



Introduction to Programming

15. Types, Classes and Objects Part 2 – Structured Programming and Object-based Programming

1



Program Execution in Java

- n As we have seen, for Java applications, execution of a program involves executing the statements in the main() method
 - n main() is the *entry point* of the program
- n The main() method may call other methods to execute the statements that they contain
 - n These methods in turn may call other methods
 - n When execution of the statements in a method are finished control returns to the point in the calling method immediately following the call

2



Structured Programming

- n Control flows from top to bottom
- n Each block has one entry point and (ideally) one exit point
- n Structured programming focuses on what the program is doing (control flow)
 - n Useful model is Input-Process-Output
 - n Data is dealt with separately
 - n In structured programs the data that a subroutine (in Java, a method) needs are passed to it as parameters
 - n In Java, if all the data that a method needs are passed as parameters it should be declared as **static**

3



A Simple Java Application

```
public class Application {  
    /* Returns the temperature in degrees Celsius of the  
       temperature supplied in degrees Fahrenheit */  
    public static double convertFtoC(double F) {  
        return (F-32) * 5 / 9;  
    } // end convertFtoC()  
  
    public static void main(String[] args) {  
        TextIO.put("Enter the temperature in °F: ");  
        double tempF = TextIO.getDouble();  
        TextIO.putln("The temperature in °C is " +  
                     convertFtoC(tempF));  
    } // end main()  
} // end Application class
```

4

Comments on Application

- n All the methods in Application are **static**
 - n Are **subroutines** in the terminology of the textbook
- n Application is never used as a type
 - n No variables of type Application are declared and no instances of the class are created
- n The data needed by convertFtoC() is passed in using the formal parameter, F
- n The **double** variable that is supplied as an argument to convertFtoC(), tempF, is declared in main()
- n convertFtoC() returns a **double** result
 - n in Application this is not stored anywhere (though it could have been) but is displayed on the console

5

Data in Structured Programming

- n "Records"
 - n Collections of data of different types, accessed by their name
 - n In Java, these can be simulated (as normally we would not have a Java class that just declares instance variables without any instance methods) using classes and their instance variables

```
public class StudentRecord {  
    public double test1, test2, test3;  
    public String name;  
}
```

6

Data in Structured Programming continued...

```
public class Application2 {  
    // Returns the average of the student's test scores  
    public static double average(StudentRecord s) {  
        return (s.test1 + s.test2 + s.test3)/3;  
    }  
    public static void main(String[] args) {  
        StudentRecord student = new StudentRecord();  
        student.name = "John Smith"; student.test1 = 54.0;  
        student.test2 = 41; student.test3 = 61;  
        double averageMark = average(student);  
        TextIO.putln(student.name + " average mark is " +  
            averageMark);  
    }  
}
```

7

Comments on Application2

- n The application's output is:
John Smith average mark is 52.0
- n StudentRecord "record type" allows a collection of data about a student to be stored as a single record
 - n A simplification of the Student class on page 192 of the textbook
- n The StudentRecord class declares all four of its fields as **public**, so that they can be accessed from other parts of the program
- n The subroutine, average(), is a function that is passed its data using the formal parameter, s (a StudentRecord)
- n The StudentRecord variable that is actually supplied as an argument to average(), *student*, is declared in the body of main() and contains a reference to a StudentRecord object that was created by main()

8

Data in Structured Programming

n Arrays

- n Collection of variables of the same type, accessed using an integer index
- n Accessing using an index makes code much more succinct than declaring separate variables for each array element
- n Element types can themselves be composite (arrays or records)
 - n In Java this means you also need to create each element of the array explicitly (as their default value is **null**)

9

Extending the example

- n Suppose we now add a subroutine that calculates the average mark for all the students enrolled in a module
- n Can use an array to hold the list of records for the students in the class and pass this as a parameter to the subroutine

10

```
public class Application3 {  
    public static double average(StudentRecord s) {  
        return (s.test1 + s.test2 + s.test3)/3;  
    }  
  
    public static double moduleAve(StudentRecord[] module) {  
        double total = 0.0;  
        int count=0;  
        while (count < module.length && module[count] != null) {  
            total += average(module[count++]);  
        }  
        if (count > 0)  
            return total / count;  
        else  
            return 0.0;  
    }  
}
```

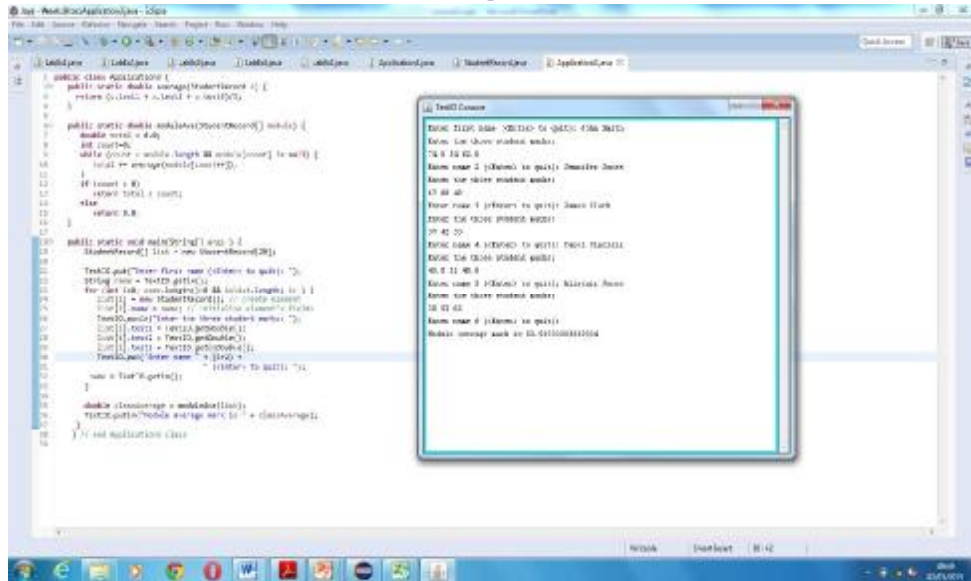
Static method names in
italics (as in Eclipse)

11

```
public static void main(String[] args) {  
    StudentRecord[] list = new StudentRecord[20];  
  
    TextIO.put("Enter first name (<Enter> to quit): ");  
    String name = TextIO.getln();  
    for (int i=0; name.length()>0 && i<list.length; i++) {  
        list[i] = new StudentRecord(); // create element  
        list[i].name = name; // initialise element's fields  
        TextIO.putln("Enter the three student marks: ");  
        list[i].test1 = TextIO.getDouble();  
        list[i].test2 = TextIO.getDouble();  
        list[i].test3 = TextIO.getlnDouble();  
        TextIO.put("Enter name " + (i+2) +  
            " (<Enter> to quit): ");  
        name = TextIO.getln();  
    }  
  
    double classAverage = moduleAve(list);  
    TextIO.putln("Module average mark is " + classAverage);  
}  
// end Application3 class
```

12

A run of the program



13

Notes on Application3

- n The subroutine `moduleAve()` is passed its data using the formal `StudentRecord[]` parameter, *module*
- n The `StudentRecord[]` array variable that is actually supplied as an argument to `moduleAve()`, *list*, is declared in the body of `main()` and contains a reference to a `StudentRecord[]` object. This array object *and* its element objects are created by `main()`
- n `moduleAve()` calls `average()` to compute the average mark for each student in the array
- n The `StudentRecord` variable that is actually supplied as an argument to `average()` is an element of the array referred to by the formal parameter, *module*, of the `moduleAve()` subroutine.
 - n So data passed to `moduleAve()` from `main()` is passed on to `average()` using its parameter.

14

The moduleAve() subroutine

- n As discussed in a previous lecture, the `moduleAve()` method assumes that any `null` elements in the array are located in consecutive positions at the end of the array
 - n If there were a “hole” in the array, any student records located after the hole would not be included in the average
 - n The condition in the for loop exits when first `null` element is encountered, or when the name is an empty String
- n If we had used an `ArrayList<StudentRecord>` instead of an array, it would have been reasonable to assume that the list contained no `null` elements, as these would only occur if `null` elements were explicitly added to the list

15

Structured Programming

- n Subroutines in structured programming take their data as parameters and can access the individual elements of the data supplied to them
 - n average() can see that a StudentRecord has *name*, *test1*, *test2* and *test3* as fields and what types these fields are) as can any code that calls the subroutine (main() and moduleAve() in this case)
- n Subroutines that act on data may be located in a part of the system far removed from where the data type was declared or where the data instances were created
 - n e.g. StudentRecord is not declared in Application3
- n Changing how the data is represented will break existing code elsewhere that uses it – fixing this may require subroutines in many parts of the system to be changed


16



Encapsulation

- n In object-based and object-oriented programming data is bundled up with the operations that act on it
 - n This is called *encapsulation*
- n Instead of subroutines that have knowledge of the internals of the data being passed to them as parameters, objects contain both the data and the methods that operate on the data
- n As both data and methods are part of the same object, the methods do not need to be passed the data as parameters by the caller and in general the caller has no access to the internals of the data
 - n so changing the representation of the data does not affect code that is far removed from where the data type is declared

17



Object-Based versus Object-Oriented Programming

- n Encapsulation – grouping data with the operations on that data into a single component – is what object-based and object-oriented programming have in common
- n Object-oriented programming adds two other ideas to encapsulation
 - n Inheritance
 - n Allowing one class to inherit methods and attributes from another class
 - n Polymorphism
 - n Allowing one object to substitute for another in responding to a method call, with different resulting behaviour depending on what type the substituting object is

18



What you need to know...

- n This module will not cover inheritance or polymorphism except that you should know about the Java class, Object
- n The class Object in Java defines some important instance methods, two of which are

```
public boolean equals(Object that);
public String toString();
```
- n All classes in Java are subclasses of Object, and every instance of every class has a `toString()` method and an `equals()` method
- n If a variable or parameter is declared as being of type Object you can assign a value of any type to it

19



Object parameters

- n This is the definition of the `put()` method in `TextIO`:

```
public static void put(Object x)
```
- n The above method is the one that gets invoked for each of these calls:

```
TextIO.put("Goodbye");           // String object
TextIO.put(6);                   // Integer object
TextIO.put(Math.random()*0.5);   // Double object
TextIO.put(student);             // Student object
```

20



The toString() method

- n What `TextIO.put()` (and `System.out.print()`) print when displaying an object on the console is the String returned by the object's `toString()` method (see Lab 11), which they call
- n For the last call on the previous slide (the one with a Student parameter) the output would be something like:

`Student@7291c18f`

- n The Student class in lecture 13 did not define its own `toString()` method so the one that gets called is the one it inherits from the type, `Object`. The String returned by that method contains the name of the actual object type (Student, in this case) followed by an @ sign and a hexadecimal number that is typically different for each object

21



Comments on Object

- n Because every class inherits `Object`'s methods, it is guaranteed that every instance of every class will have a `toString()` method and an `equals()` method (and others we haven't discussed) that can be called
- n `TextIO.put()` is only able to display any object on the console because it can rely on that object having a `toString()` method that returns a String representation of its value, and `put()` "knows" how to display a String


22



"Overriding" a method

- n Object-oriented languages allow a class to change the behaviour of a method that the class inherits by redefining that method in the class, so that it is not the inherited method but the new method that is called when the method is invoked on one of the class's instances
- n This is called "overriding" the method
- n The new method declaration must have the same name, parameters and return type as the inherited method

23



The Student class from last week's lecture

```
public class Student {  
    private String name;  
    private double test1, test2, test3;  
    Student(String theName) { ... }  
    public String getName() { ... }  
    public double getAverage() { ... }  
    public void setTest1(double score) { ... }  
    public double getTest1() { ... }  
    public void setTest2(double score) { ... }  
    public double getTest2() { ... }  
    public void setTest3(double score) { ... }  
    public double getTest3() { ... }  
}
```

24



Now add toString()...

```
public String toString() {  
    return "Name: " + name + "; Test scores: "  
        + test1 + ", " + test2 + ", "  
        + test3 + "; " +  
        "\nAverage grade = " + getAverage();  
}
```

- n The purpose of the method is to give a String representation of the state of the instance of the class, in this case the name of the student and their three test scores.
- n Now calling `TextIO.put()` with a `Student` argument will produce something like:

```
Name: John Smith; Test scores: 30.0, 70.0, 50.0;  
Average grade = 50.0
```

25



Terminology from today

n Encapsulation

- n The grouping together of data and methods in a single component, hiding representation details and providing a public interface to the component

n Inheritance

- n An object-oriented concept whereby one type can inherit properties and behaviour from another type. In Java, all classes inherit methods from the class, `Object`, and values of any type can be assigned to a variable of type `Object`

26



Reading

- n A good extended example of object-based programming can be found in section 5.4 of the book
- n Study this

27