

Software Development (CS2500)

Lecture 54: Summary

M.R.C. van Dongen

March 7, 2011

1 Basic Java

Java is a high-level object oriented programming language. You write your programs in a Java source file. The `javac` compiler turns them into object files. The object files consist of byte code. Executing a program is done by the Java virtual machine. It interprets the object files at run time. The program starts executing with 'the' method `main` of the main class.

2 Classes and Objects

OO programming lets you extend working programs without touching working code. All Java code is developed in a class. Classes are used to define/create objects. An object is called an *instance* of its class. The class is a blueprint of its instances. An object knows and does things. What it knows is determined by its instance variables. This is called the object's state. What it does are its methods. This is called the object's behaviour.

3 Variables

Primitive: Their values are bits representing primitive type values.

Reference: A reference value references an object. The value `null` doesn't reference any object. Any other reference value is like a remote control. Using the dot operator you control the remote: Lets you access the value of an instance variable. Lets you call an instance method. Using dot operator on `null` causes runtime error. Arrays are object too.

4 State and Behaviour

Classes define what an objects may know and do.

State: Each object knows (the values of) its instance variables.

Behaviour: A method does things by calling its instance methods.

Instance methods may use instance variables. This allows different objects to have different behaviour. Method calls are carried out using the pass-by-value paradigm.

Methods have two kinds of parameters:

Formal: Name in the parameter list of a method definition.

Actual: Expression in the parameter list of a method call.

To evaluate a method call with n parameters:

1. For i from 1 to n (from left to right):
 - (a) Evaluate the i th actual parameter.
 - (b) Create fresh variable to represent i th actual parameter.
 - (c) Assign result of evaluation to the fresh variable.
2. Carry out statements in the method body.
3. Return result (if any) and free up temporary variable space.

5 The Java API

`ArrayList` is a class in the Java API. It is an example of a generic class: Class is parameterised over the type of objects it contains.

Declaring: `ArrayList<Integer> list;`

Creating: `list = new ArrayList<Integer>();`

Adding: `list.add(1);`. Uses autoboxing.

Removing: `list.remove(thing);`

...

6 Writing Classes

Classes contain variable and method definitions. Class variables/methods are `static`. They are owned by the class they're in. Instance variable/methods are `non-static`. They're owned by instances of the class they're in. Visibility modifiers restrict access to variables/method.

`public class:` Anywhere.

`public instance:` If you have an object reference.

private class: Inside the class.

private instance: Inside the class: also needs object reference.

Encapsulation helps protect class and instance variables. Defining class and instance methods/attributes:

Instance attribute: If value determines object state.

Class attribute: If value determines class state.

Instance method: If it uses attributes of `this`.

Class method: If it doesn't use attributes of `this`.

7 Inheritance and Polymorphism

A subclass extends its superclass. Subclass inherits all *public* instance variables/attributes. Inherited methods define common default behaviour. They allow for automatic code reuse. Inherited methods can be overridden. This allows you to define class-specific behaviour. When an instance method is called, the lowest overridden definition is used. Overloading a method: The method has the same name, and A different list of parameter types. The *IS-A* test determines when classes extend other classes: Dog IS-A Canine. Cat IS-A Feline. But not: Animal IS-A Dog. And not: Conservatory IS-A House. Class extension works only in one direction. Class extension is transitive.

8 Polymorphism

Polymorphism lets us use a subclass instance as if it is a superclass instance. *Abstract classes* are classes which should not be instantiated. You mark them with the keyword `abstract`. They have abstract and concrete methods. All abstract methods must be implemented in some concrete subclass. The `Object` class is Java's ultimate superclass. An *Interface* is 100% abstract class: Interface methods are implicitly public and abstract. Classes can only extend one superclass. They can implement more interfaces. Superclass constructor may be called in own constructor: `super()` (but only as first statement). Overridden superclass methods may also be called: `super.method()`.

9 Exceptions

Risky methods may throw exceptions. An exception is a subclass of the `Exception` class. An exception must be declared if it's not a `RuntimeException`. Use `throws Exception` after method argument list. You can "return" an exception by throwing it: `throw new Exception();` Exceptions must eventually be caught. Ignoring exceptions usually means more complicated handling. You catch exceptions with `catch` block.

```

try {
    ...
} catch (Exception e) {
    // handle it.
} finally {
    // code that must always run.
}

```

Java

10 Special Classes

An nested class is a class within an other class. There are two kinds of nested classes.

Inner class: Access to all instance attributes/methods.

Static class: No access to all instance attributes/methods.

An anonymous class is a class without a name. Use them if the class is needed only once. An anonymous class has access to all instance attributes. You create them like this: `new Class() { <body body> }.`

11 Events

Events let us deal with event-driven behaviour. To deal with events you need event handlers (listeners). Different events may require different kinds of listeners. Detecting a button clicks requires listening to an `ActionEvent`. Implement the `ActionListener` interface. Override the `actionPerformed()` method. Register listener: `button.addActionListener(listener)`. Usually, the listener is `this`. When the user clicks the button `listener.actionPerformed()` is called. This lets listener handle the event.

12 Threads

A thread is a separate thread of execution. Each thread has its own call stack. It is represented by a `Thread` object. You create a `Thread` with a `Runnable` instance:

1. `Runnable job = new Runnable();`
2. `Thread thread = new Thread(job);`
3. `thread.start();`

`Runnable` is an interface. All you need to do is overriding `void run()`.

Threads share their resources of the process they're in. If threads don't cooperate *race conditions* may occur: The output/result of the program is ill-defined. Program depends on right sequence/timing of other events. Race conditions may be avoided by synchronising methods. Simply add `synchronized` before return type. Synchronising the method allows at most one process inside the method. Unfortunately, synchronisation may cause deadlock: Two or more threads are waiting for each other to return from the method they're in.

13 Enumerated Classes

Enumerated classes overcome enum int problems.

```
public enum Colour { RED, GREEN, BLUE; }
```

Java

Enumerated constants may have constant-specific attributes.

```
public enum Example {  
    CONSTANT_A( 1 ), CONSTANT_B( 2 );  
    private int value;  
    public Example( int value ) { this.value = value; }  
}
```

Java

They may also have constant-specific methods:

```
CONSTANT_A( 1 ) { public void method( ) { ... } },  
CONSTANT_B( 2 ) { public void method( ) { ... } };  
public abstract void method( );  
...
```

Java

14 Coding Conventions

Pay attention to them!