CM1209/CM1210

Object Oriented Java Programming

SESSION 2 [TEACHING WEEK 2]

A CLASS ON CLASSES

Example

From Last Week:

Your task was to construct an application that will determine the average of a list of marks entered by the user.

You've been provided the following initial psuedocode:

```
Loop:
    Input next mark
    Update total
End loop
Calculate average = total / number of marks
Output average
```

** TASK REQUIREMENTS ARE ON THE NEXT SLIDE **

Requirements

You are required to construct three different versions of the given application as follows:

1. calculates an average (using `for`)
2. calculates an average (using `while`)
3. calculates an average (using `do...while`)

NOTE: pseudocode specific to each of these applications/conditional statements is provided on the next three slides.

Requirements

Average.java

Calculating an average (using *for*)

Psuedocode:

```
Input number of marks
For i = 0 to number of marks - 1
    Input mark
    Update total
Calculate average = total / number of marks
Output average
```

Example solution:

See **Average.java** in **average_progs.zip**

Requirements

Average2.java

Calculating an average (using ***while***)

Psuedocode:

```
Input mark
While last mark input != -1
    Update total
    Input mark
Calculate average = total / number of marks
Output average
```

Example solution:

See **Average2.java** in **average_progs.zip**

Requirements

Average3.java

Calculating an average (using ***do...while***)

Psuedocode:

```
Do
    Input mark
    Update total
    Calculate average = total / number of
marks
Output average
While user has more data
Output final average
```

Example solution:

See **Average3.java** in **average_progs.zip**

Formatted output (using `printf`)

The easiest way to format output (e.g. to limit decimal places) in Java is to use the `printf` method. This method takes an arbitrary number of arguments, with the first specifying the format to be used. For example:

```
System.out.printf("%.3f%n", 1/3f);
```

will output the 2nd argument (i.e. `1/3f`) as a floating point number rounded to 3 decimal places (all shown), followed by a new line.

General Form

```
%[argumentIndex$][flags][width][.precision]conversion
```

- The optional `argumentIndex` is a decimal integer indicating the position of the argument in the argument list. The first argument is referenced by `1$`, the second by `2$`, etc.
- The optional `flags` is a set of characters that modify the output format. The set of valid flags depends on the conversion.
- The optional `width` is a non-negative decimal integer indicating the minimum number of characters to be written to the output.
- The optional `precision` is a non-negative decimal integer usually used to restrict the number of characters. The specific behavior depends on the conversion.
- The **`required`** conversion is a character indicating how the argument should be formatted. The set of valid conversions for a given argument depends on the argument's data type. The most common are `%d` – decimal integer, `%f` – decimal floating point, `%e` – decimal floating point in scientific notation, `%o` – octal integer and `%x` – hexadecimal integer. `%n` denotes a newline.

Formatted output

EXAMPLE: PrintfExample.java

Some Examples:

```
1   System.out.printf("%.2f", Math.PI);

2   System.out.printf("%.4f", Math.PI);

3   System.out.printf("%.4e", 1234.5678);

4   System.out.printf("%4d", 12);

5   System.out.printf("%2$d %1$d", 48, 47);

6   System.out.printf("%2$x %1$o", 48, 47);
```

OOP

Object-oriented programming (OOP)

- Objects (or more precisely the classes objects come from), are reusable software components.
- There are time objects, video objects, etc....
- Almost any noun can be reasonably represented as a software object in terms of:
  - attributes (e.g. name, colour, size, etc...)
  - behaviours (e.g. calculating, moving, communicating, etc..)
- Software developers (particularly groups) can be more productive and programs are often easier to understand, correct and modify.

Car Example

- *Your goal*: drive a car and make it go faster by pressing the accelerator pedal.
- *To achieve this*: Someone must ***design*** it (typically starts with engineering drawings or ***blueprints***).
- The drawing will include the design for the accelerator pedal.
- The actual pedal, however, ***hides*** the complex mechanisms that actually make the car go faster from the driver.
- This enables anyone with little or no knowledge of how engines work to easily drive a car.

Car Example

- However, you cannot drive a cars blueprints!!
- The car must be ***built***, using the blueprints that define it, before it can be driven.
- The completed car has an ***actual*** accelerator pedal, which makes the car go faster.
- But even this is not enough – it won't accelerate on its own.
- It requires a driver to ***press*** the pedal to get the car to accelerate.

Lets use the car example to introduce some OOP concepts

Car Example

Methods and Classes

- Performing a task in a program requires a **method**.
- The method houses the program statements that actually perform its tasks.
- The method *hides* these statements from its user, just as the car hides from the driver the mechanisms that it employs to make the car go faster.
- In Java, we create a program unit called a **class** to house the set of methods that perform the class's tasks.
- A class is similar in concept to a car's blueprint, which contain the design of the accelerator pedal, steering wheel, etc...

Car Example

Some other class/methods examples

- A ***class*** that represents a **bank account**, might have ***methods*** to *withdraw* and *deposit* money, also to check the *balance*.
- A ***class*** that represents a **door**, might have ***methods*** to *lock*, *unlock*, *open* and *close* it.
- A ***class*** that represents a **phone directory**, might have ***methods*** to insert, delete, edit, retrieve and search entries within it.

Car Example

Instantiation

- We stated earlier that we need to **build** the car from its blueprints, before it could actually be driven.
- Before a program is able to perform the tasks that the class's methods define, we must **build** an object to represent that class.
- This process is known as **instantiation**
- The object is then referred to as an **instance** of its class.

Car Example

Reuse

- The blueprints for the car can be **reused** many times to build many cars.
- Similarly, you can **reuse** a class many times to build many objects.
- Reuse of existing classes when building new classes, saves time and effort.
- It also helps to build more reliable and effective programs, as existing classes will have gone through extensive *testing*, *debugging* and *performance* tuning.
- Reusable classes are a crucial part of software development.

Car Example

Messages and Method Calls

- When you drive a car, pressing the accelerator pedal sends it a **message** to perform a task – accelerate.
- In a similar fashion, you can **send messages to objects**.
- Each message is implemented as a **method call** that tells a method of the object to perform its task.

For example: a program might call a particular bank account's *deposit* method to increase that account's balance.

Car Example

Attributes and Instance Variables

- In addition to having the ability to accomplish tasks, a car also has **attributes** (e.g. colour, number of doors, current speed, etc....)
- Like its capabilities, the car's attributes are represented as part of its design in its blueprint.
- As you drive an actual car, these attributes are carried along with the car.
- Every car maintains its **own** attributes.
- E.g. each car knows how much fuel is in its tank, but **not** how much is in the tanks of **other** cars.

Car Example

Attributes and Instance Variables

- Similarly, an object has attributes that it carries along as its used in a program.
- These attributes are defined as part of the object's class.
- Attributes are defined by the class's **instance variables**.
- E.g. a bank account object has a **balance attribute** to represent the amount of money in the account.
- Each bank account object knows the balance in the account it represents, but **not** the balances of **other** accounts in the bank.

Car Example

Encapsulation

- Classes **encapsulate** (i.e. wrap) attributes and methods into objects
  - An object's attributes and methods are intimately related.
- Objects may communicate with one another, but they're normally not allowed to know how other objects have been implemented.
  - Implementation details are **hidden** within the the objects themselves.
- This **information hiding**, as we'll see later one, is crucial to good software engineering.

Car Example

Inheritance

- A new class of objects can be created quickly and conveniently by **inheritance**.
  - The new class **absorbs** the characteristics of the existing class, possibly customising them or adding unique characteristics of its own.
- Consider our car, an object of class ‘convertible’ **is certainly an** object of the more **general** class ‘automobile’, but more **specifically**, the roof can be lowered and raised.

We will come back to look at OOP later

Constructing objects

- As outlined in the car analogy, objects must be **declared** before they can be used.
- Objects are initialised using the **new** operator.
- The new operator calls the **appropriate constructor** for the class. (**NOTE:** We'll come back to cover constructors in a little while).

EXAMPLE: HelloDate.java

Constructing objects

```
Date d = new Date();
System.out.println( "The current date: " );
System.out.println( d );

d = new Date( -100000 );
System.out.println( "Another date: " + d );

System.out.println( "The current date: " +
new Date() );
```

Accessing fields and methods

- Fields and methods are accessed via the **dot operator**

```
Point p = new Point( 10, 20 );
System.out.println(p.y);
p.x = 20;
p.translate(20,10);
```

NOTE: For this code to work you would need to import java.awt.Point;

- In the last line, p is called the ***implicit parameter*** of the method call
- 20, 10 are the ***explicit parameters*** of the method call

References

- **Object variables** do not store the actual value or instance directly in their memory location (unlike **primitive type variables** such as **int**).
- Instead they store a reference to the memory location of the object.
- There can be **more than one reference** to a single object.

EXAMPLE: ReferenceVariableExample.java

Multiple References:

```
Point a = new Point( 10, 20 );
Point b = a;
System.out.println( "a:" + a + " b:" + b );
System.out.println( "Excecuting a.translate(
10, 10 );" );
a.translate( 10, 10 );
System.out.println( "a:" + a + " b:" + b );
System.out.println( "Excecuting b.translate(
10, 10 );" );
b.translate( 10, 10 );
System.out.println( "a:" + a + " b:" + b );
```

Wrapper classes

Wrapper classes

- ***Wrapper classes*** allow objects to be created from primitive types e.g:

```
Integer i = new Integer(57);
```

- These classes also contain some useful methods for working with primitive type variables
- Placing a primitive type within an object is referred to as ***boxing***
- Java 1.5 added a feature called ***autoboxing*** where primitive type variables are ***automatically*** converted to an object of the equivalent type-wrapper class.

Autoboxing

EXAMPLE: AutoboxingExample.java

Autoboxing:

```
int x = 3;
Integer y = x + 5;
System.out.println("y = " + y);

System.out.println("y == 8 : " + (y == 8));

Integer z = new Integer(8);
System.out.println("y == z : " + (y == z));

// Use the intValue() method of class Integer to obtain
the int value in the Integer object

int v = z.intValue();
System.out.println("y == v : " + (y == v));
```

EXAMPLE: EqualsTest.java

Comparing objects

```
Integer a = new Integer( 57 );
Integer b = new Integer( 57 );
Integer c = a;

System.out.println( (a == b) );
System.out.println( a.equals(b) );
System.out.println( (a == c) );
System.out.println( a.equals(c) );
System.out.println( (b == c) );

System.out.println( b.equals(c) );
```

NOTE:

- == on primitive types tests for equivalence of ***values***
- == on “object variables” tests for equivalence of ***references***
- .equals on “object variables” tests for equivalence of ***values***

Arrays

EXAMPLE: ArrayExample.java

Definition

An array is a data structure consisting of a set of related data items of the same type referred to as ***elements*** of the array.

```
int[] a = { 1, 2, 3, 4, 5 };
for (int i = 0; i < a.length; ++i) {
    System.out.print( a[i] + " " );
}
System.out.println();

float[] b = new float[5];
for (int i = 0; i < b.length; ++i) {
    b[i] = (float) i / 5;
}
```

Command line arguments

Every Java application includes a method:

```
public static void main( String[] args )
```

The parameter **`args`** is an array of String objects containing the ***`command line arguments`*** passed to the method/application by the operating system.

For example, given a Java application **`Foo`**, executing the following at the command line:

```
java Foo bar1 bar2 bar3
```

will pass the arguments `bar1`, `bar2`, `bar3` to the `main` method as an array of 3 elements named `args`.

Garbage Collection

- In Java we explicitly create objects with new but there is **no need** to explicitly free their memory when they are no longer needed .
- The **garbage collector** periodically frees the memory of objects that are no longer in use .
- Objects are in use if they can be accessed by the program in its current state, i.e. there is a reference to the object.

Adding Two Numbers

EXAMPLE: EasyAdd.java

```
// EasyAdd.java
// Adds two specified integers.
import java.util.Scanner;

public class EasyAdd {
   public static void main( String args[] ) {
      Scanner in = new Scanner(System.in);
      System.out.print("Enter first number: ");
      int x = in.nextInt();
      System.out.print("Enter second number: ");
      int y = in.nextInt();
      int sum = x + y;
      System.out.println( "Their sum is " + sum);
   }
}
```

Application to add together 2 integers, requested from the user by using the Scanner class. **TASK:** Lets now re-write our application to create a method that will get integer values using the Scanner Class.

Methods

Definition

A set of statements to solve some subproblem should be coded as a ***method***.

```
// Example method to count sheep
public static int countSheep( int n ) {
    int sheepCount = n * n;
    return sheepCount;
}
```

static methods

Example – Method No Return [void]

```
public static void outputInt( String s, int n ) {
    System.out.println(s + " " + n);
}
```

Example – Method With Return [int]

```
public static int squareNumber( int n ) {
    int result = n * n;
    return result;
}
```

```
public class Foo {
    public static void main(...) {
        bar();
    }
    public static void bar() { ... }
}
```

```
public class Bob {
    public static void main(...) {
        Foo.bar();
    }
}
```

static methods

- At first, the methods we write will be static.
- Static methods are methods that do not operate on objects.
- For example, the `pow` method of the `Math` class is a `static` method. The expression:

```
Math.pow(x,a);
```

- computes the power x^a, but you do not need to instantiate a Math object to use the method. i.e.

```
Math myMath = new Math();
myMath.pow(x,a);
```

NOT NEEDED

`static` method – general form:

```
public static returnType methodName( parameter_list ) {
    // Method body
}
```

Parameter list (comma separated)

- Each parameter consists of a **type** and **identifier**.

Definition

- ***explicit parameters***: parameters in the method declaration.
- ***arguments***: parameters supplied when the method is called.

Parameters and scope

EXAMPLE: MethodScope.java

Remember -> Each parameter is only in scope in the corresponding method body.

```
public static void main( String[] args ) {
      squareNumber(5);
      System.out.println(result);
}


public static int squareNumber( int n ) {
      int result = n * n;
      return result;
}
```

OUT OF SCOPE

Adding Two Numbers

EXAMPLE: SimpleAddWithMethod.java

```
// SimpleAddWithMethod.java
// Adds two specified integers using a getInt()
method.
import java.util.Scanner;

public class SimpleAddWithMethod {
    public static void main( String args[] ) {
        int x = getInt("Enter first integer");
        int y = getInt("Enter second integer");
        int sum = x + y;
        System.out.println( "Their sum is " + sum);
    }

    public static int getInt( String prompt ) {
        Scanner in = new Scanner(System.in);
        System.out.print( prompt + " : " );
        int result = in.nextInt();
        return result;
    }
}
```

METHOD CALLS

METHOD

NOTE: Same functionality, but this time using a **method**

EXAMPLE: PrimitiveArguments.java

Passing arguments (primitive types)

Primitive type variables are **passed** using ***call by value*** – a **copy** of the variable is passed to the method.

```
public static void main( String[] args ) {
     int i = 0;
     System.out.println( "Before someMethod i = " + i );
     someMethod( i );
     System.out.println( "After someMethod i = " + i );
}

static void someMethod( int a ) {
     System.out.println( " In someMethod a = " + a );
     i++;
     System.out.println( " In someMethod a = " + a );
}
```

Passing arguments

EXAMPLE: ReferenceArguments.java

Passing arguments (Objects)

- **Objects** are **never passed** as **arguments** to **methods**
- References to objects are passed using ***call by reference*** *value*

```
public static void main( String[] args ) {
     java.awt.Point p = new java.awt.Point( 20, 20 );
     System.out.println( "Before someMethod p = " + p );
     someMethod( p );
     System.out.println( "After someMethod p = " + p );
     someOtherMethod( p );
     System.out.println( "After someOtherMethod p = " + p );
}
```

```
static void someMethod( java.awt.Point q ) {
     System.out.println( " On entering someMethod q = " + q );
     q.translate( 10, 10 );
     System.out.println( " On leaving someMethod q = " + q );
}
```

EXAMPLE: ReferenceArguments.java

Passing arguments (Objects)

```
public static void main( String[] args ) {
     java.awt.Point p = new java.awt.Point( 20, 20 );
     System.out.println( "Before someMethod p = " + p );
     someMethod( p );
     System.out.println( "After someMethod p = " + p );
     someOtherMethod( p );
     System.out.println( "After someOtherMethod p = " + p );
}
```

```
static void someOtherMethod( java.awt.Point q ) {
     System.out.println( " On entering someOtherMethod q = " + q );
     q = new java.awt.Point( 50, 50 );
     System.out.println( " In someOtherMethod q = " + q );
     q.translate( 10, 10 );
     System.out.println( " On leaving someOtherMethod q = " + q );
}
```

Group related methods

Lets go back to our adding application

```
public static void main( String args[] ) {
    int x = getInt("Enter first integer");
    int y = getInt("Enter second integer");
    int sum = x + y;
    System.out.println( "Their sum is " + sum);
}

public static int getInt( String prompt ) {
    Scanner in = new Scanner(System.in);
    System.out.print( prompt + " : " );
    int result = in.nextInt();
    return result;
}
```

- Why copy and paste this **method** into every class that needs it!!
- Lets create a ***class*** that can be easily reused.

Input Class

```
import java.util.Scanner;

public class Input {
    // Prompted input of an int
    public static int getInt( String prompt ) {
        Scanner in = new Scanner(System.in);
        System.out.print( prompt + " : " );
        int result = in.nextInt();
        return result;
    }
    // Prompted input of an float
    public static float getFloat( String prompt ) {
        Scanner in = new Scanner(System.in);
        System.out.print( prompt + " : " );
        float result = in.nextFloat();
        return result;
    }
    // Prompted input of an word/token
    public static String getToken( String prompt ) {
        Scanner in = new Scanner(System.in);
        System.out.print( prompt + " : " );
        String result = in.next();
        return result;
    }
}
```

Using our new class

Lets now write an application to make use of your new class:

```
public class SimplerAdd {
    public static void main( String args[] ) {
        int x = Input.getInt("Enter first integer");
        int y = Input.getInt("Enter second integer");
        int sum = x + y;
        System.out.println( "Their sum is " + sum);
    }
}
```

But the compiler and JVM must be able to find the **Input** class (i.e. the .class file)

(Note: it will automatically find files that are in the **same folder** as the **.java file**)

NOTE: We will explore classpath and packages later.

`static` vs `instance` methods

As outlined, a **`static`** method is special, you can call it without first creating an object of the class in which the method is declared.

The **`sqrt`** method in the class **`Math`** is **`static`**. This method does not need to be invoked on a **`Math`** object, instead we just write:

```
double d = Math.sqrt(25);
```

static method

The method **`translate`** in the **`Point`** class is **not static** – it **must** be invoked on a **`Point`** object

```
Point p = new Point(0,0);
p.translate(10,33);
```

instance method

Input class

Improving our `Input` class

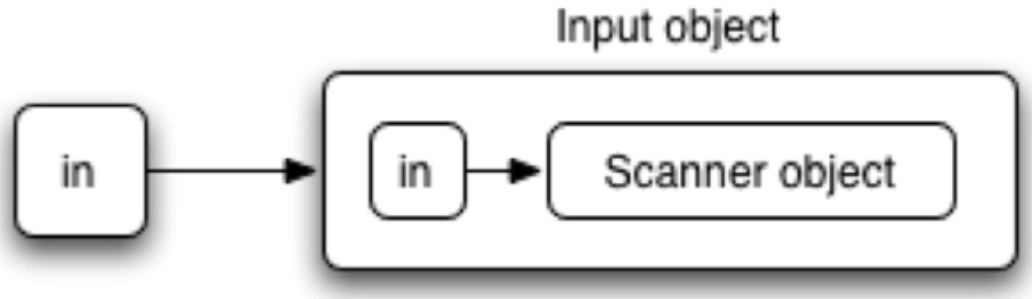
- Each method call creates a new **`Scanner`** object
- **Adding** a **`field`** and a **`constructor`** to the class, we can avoid this and **remove** the Scanner from each method call:

```
public class Input {
    Scanner in; // Instance Variable

    // Constructor
    public Input() {
        in = new Scanner(System.in);
    }

    public static int getInt( String prompt ) {
        Scanner in = new Scanner(System.in);
        System.out.print( prompt + " : " );
        int result = in.nextInt();
        return result; }
}
```

Remove **static** and becomes **instance** method

Improving our `Input` class

- The **methods** of the class are **no longer `static`**.
- Each object of our **`Input`** class is ***composed*** of a single **`Scanner`** object.



New Input class

EXAMPLE: Input.java

```
import java.util.Scanner;

public class Input {
   Scanner in;

   // Constructor
   public Input() {
      in = new Scanner(System.in);
   }

   // Prompted input of an int
   public int getInt( String prompt ) {
      System.out.print( prompt + " : " );
      int result = in.nextInt();
      return result;
   }

   // Prompted input of a float
   public float getFloat( String prompt ) {
      System.out.print( prompt + " : " );
      float result = in.nextFloat();
      return result;
   }

   // Prompted input of a word/token
   public String getToken( String prompt ) {
      System.out.print( prompt + " : " );
      String result = in.next();
      return result;
   }
}
```

Method overloading

EXAMPLES: Output.java

```
public class Output {
  public static void printArray( int[] a ) {
     for ( int i = 0; i < a.length; ++i {
        System.out.print( a[i] );
        if ( i < a.length - 1 )
           System.out.print( ", " );
        }
        System.out.println(); }

  public static void printArray( String[] a )
{
     for ( int i = 0; i < a.length; ++i ) {
        System.out.print( a[i] );
        if ( i < a.length - 1 )
           System.out.print( ", " );
        }
        System.out.println();}
}
```

Using the new Input class

EXAMPLE: SimplerAdd.java

```
public class SimplerAdd {
     public static void main( String args[] ) {
          Input in = new Input();
          int x = in.getInt("Enter first integer");
          int y = in.getInt("Enter second integer");
          int sum = x + y;
          System.out.println( "Their sum is " + sum);
     }
}
```

Method overloading

- Note that both methods on the previous page have **identical names** but **differ** in their **arguments**:

```
public static void printArray( int[] a )

public static void printArray( String[] a )
```

- A method is **overloaded** if the class containing it has at least **one other method** with the **same name** (but with different parameter types).
- Any number of methods can be defined with the same name as long as each overloaded method takes a **unique list of argument** types.

Worked example

Worked example: writing classes

Problem

- Create a phone book application that stores names and numbers, allows a user to add entries and retrieve the number for a given name, and load/save the data to file.

Classes

- PhoneBookEntry:
- PhoneBook:

Worked example

`PhoneBookEntry` Implementation

```
public class PhoneBookEntry {
    String name; // instance variables
    String number; // to store data
}
```

1. We want to ensure that every entry has a name and a number - **CONSTRUCTORS**
2. We'd like to control how code accesses the name and number – **ACCESS SPECIFIERS**
3. We'd like to make it easy to output `PhoneBookEntry` objects – **OVERRIDE toString()**

Worked example

Constructors (definition)

```
public class MyClass {
    public MyClass() {
        // Initialisation code here
    }
}
```

Remember:

A **constructor** is a special method that allows you to control how objects are instantiated.

Worked example

Constructors (PhoneBookEntry)

```
public class PhoneBookEntry {
   public PhoneBookEntry( String inName, String inNumber ) {
        name = inName;
        number = inNumber;
    }
}
```

OR

```
public class PhoneBookEntry {
   public PhoneBookEntry( String name, String number ) {
        this.name = name;
        this.number = number;
    }
}
```

`this` refers instance variables

Notes on constructors:

- Constructors **do not** have a **return value** specified.
- **Constructors** must have the **exact same name** as the **class**.
- If you create a constructor that takes arguments, then you cannot use a default constructor unless you explicitly define one.

Worked example

Access specifiers – `public`

The **`public`** keyword means that the following member or class declaration is available to everyone.

MyClass.java

```
public class MyClass {
    public void someMethod() {
        // Implementation
    }
}
```

MyOtherClass.java

```
public class MyOtherClass {
    public void someOtherMethod() {
        MyClass m = new MyClass();
        m.someMethod();
    }
}
```

Worked example

Access specifiers – `private`

The **`private`** keyword means that no one has access to the method except inside others of that particular class.

MyClass.java

```
public class MyClass {
    private int myField;
    private void someMethod() {
        // Implementation.
    }
    public void otherMethod() {
        System.out.prinltn(myField);
        someMethod();
    }
}
```

MyOtherClass.java

```
public class MyOtherClass {
    private void print() {
        MyClass m = new MyClass();
        m.someMethod();
        System.out.prinltn(myField);
    }
}
```

Worked example

Access specifiers – `protected` and package access

Definition (**`protected`**)

- The `protected` keyword means that access is only allowed from within the classes that extend the particular class.

Definition (**package access**)

- If no access specifier is included then the member is only available to other classes in the same package. This is sometimes termed as *friendly* access.

Worked example

Access specifiers – Accessors

private data

```
public class PhoneBookEntry {
    private String name;
    private String number;
}
```

Accessor

```
public String getName() {
    return name;
}
```

An ***accessor*** method allows ***read access*** to a private field (instance variable).

Worked example

Access specifiers – `Mutators`

private data

```
public class PhoneBookEntry {
    private String name;
    private String number;
}
```

Accessor

```
public String setName() {
    name = “matt”;
}
```

An ***mutator*** method allows ***write access*** to a private field (instance variable).

Worked example

`String` and `StringBuffer` classes

- As a quick aside, lets introduce the StringBuffer classes.
- String objects are *immutable* (i.e. they cannot be modified) whereas StringBuffer objects are *mutable* and can be modified.

String

```
String s = “abc”;
s = s.toUpperCase();
s.toLowerCase();
```

StringBuffer

```
StringBuffer sb = new StringBuffer(“ABC”);
sb.append(“DEF”);
```

Worked example

Easy output of the `PhoneBookEntry` class

- We want a way to output an entry to standard output.
- We can achieve this, by overriding the toString() method.
- We'll come back a little later on to look at exactly what is going on here.
- If we add the following method:

```
public String toString() {
    String s = name + "\t(" + number + ") ";
    return s;
}
```

- We can then print phonebook entries.

Worked example

PhoneBookEntry.java

Finished PhoneBookEntry application

```
class PhoneBookEntry {

    private String name;
    private String number;

    public PhoneBookEntry( String inName, String inNumber ) {
        name = inName;
        number = inNumber;
    }

    public String getName( ) {
        return name;
    }

    public String getNumber( ) {
        return number;
    }

    public void setNumber( String inNumber ) {
        number = inNumber;
    }

    public String toString( ) {
        String s = name + "\t(" + number + ") ";
        return s;
    }
}
```

Worked example

PhoneBookEntryTest.java

PhoneBookEntryTest application

Lets now write an application to use our new PhoneBookEntry class and test our code:

```
public class PhoneBookEntryTest {
    public static void main( String[] args ) {
        PhoneBookEntry e = new PhoneBookEntry( "Bob", "+44
          0123456789" );
        System.out.println( e ); }
    }
}
```

Anatomy of a Java class

JavaClass.java

```
public class JavaClass {

    private final String name;
    private String description;
    private static int staticInstanceVariable = 0;
    private int instanceVariable = 1;

    public JavaClass( String id, String description ){
            name = id;
            this.description = description;
    }

    public static void staticMethod() {
            System.out.println("JavaClass static method.");
    }

    public static int staticMethodWithReturn() {
            return staticInstanceVariable;
    }
    public void instanceMethod() {
            System.out.println("JavaClass instance method.");
    }

    public int instanceMethodWithReturn() {
            return instanceVariable;
    }

    public String toString() {
            return name + " -> " + description;
    }

    public static void main( String[] args ) {

            // Using static methods

            JavaClass.staticMethod();
            staticMethod();
            System.out.println( JavaClass.staticMethodWithReturn() );
            System.out.println( staticMethodWithReturn() );

            // Using instance methods - need to call from instantiated object

            JavaClass jc = new JavaClass("matts_class", "Anatomy of a Java Class");
            jc.instanceMethod();
            System.out.println( jc.instanceMethodWithReturn() );
            System.out.println( jc );

    }

}
```

class name

this refers to instance variable, without this it refers to local variable

instance variables

constructor

static methods

note static variable in static method

instance methods

override toString()

instantiate new JavaClass object

invoke constructor

use dot operator to access methods of JavaClass object

automatically invoke toString() method

Feedback

Are there things you like?, Things you don't like?, Suggestions of any kind? I really would like to hear from you.

Let me know anonymously at :
https://www.suggestionox.com/r/Azaz5o