M257: Putting Java to Work

Introduction
Unit 1: Java Everywhere
Introduction
   – Welcome!
   – Before you start
   – Course Assessments
   – The course at a glance
   – How to pass M257

Unit 1: Java Everywhere
1. Java background
2. Getting Java running
3. Objects and state changes
4. Classes in Java
5. Inheritance
Introduction

The Tutor...

Name

Email:
Welcome!

• In this course, you will learn:
  – how Java originated and how it is used nowadays.
    • Java is the language most identified with the internet and its friendly face, the web.
    • Java was introduced by Sun Microsystems.
  – the language used to build systems that power so much of the web in applications ranging from e-commerce to online games, downloadable music to online banking.
Before you start

- **Course materials:**
  - **12 units of study text** (units 11 and 12 are optional)
  - The Course Guide (**important to read!**).
  - M257 Java Quick Reference.
  - The NetBeans Guide.
  - The Course Software
    - 2 CD-ROMs: the NetBeans CD-ROM and the M257 Software
  - The Glossary.
  - The Index.
  - The M257 course website.
  - The Study Calendar (available on the course website)
Before you start

• **Prerequisite knowledge:** You should be familiar with at least some of the following concepts:
  • objects;
  • objects interacting by invoking methods (exchanging messages);
  • classes;
  • inheritance;
  • simple object-oriented design;
  • control structures such as if statements.

*You could have obtained this knowledge from studying AOU courses such as M255 or other equivalent courses.*

• **Computing facilities:** You need to have:
  • Access to a computer
  • Obtained access to the internet
  • Word-processing software for completing your TMAs
  • Installed the supplied software as explained in the NetBeans Guide.
Before you start

• Activities for you to run!
  – At various points in the units, there will be practical activities for you to carry out.
  – These will be indicated by a keyboard icon
  – The activities for Units 1–10 are available on the M257 Software CD-ROM.

Activity 2.1
Experimenting with data types and operators. (*Example*)
The course at a glance

- The course has 12 units:
  - Unit 1: Java everywhere
  - Unit 2: Java in the small
  - Unit 3: Java in the large
  - Unit 4: Input, output and exceptions
  - Unit 5: Packages and abstraction
  - Unit 6: Graphical user interfaces
  - Unit 7: Event-driven programming
  - Unit 8: Threads
  - Unit 9: Internet programming
  - Unit 10: let’s do Java
  - Unit 11: Web-based case studies (Optional)
  - Unit 12: Web-based case study(Optional)
Course Assessments

• T257:
  – 10 obligatory units + 2 optional units.
  – 5 credit hours, one semester.

• Course grade distribution:
  – 1 TMA: 20%
  – 1 MTA: 30%
  – 1 Final: 50%

• Like any other course at AOU, to pass M257 you should:
  1- Get at least 40/100 as average grade of TMAs and Quizzes, AND
  2- Get at least 40/100 as average grade of final one and final two, AND
  3- Get at least 50/100 as average of both
Course Assessments

• Power point presentation are **NOT** enough to study the course: you must study the material’s units

• TMA **late submission** will lead to grade deduction penalties

• **Attendance:**
  – *3 unjustified absences will lead to course drop*
  – Attending other than your tutorials session is not allowed unless you hold a permission from the coordinator.
Course Assessments

PLAGIARISM

Case 1: Two TMAs are identical.
  - Both TMAs will get a total grade of ZERO.

Case 2: A part of a TMA is copied from another TMA:
  - Both TMAs will get a zero for this part.
  - An additional penalty will be applied.
M257 is an intensive course that requires significant effort from students. Here are some guidelines to help you pass the course:

- **Study**
  - Always study what has been given in a lecture asap.

- **Revise**
  - Always revise what you have studied in previous units before coming to the lecture.

- **Prepare**
  - Before coming to the lecture, skim through the contents of the unit(s) to be given in that lecture. Use the study calendar to help you in that matter.

- **Practice.. Practice.. Practice!!**
  - Don’t just read or memorize, practice what you have learned and watch your skills improve. Practicing also helps you understand and memorize! You may use Java NetBeans for that purpose.
Introduction
– Welcome!
– Before you start
– Course Assessments
– The course at a glance

Unit 1: Java Everywhere
1. Java background
2. Getting Java running
3. Objects and state changes
4. Classes in Java
5. Inheritance
1. Java background

• Java runs on servers, PCs, mobile phones and PDAs (personal digital assistants).

• It can be found in
  – TV set-top boxes,
  – in embedded devices (forming part of larger systems such as cars, robots or printers),
  – in smart cards, and
  – wearable computers.

Java really is everywhere.
1. Java background

The aims of the Java language

- **Simple** to learn, & closely based on C++, but reduced in complexity.
- **Object-oriented**
- **Robust**: Java programs are strictly checked by software before they run
- **Secure**: Java ensures that programs running **over networks** cannot damage your computer files or introduce viruses.
- **Portable**: Java programs can easily be transferred from one platform (e.g. Windows) to run on another platform (e.g. Linux) with little or no change.
- **High performance** (fast)
- **Interpreted**: a key aspect of Java portability (more in Section 3).
- **Threaded**: allow a program to do several things at once.
- **Dynamic**: Java programs can adapt to **changes in their environment** even while the program is running.
1. Java background

Versions of Java

• The first publicly available version was Java 1.0.
• Now, Java 2 is the standard language
  – and there have been a number of further versions (Java 2 version 1.3, Java 2 version 1.4, Java 2 version 1.5 and so on), each introducing relatively minor improvements

Editions of Java

• Why Java editions was introduced?
  – to cater for the different needs of, say, large international business systems to software running on mobile phones with very limited hardware resources.
• Current editions:
  – Java 2 Standard Edition (J2SE) → focus of our study!
  – Java 2 Enterprise Edition (J2EE) → for large-scale systems
  – Java 2 Micro Edition (J2ME) → for mobile phones (unit 10)
1. Java background

A simple Java program

```java
public class HelloWalrus {
    public static void main (String[] args) {
        System.out.println("The time has come");
        System.out.println("the walrus said");
    }
}
```

The output:

The time has come
the walrus said
The first line of the program introduces a class, called HelloWalrus.

- When the program runs, the class gives rise to an object, which executes the code of the class.
- This class is defined as public, which means that objects of this class can be freely accessed by other objects in a larger system.
- The curly bracket '{' on the line following HelloWalrus matches the closing bracket '}' at the end of the example, as these enclose the whole class definition.

The code in this class is in the form of a method, in this case a method called main.

- The first line of the method is called the method header.
- We defer explaining the words static and void until later, but they are always used with main methods like this one.
- The method header defines the name of the method (in this case, main) and is followed by the method body, which is enclosed in curly brackets.

The System.out.println method displays, in a screen window, the exact text between the quotes.

- Each of these two lines of code constitutes a statement, a single command to be carried out.

Each statement is terminated by a semicolon ';'.

Unit1: Java Everywhere

1. Java background
Using comments in Java programs

• A comment is ignored by the computer system
  — **Form 1: Block Comment**. Text enclosed in the symbols /* and */

    /* A simple Java program
    Author: The Course Team
    Date of creation: 01/01/01 */

    public class HelloWalrus
    ...

  — **Form 2: Line Comment or In-line Comment** (using //)
    • Line comment:  // display a poetic message
    • In-line comment: } // end of class HelloWalrus
Introduction

- Welcome!
- Before you start
- Course Assessments
- The course at a glance

**Unit1: Java Everywhere**

1. Java background
2. Getting Java running
3. Objects and state changes
4. Classes in Java
5. Inheritance
2. Getting Java running

Here we discuss the portability issue: the way that programs written in Java can be run on many different platforms.

The conventional way
- source code is translated by a compiler to native code (the base language of the computer) which is then executed.
- The native code that has been generated can be executed only by the particular type of computer that recognizes it.

In the case of Java
- source code is translated by a compiler to bytecode (intermediate form) which is translated by an interpreter into the native code of the computer it runs on which is then executed.
- The interpreter is a program that translates bytecode into the native code of the computer it runs on.
- Java interpreters are available for many different platforms.
2. Getting Java running

The Java Software Development Kit (abbreviated as SDK or JDK)

- SDK tools are used for compiling and running Java programs on a variety of platforms
- SDK tools are freely available from Sun Microsystems
- SDK tools include:
  - The Java compiler (javac) translates Java source into bytecode.
  - The Java interpreter (java) translates and executes Java bytecode. Java interpreters are available for many different computer systems.
  - The document generator (javadoc) processes Java source files to produce useful documentation, such as standardized descriptions of components of Java code.
  - The Java debugger (jdb) helps to look for errors in programs.
  - The Java disassembler (javap) reads bytecode files created using the Java compiler and displays the corresponding source code.
  - The applet viewer (appletviewer) allows you to run and debug Java applets without a web browser (Java applets are explained in the next section)
2. Getting Java running

• Java programmers can produce two types of software:
  1. **Applications** are stand-alone programs, which can run independently.
  2. **Applets** are programs that can be included in web documents and are run using a browser.

• Two ways to develop and run Java software:
  – To use **Java SDK** along with a **text editor** (mostly run from command line)
  – To use an **integrated development environment (IDE)**,
    • such as the tool supplied with this course.
    • IDE provides a sophisticated GUI and editor, with integrated facilities for compiling, running and debugging programs.
    • There are many IDEs available for Java, and they typically build on top of the free software from Sun.
Outline

Introduction
  – Welcome!
  – Before you start
  – Course Assessments
  – The course at a glance

Unit 1: Java Everywhere
  1. Java background
  2. Getting Java running
  3. Objects and state changes
  4. Classes in Java
  5. Inheritance
3. Objects and state changes

• Assume we have robots moving around a 2D grid, displayed on a computer screen.
  – The position of a robot on the grid is expressed as x- and y-coordinates,

• Each robot is represented by an object in the program, and it has a name (upper figure)

• Each robot object stores its current position, i.e. stores its state (lower figure)

• Objects typically have a number of operations that they can carry out on request
  – The Java code that carries out an operation is known as a method.
  – Requesting the operation is known as invoking the method
  – if the method is invoked correctly, the object is then said to execute the method.
3. Objects and state changes

Examples of invoking methods:

```java
artoo.moveNorth();
deetoo.moveEast();
deetoo.moveSouth();
meetoo.moveTo(5,3);
```
3. Objects and state changes

• Methods can be:
  
  – **Mutator Methods**: have an effect on the state of the object
    • For example, artoo.moveTo(4,3);
    • Note: The two numbers within the brackets are known as arguments

  – **Accessor Methods**: methods correspond to a simple request for information and cannot alter the state of the object.
    • For example, artoo.getX(); ➔ if we want a robot object to indicate its x position.
3. Objects and state changes

• An object has **state** → **a state is the data that is contained in that object**

• In our example, states were represented by **integers**.
  — Integers, characters and floating-point numbers are **primitive data types**.
  — Since integers **occupy a space in the memory** of the computer and need to be **identified**, Java provides a **declaration** facility; an example of declaring two **variables**:
    ```
    int x;
    int y;
    ```
    or
    ```
    int x, y;
    ```

**Assignment statement:**

— To give values to the variables, we use **assignment** statement:

```java
variable = expression;
```

For example, `x = 23; y = 44;`
3. Objects and state changes

Assignment statement (Cont’d):

• Assignments can be combined with declarations. For example, the statement:

```java
int newPos = 99;
```

is equivalent to the code:

```java
int newPos;
newPos = 99;
```

• Another example; we declare and initialize some variables:

```java
int oldPos = 33;
int newPos;
int increment = 3;
```

• Here, we use the variables in mathematical expressions:

```java
newPos = 12 + oldPos;
oldPos = (2 * increment) – 12;
oldPos = (x + y) * (x + z);
```

The first statement places 45 (33 + 12) into the location labelled `newPos`. The contents of `oldPos` are unchanged. The second statement multiplie
Assignment statement (Cont’d):

• Another important example:

```java
y = y + 1;
```

• Assume that the variable `y` contains the value 333. Then when this statement is executed, the action that is carried out is to extract the current value of the variable (333), add one to it and then place this new value (334) into the variable.

![Diagram](image-url)
Outline

Introduction
– Welcome!
– Before you start
– Course Assessments
– The course at a glance

Unit 1: Java Everywhere
1. Java background
2. Getting Java running
3. Objects and state changes
4. Classes in Java
5. Inheritance
4. Classes in Java

- **Objects** in Java are defined by means of **classes**.
- **A class** is much like a **template** that defines the **structure** of:
  1. the stored data associated with an object and
  2. the program code that is executed when particular methods are invoked.

- **For example:**
  - All the robot objects in Section 5 could be defined as instances of a class called Robot.
  - Each object of the class has the same methods and the same structure for its stored data.
  - Only the values of the data (the state of the object) may differ between objects.
Defining a Java class

• The general structure of a Java class is as follows:

```
class ClassName
{
    Data declarations
    Method definitions
}
```

– The Java keyword `class` is used with an appropriate `identifier` as the name of the class. Class names conventionally start with an `upper-case letter`.
– The `body of the class` is delimited by curly brackets `{}` and contains both `data declarations` and details of the `methods` for the class.
Defining a Java class (Cont’d)

Example:

```java
public class Robot {
    private int x; // x-position (W-E) of robot
    private int y; // y-position (N-S) of robot
    ...
    public void moveNorth () {
        y = y + 1;
    }
}
```

The instance variables:

- The data for objects of the **Robot class** consists of two **instance variables**, `x` and `y`, which define where the robot is on the grid; these variables will hold **integer** values. These variables are **private**, which means that that this data cannot be directly accessed by objects of other classes.

The methods:

- The second part of the Robot class is made up of definitions of the **methods** that can be invoked on Robot objects (e.g. the `moveNorth` method). Methods are normally declared **public**, which means that they can be invoked by objects of other classes. The keyword **void** indicates that the method does not return a value.
4. Classes in Java

Naming guidelines for Java programs

– The language is case-sensitive, so that it does matter whether you write a variable name in upper case, lower case or a mixture
– Java keywords such as class or public must always be in lower case.
– There are standard Java conventions for identifiers, which are aimed at making programs consistent and readable:
  • Class names are in mixed case, always starting with an upper-case letter. Class names will normally consist of one or more nouns.
  • Variable names are in mixed case, always starting with a lower-case letter.
  • Method names are in mixed case, always starting with a lower-case letter. Normally a method name should start with a verb.
  • We usually use further upper-case letters for the start of any new word within a class, variable, or method name.
Creating and using objects

• To create an object named `artoo` of the type `Robot`,

```java
Robot artoo;  // This declares the variable artoo, which is used to access Robot object
artoo = new Robot();  // This allocates memory space for the data stored by artoo.
```

or simply

```java
Robot artoo = new Robot();
```

— Following these statements, we can now invoke methods on the newly created objects, using the syntax we have seen earlier. For example:

```java
artoo.moveNorth();
```
4. Classes in Java

Methods that return a value

- An example of a method called `getX` which return a value, the current x position of the Robot object.

```java
public int getX()
{
    return x;
}
```

- The method header includes the keyword `int` (rather than `void` as before) to indicate that the method will return an integer value.
- In the body of the method, the `return` keyword precedes the value to be returned.

- **Invoking the `getX()` method:**

```java
int currentX;
currentIndex = deetoo.getX();
```

- Note that if we simple write `deetoo.getX();` without assigning it to a variable then this statement would be useless and **will be discarded.**
4. Classes in Java

Methods that have arguments

Example:

```java
class Bar
{
    public void moveTo (int nextXPos, int nextYPos)
    {
        x = nextXPos;
        y = nextYPos;
    }
}
```

- The method header shows that this method returns no value and has two integer arguments, `nextXPos` and `nextYPos`. The two arguments specify the desired next position of the robot.
- The arguments `nextXPos` and `nextYPos` are known as formal arguments.

- **Invoking the `moveTo()` method:**

  ```java
  myRobot.moveTo(5, 1);
  ```

  - The values used as arguments when the method is invoked (such as 5 and 1 in the above example) are known as actual arguments.

- The name of a method together with the number and types of its arguments is known as the method signature \( \text{moveTo}(\text{int}, \text{int}) \) in the above example.
4. Classes in Java

Constructors

- Constructors are used to **give an object an initial state** when it is created. For example,

```java
public Robot ()
{
    x = 1;
    y = 1;
}
```

- So, when using the statement `Robot meetoo = new Robot();` the initial values for `x` and `y` is 1 and 1.
4. Classes in Java

Constructors (Cont’d)

• Like methods, constructors can have arguments, for example:

```java
public Robot (int xPos, int yPos)
{
    x = xPos;
    y = yPos;
}
```

  – We could invoke the constructor as follows:

```java
Robot = new Robot (6, 5);
```

• We could invoke the constructor as follows:
The Robot class:

class Robot
{
    // instance variables
    private int x;
    private int y;

    // constructors
    public Robot () {
        x = 1;
        y = 1;
    }
    public Robot (int xPos, int yPos) {
        x = xPos;
        y = yPos;
    }

    // methods
    public void moveNorth () {
        y = y + 1;
    }
    public void moveSouth () {
        y = y - 1;
    }
}

public void moveEast () {
    x = x + 1;
}
public void moveWest () {
    x = x - 1;
}
public void setX (int xPos) {
    x = xPos;
}
public void setY (int yPos) {
    y = yPos;
}
public int getX () {
    return x;
}
public int getY () {
    return y;
}
public void moveTo (int xPos, int yPos) {
    x = xPos;
    y = yPos;
}
} // end of class
The role of the main method

The Role of The main method: it is a starting point for creating objects and invoking their methods

Example:

- Test the Robot class:
  - First, we need to define a class that can be used to carry out the testing; let us call it TestRobot.

- It would actually be possible to put this main method into the class Robot, rather than creating a new TestRobot class.

- However, this is not normally good practice. It is better for the Robot class to be more general purpose, so it could potentially be used in lots of different programs.
Introduction
- Welcome!
- Before you start
- Course Assessments
- The course at a glance

Unit 1: Java Everywhere
1. Java background
2. Getting Java running
3. Objects and state changes
4. Classes in Java
5. Inheritance
5. Inheritance

- If we need various objects that are similar in structure, but not exactly the same → use inheritance.

- When a class B inherits from a class A, then objects of class B are in some sense extended versions of class A objects.
  - Class B objects have the same methods and instance variables as class A objects; they can also have additional methods and instance variables, or can modify the methods they inherit.
  - B is called subclass of A
  - A is called superclass of B.

- The key word extends is used to express inheritance

```java
public class B extends A {
    // ...
}
```

Figure 12: UML diagram of class B inheriting from class A
5. Inheritance

- In Java,
  - a class can inherit directly from only ONE other class
  - a class can have more than one subclass
  - It is also possible to have more levels of inheritance
  - This following diagram is known as a class hierarchy diagram. Class hierarchies are a very powerful way of describing the relationship between classes.

![Class Hierarchy Diagram](image-url)
Inheritance example

• Notice that we do not have to repeat any of the methods or the instance variables of the Robot class. The extends keyword ensures that these are available to MagicRobot objects automatically.

• We can also add a constructor as follows:

```java
public MagicRobot() {
    makeInvisible();
}
```
5. Inheritance

**Overriding methods**

- Overriding means that subclasses modifies the methods they inherit from a superclass.
- This modification is limited in that the body of the method can be modified, but the method signature must stay the same.
5. Inheritance

Overriding methods (Cont’d)

Example:

- Suppose we want to define another specialized kind of robot, this time a variable-speed robot, modeled by the class SpeedRobot. This robot can move across the grid at different speeds depending on an internally stored step size, rather than always taking one step for basic moves.

- Assume the SpeedRobot class is as shown on right.

- We can invoke the standard move methods, moveNorth, moveWest and so on, as the SpeedRobot class inherits them from the Robot class. However, these do not take account of the new speed data, so we need to rewrite them appropriately.
Overriding methods (Cont’d)

Example:
- Suppose we want to define another specialized kind of robot, this time a variable-speed robot, modeled by the class SpeedRobot. This robot can move across the grid at different speeds depending on an internally stored step size, rather than always taking one step for basic moves.
- Assume the SpeedRobot class is as shown on right.
- We can invoke the standard move methods, moveNorth, moveWest and so on, as the SpeedRobot class inherits them from the Robot class. However, these do not take account of the new speed data, so we need to rewrite them appropriately.

```java
public class SpeedRobot extends Robot {  // incomplete
    private int speed;  // step size for moves

    // constructor
    public SpeedRobot (int initialSpeed)
    { speed = initialSpeed; }

    public void setSpeed (int newSpeed)
    { speed = newSpeed; }

    public int getSpeed ()
    { return speed; }
    ...
}
```
Overriding methods (Cont’d)

Example (Cont’d):

• we need is to override these methods as shown below
• Note that the other methods of the Robot class, like moveTo and getXPosition, still work perfectly well for SpeedRobot objects

```java
// new version, overriding previous methods
public void moveNorth ()
{
    setY (getY() + speed);
}
public void moveSouth ()
{
    setY (getY() - speed);
}
public void moveEast ()
{
    setX (getX() + speed);
}
public void moveWest ()
{
    setX (getX() - speed);
}
```
Overriding methods (Cont’d)

Example (Cont’d):

• Here we show an example of using both the original and the new, overridden versions of these methods.

```java
SpeedRobot gonzales;
gonzales = new SpeedRobot(5); // 5 steps
Robot crawler;
crawler = new Robot(); // create normal one
...
// invoke original, then overridden method
crawler.moveNorth(); // moves one square only
gonzales.moveNorth(); // speedy robot moves 5
...
gonzales.moveTo(3, 3); // just like a Robot
crawler.moveTo(3, 3); // it is a Robot!
// now they are in the same place
...
crawler.moveEast(); // moves one square only
gonzales.moveEast(); // speedy robot moves 5
// now they are in different places again
...```
5. Inheritance

Overloading methods

- A class may have more than one method of the same name, but with different signature.
  - Overloading is not directly related to inheritance, but it is important to understand overloading when dealing with classes in an inheritance hierarchy

Example

We could extend the SpeedRobot class to have alternative versions of each move method

- one that always moves the current standard step size
- and the other that accepts an argument specifying the size for that step only
5. Inheritance

Overloading methods (Cont’d)

Example (Cont’d)

The class definition →

```java
public class VariableSpeedRobot extends SpeedRobot {
    ...
    // define OVERLOADED method – not OVERRIDING
    public void moveNorth (int theSpeed) {
        setY (getY() + theSpeed);
    }
    ...
    // and for other directions, MoveEast etc.
    ...
}
```

Invoking the methods →

```java
VariableSpeedRobot sillyWalker;
// create a speed robot with "normal" speed 5
sillyWalker = new VariableSpeedRobot(5);
...
sillyWalker.moveNorth(); // move 5 steps
sillyWalker.moveNorth(3); // 3 steps this time
sillyWalker.moveEast(); // move 5 steps
sillyWalker.moveEast(2); // 2 steps this time
..."