  - A list of parameters
  - The type of object or primitive type returned by the method
  - The body of the method

Methods

- Structure:

```
returnType methodName(type1 arg1, type2 arg2, type3 arg3 ...) {
    // body of the method
}
```

- The first two parts form what's called the method's **signature**. All methods must have a unique signature within a class.
- The return type can be any object, primitive type or an array of objects or primitive types. If the method doesn't return anything, the **void** keyword is used.

Example: RangeLister

The “this” keyword

- The **“this”** keyword is used in methods to refer to the object the method belongs to therefore accessing his variables and methods.

  Examples:

  - `t = this.x; // the x instance variable for this object`
  - `this.resetData(this); // call the resetData method, defined in this class, and pass it the current object`
  - `return this; // return the current object`

- It's usually not necessary to do this explicitly, it's assumed.

  Examples:

  - `t = x;`
  - `resetData(this);`

Variable scope

- Scope is the part of the program in which the variable or another type of information exists. When the part defining the scope has completed execution, the variable ceases to exist.
  - A **local variable** exists inside the method or block in which it's defined.
  - An **instance variable** has a scope that extends to the entire class, so they can be used by any of the instance methods within that class.
  - A **class variable** exists at all time during which the program runs.

Example: ScopeTest

Passing arguments to methods

- When used as arguments during a method invocation, objects are passed **by reference**. Any change to the object made inside the method will be visible outside the method. This also includes objects or primitive types stored in an array.
- Primitive types on the other hand as passed **by value**. Upon returning from the method, the original value before the method was invoked is kept

Example: Passer

Class methods

- A class method does not require an object instance in order to be invoked. It can be invoked by the class in which it's defined or any other class.

  Examples:

  - `System.exit(0);`
  - `int now = System.currentTimeMillis();`
  - `int count = Integer.parseInt("42");`
- Class methods are defined using the static keyword.

  Example:

```
static void exit(int arg1) {
    // body of the method
}
```

Java applications

- A Java application is a collection of Java classes which can run as a program.
- A Java application requires one class that serves as a starting point. This class has to have a **main** method which has the following form:

```
public static void main(String[] arguments) {
    // body of method
}
```

Passing arguments to Java applications

- To pass arguments to a Java program with the java interpreter included with the JDK, the arguments should be appended to the command line when the program is run.

  Examples:

  - `java EchoArgs April 450 -10`
  - `java EchoArgs Wilhelm Niekro Hough “Tim Wakefield” 49`

Example: Averager

Overloading methods

- Two things differentiate methods with the same name:
  - The number of arguments they take
  - The data type or objects for each argument
- These two form the method's signature. Using several methods with the same name but different signatures is called **overloading**.

Example: Box

Constructors

- When creating a new instance of a class, Java does three things:
  - Allocates memory for the object.
  - Initializes that object's instance variables, either to initial values or to a default (0 for numbers, null for objects, false for Booleans, or '\0' for characters).
  - Calls the constructor method of the class, which might be one of several methods.
- A **constructor method** is a method called on an object when it is created.
- Differences between constructors and regular methods:
  - They always have the same name as the class.
  - They don't have a return type.
  - They cannot return a value in the method by using the return statement.

Constructors

Example:

```
class VolcanoRobot {
    String status;
    int speed;
    int power;

    VolcanoRobot(String in1, int in2, int in3) {
        status = in1;
        speed = in2;
        power = in3;
    }
}

// …

VolcanoRobot vic = new VolcanoRobot(“exploring”, 5, 200);
```

Calling another constructor method

- If a constructor method duplicates the behavior of an existing constructor method, the first one can call the second one from within its body.

  Example:

```
class Circle {
    int x, y, radius;

    Circle(int xPoint, int yPoint, int radiusLength) {
        this.x = xPoint;
        this.y = yPoint;
        this.radius = radiusLength;
    }

    Circle(int xPoint, int yPoint) {
        this(xPoint, yPoint, 1);
    }
}
```

Overloading constructor methods

- Constructor methods can be overloaded much the same as other regular methods.

Example: Box2

Overriding methods

- Creating a method in a subclass with the same signature (name and argument list) as a method defined in that class's superclass is called **method overriding**.
- There are usually two reasons for method overriding:
  - To replace the definition of that original method completely
  - To augment the original method with additional behavior

Example: Printer

Calling the original method

- The **super** keyword can be used to call the original method from inside a method definition which passes the call up the hierarchy.

  Example:

```
void doMethod(String a, String b) {
    // do stuff here
    super.doMethod(a, b);
    // do more stuff here
}
```

Overriding constructors

- When a class's constructor method is called, the default constructor method of the superclass is also called implicitly.
- In case another constructor method of the of the parent class needs to be invoked, it can be done so explicitly using the super keyword.

  Example:

  ```
  super(arg1, arg2, ...);
  ```

- This needs to be the first statement in a constructor definition.

Example: NamedPoint

Finalizer methods

- In many ways opposite to a constructor, a **finalizer** method is called just before the object is removed by the garbage collector allowing you to do some "clean up" work if necessary.
- Override the default finalize method provided by Object:

  Example:

```
protected void finalize() throws Throwable {
    super.finalize();
}
```

Exercises

Exercise: Numbers

- Create a class that takes words for the first 10 numbers (“zero” up to “nine”) and converts them into a single integer. Use a switch statement for the conversion and command-line arguments for the words.

  Example:

  `"java Numbers one five zero“` should print out `"150"`