CS1002: COMPUTER SCIENCE
OO MODELLING & DESIGN: WEEK 5

School of Computer Science
University of St Andrews

Graham Kirby
Alan Dearle

This Week

- More on Java classes
- Constructors
- Modifiers

Fields and Variables

- A field:
  - has a type and identifier
  - holds information relevant to object of specific class
  - is defined in class
  - exists as long as the object does

- A variable:
  - has a type and identifier
  - holds temporary information relevant to method execution
  - is defined in method
  - exists only while method is being executed

W05 Example 1: Variables & Fields

```java
public class C {
    private int a = 2;
    public void m() {
        int b = 7;
        b = 16;
        a = -9;
    }
}

public class D {
    public static void main(String[] args) {
        C obj = new C();
        obj.m();
    }
}
```
**Memory States**

```java
public class C {
    private int a = 2;
    public void m() {
        int b = 7;
        b = 16;
        a = -9;
    }
}
public class D {
    public static void main(String[] args) {
        C obj = new C();
        obj.m();
    }
}
```

**W05 Example 2: Variable Scope**

```java
public class C {
    public void m1() {
        int i = 32;
        System.out.println("i: "+i);
    }
    public void m2() {
        i = i - 3;
    }
}
public class D {
    public static void main(String[] args) {
        C obj = new C();
        obj.m1();
    }
}
```

**W05 Example 3: Method Calls**

```java
public class C {
    public void m1() {
        C another_one = new C();
        another_one.m2();
        this.m2();
        m2();
    }
    public void m2() {
        ...
    }
}
public class D {
    public static void main(String[] args) {
        C obj = new C();
        obj.m1();
    }
}
```

**Memory States**

```java
public class C {
    public void m1() {
        C another_one = new C();
        another_one.m2();
        this.m2();
        m2();
    }
    public void m2() {
        ...
    }
}
public class D {
    public static void main(String[] args) {
        C obj = new C();
        obj.m1();
    }
}
```
Method Parameters

- Methods take parameters:
  - information passed to the method when it's called, to tell it specifically what to do
  - each parameter is treated like a variable within the method body
    - equivalent to a variable containing the value specified by the caller of the method

Parameter and Field Names

- Sometimes convenient to use fields and parameters with the same names:

```
public class C {
    int salary = 0;
    public void setSalary(int salary) {
        salary = salary;
    }
}
```

```
public class D {
    public static void main(String[] args) {
        C obj = new C();
        obj.setSalary(400);
        System.out.println("obj.salary = " + obj.salary);
    }
}
```

Distinguishing Parameters and Fields

```
public class C {
    int salary = 0;
    public void setSalary(int salary) {
        this.salary = salary;
    }
}
```

```
public class D {
    public static void main(String[] args) {
        C obj = new C();
        obj.setSalary(400);
        System.out.println("obj.salary = " + obj.salary);
    }
}
```
Constructors

- We may wish to ensure that new objects of a particular class are always initialised in a certain way

```java
Student s = new Student();
```

### Constructors cont'd

- A constructor is like a special method that is executed when an object is created
  - can initialise the object in any required way
  - can take parameters, like method
  - doesn't return any result
  - always has the same name as the class

---

**W05 Example 7: Single Constructor**

```java
public Student(String name, String id) {
    this.name = name;
    this.id = id;
    year = 2004;
}
```

- Can now create student object with

```java
Student s = new Student("Jane Jones", "040412716");
```

---

**Test Class**

- Can put code in Test class to test other class implementation e.g.
  - instantiate student object with `name` and `id`
  - print out the details of the student object

```java
public class Test {
    public static void main(String[] args) {
        Student s = new Student("Jane", "040412716");
        s.printDetails();
    }
}
```
Multiple Constructors

- May want several different ways to initialise objects
  - can have more than one constructor in a class
  - but they must all take different parameters – why?

- Use
  
  ```java
  Student s1 = new Student("Jane Jones", 040412716);
  Student s2 = new Student();
  Student s3 = new Student("Jane Jones");
  ```

- Example of overloading
  - the identifier of the constructor refers to different bits of code depending on the number and type of its arguments

Default Constructor

- Simple constructor defined automatically if we don't write any
  - just creates instance and fills in default attribute values
  - equivalent to, for example:

  ```java
  public Student() {
      this.name = null;
      this.id = null;
      this.year = 0;
  }
  ```

Linking Constructors

- So we have seen
  - a class can have multiple constructors
  - distinguished by different parameters

- May want to call one constructor from another, to avoid duplicating code
  - use this keyword
**Method Overloading**

- We have seen multiple constructors with the same names but different parameters
  - can also do the same with other methods
  - only do this when the methods are closely related in purpose...

**Encapsulation**

- An object’s field values are (should be) encapsulated
  - they can’t be seen directly by other objects
  - but can be accessed by that object’s methods

- Benefits
  - improves control over how field values are read and updated
  - helpful in structuring software
    - can change one part without affecting others

**Encapsulated Objects**
Getters and Setters

- So, may need to write methods for each class that allow other objects to access its fields.

- Accessor methods:
  - set the content of a field to a certain value (setter)
  - return the value held in a field (getter)

- Provides standardised access to fields by
  - other parts of your program (other classes)
  - other programmers when using your code

Visibility and Other Modifiers

- Some modifiers:
  - public, private, protected, static, final

- Apply to classes and other members:
  - fields, methods, constructors

public Modifier

- Modifier for classes, fields, methods, constructors
  - means that class or member can be accessed from anywhere
  - no restrictions:
    - class can be instantiated
    - field can be read or updated
    - method can be called
    - constructor can be called

```java
public class Person {
    public String name;
    public int getAge() {...}
    public Person(String s) {...}
}
```

private Modifier

- Modifier for fields, methods, constructors
  - means that member can't be accessed from anywhere outside the class
  - restrictions:
    - field can't be read or updated
    - method can't be called
    - constructor can't be called

```java
public class Person {
    private String name;
    private int getAge(); ...
    private Person(String s); ...
}

* the normal modifier for fields
**protected Modifier**

- Modifier for fields, methods, constructors
  - means that member can be accessed only from the defining class or another class in the same package

```java
public class Person {
    protected String name;
    protected int getAge(); ...
    protected Person(String s); ...
}
```
- the default if you don't write any visibility modifier

**Static Fields**

- Fields can belong to an instance of the class (object) or to all instances (objects) of the class
- For non-static fields there is a memory slot inside every object of a certain class
- For static fields there is a single memory slot which pertains to the whole class
- Need to decide which is sensible...

**static Modifier**

- Modifier for fields and methods
  - means that member is associated with class as a whole rather than individual object

```java
public class Person {
    private String name;
    private static int number_of_people;
}
```

**Static Field Example**

- Code example

![Static Field Image](www.tyhen.com/images/StaticField.jpg)
Static Fields cont’d

- Static fields can be referred to from outside a class (subject to visibility) using the class name, e.g.
  
  `Person.number_of_people`

- Within the class, they can be referred to just like non-static fields i.e. without the class name prefix

Static Methods

- Methods can also be declared static
  - already seen this with main method:
    ```java
    public static void main(String[] args){...}
    ```

- Only makes sense when method doesn’t have anything to do with any particular instance of the class, e.g.
  ```java
  public static int getTotalPeople(){
    return number_of_people;
  }
  ```

Static Methods cont’d

- Static methods can also be referred to from outside a class using the class name, e.g.
  ```java
  int count = Person.getTotalPeople();
  double x = Math.sin(0.6);
  String string_rep = String.valueOf(4.251);
  ```

Rule

- A static thing can’t access a non-static thing

  - code in a static method
  - code initialising a static field
  - a non-static field
  - a non-static method
**final Modifier**

- The static keyword is perhaps confusing
  - suggests that the value of the field cannot change

- Use final keyword to declare a field to be constant
  - once a value has been assigned to it, it cannot be changed

  ```java
  public static final int MAX_ARRAY_SIZE = 100;
  private static final String PROMPT = "enter int";
  final int x = y + z;
  ```

- Modifier applies to methods too
  - we'll see later what this means (to do with inheritance)