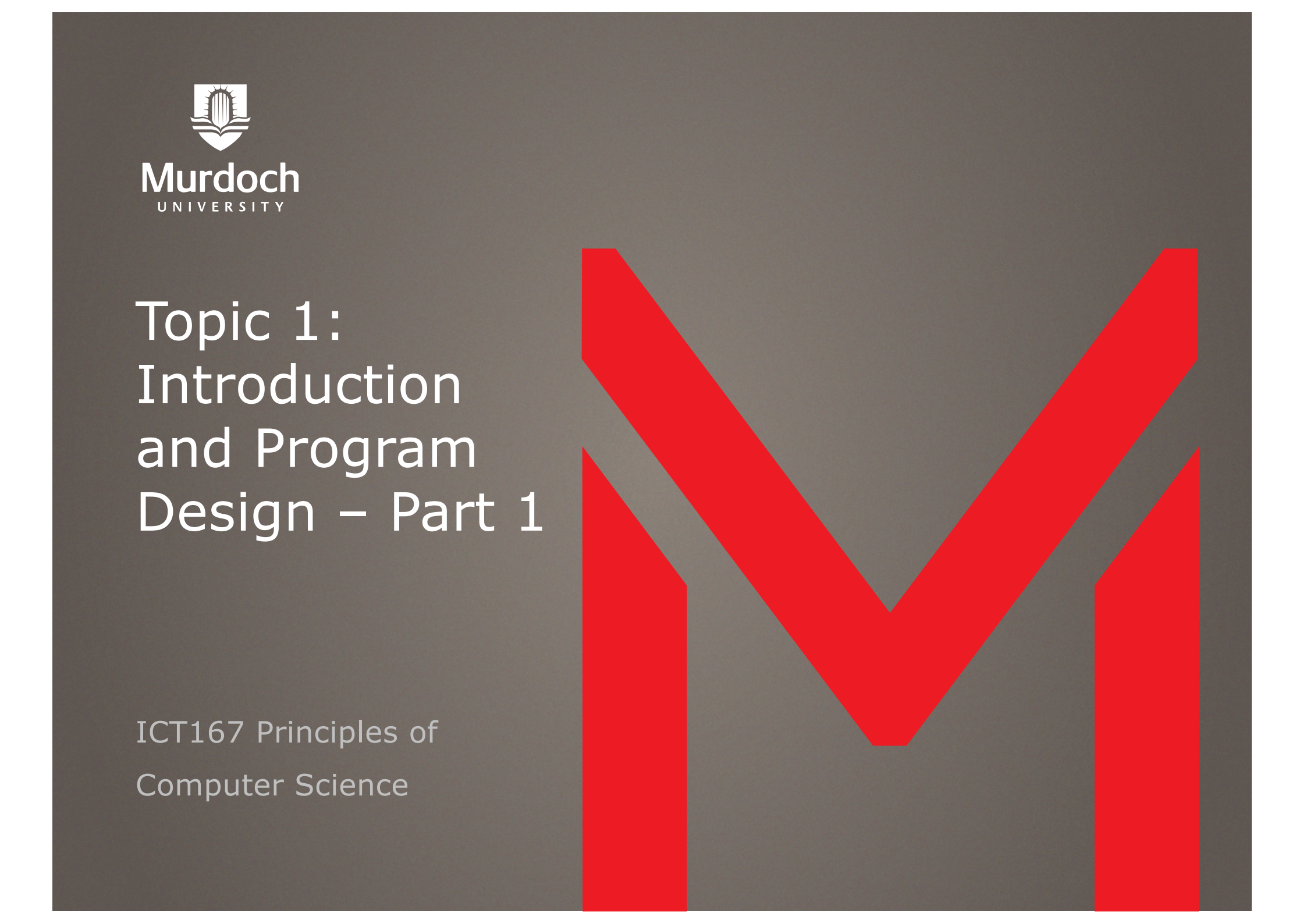

Topic 1: Introduction and Program Design – Part 1

ICT167 Principles of Computer Science

Murdoch UNIVERSITY

Topic 1 – Part 1

- Objectives
  - Understand the nature and role of programming in modern computing
  - Understand the main steps involved in programming
  - Explain the difference between high-level and low-level languages and give examples
  - Describe the steps involved in executing a high-level program
    - That is, compiling, linking and interpreting
  - Explain why Java is a popular language

Topic 1 – Part 1

- Explain the role of the Java SDK, compiler, interpreter, byte-codes, and the JVM in developing Java software
- Give correct Java syntax for very basic programs
- Describe the steps involved in running a Java program
- Produce simple standard output in Java
- Be able to use the **Scanner** class of Java API for simple standard input
- Know how to recognize a simple infinite loop and stop the running of a program

Topic 1 – Part 1

- Be able to use the primitive data types of Java
- Be able to write arithmetic and Boolean expressions in Java
- Be able to use the Java constructs for sequence, selection and repetition

Reading:

Savitch Chapters 1, 2, 3 and 4

Recommended self test questions:

Chapters 1.3, 1.4, 2.1 and 2.2

Programming and Software Development

- Developing a piece of software or a system involves many skills and usually many people
- These days software is often complex and it is important to get it to work correctly and for it to be easy to use and easy to maintain
- Systems analysts, software architect/designers and software engineers all play a part in managing this complexity

Programming and Software Development

- Also, a lot of the hard work has been done and it is out there to be used in the form of friendly operating systems, high-level languages, libraries of code and networks which are already set up and easy to use
- However, there is still a very important role for people who can do the small fiddly bits: the programmers

Programming and Software Development

- To become a good programmer you will need to:
  - Have a commitment to getting the small details right and
  - Know how all bits fit together. That is,
    - Know how to write code that can be used in a bigger system
    - Know what the rest of a team wants from your code
    - Know the quickest and most reliable ways of getting the job done

Programming and Software Development

- Programming is a creative activity
- You will be given a description of a finished product
- For example: “it is round, green and tastes like almond with the hint of an after-taste of elderberry and it feeds ***n*** people”
- You have to invent a recipe which is guaranteed to work (for any ***n***) and which can be followed by a real moron with absolutely no common sense

Writing a Program

- Using a computer for problem solving involves **four steps:**
  1. ***Establish the requirements: specifying the problem*** in terms of
     - The input data to be supplied
     - The tasks to be performed and
     - The output results to be produced
  2. ***Create a design: devising an algorithm***, or sequence of steps, by which the computer can produce the required output from available input

Writing a Program

3. ***Implement the code: expressing the algorithm as a computer program*** in a programming language such as Java
4. ***Test the implementation: debugging and testing the program*** to eliminate errors so that the program produces the intended output every time it is executed
   - You can only prove the presence of bugs, not their absence – computing proverb

- The process of ***documentation*** runs as a thread through all four steps above

High/Low Level Languages

- Programs may be written in a variety of languages. For example: Fortran, COBOL, Pascal, Smalltalk, Ada, LISP, PROLOG, Visual Basic, Delphi, C, C++, Java, C#, Python, Ruby
- High-level languages (vaguely) resemble human languages but have strict syntax rules
  - The above is a list of popular high-level languages
- Computers only understand machine code (0's and 1's). Assembly languages use mnemonics to make it less difficult to write machine code

High/Low Level Languages

- Assembly languages are called *low-level* languages
- Nearly all programmers can work with high-level languages to be most productive
- There are some specialized jobs which require assembly language programming skills: like developing new computers and languages, robots, embedded systems, or where speed is essential (eg: in games)

High/Low Level Languages

- Most programmers don’t need to be experts in the use of assembly language (although some understanding of assembly language is desirable in order to fully understand the low level functions)

Executing High-level Source Code

- Programs written in high-level languages are almost always just put in text files
  - Any text editor (notepad, vi, pfe, gvim, JBuilder, Eclipse, **NetBeans**) will let you type the program in
- The text version is known as the *source code* or *source program*
- A program must be translated into machine code before it can be executed on a particular type of computer (CPU)
- This can be accomplished in several ways

Executing High-level Source Code

- A ***compiler*** is a software tool which translates a source program into a specific target (low-level) language the computer can run
  - At this stage, the program is not yet running
  - Various errors may show up in this step
- There is also (usually) a ***linking*** step, in which other pieces of code (usually compiled) are found from elsewhere (in the current or another directory) and put together with the current program to produce the *target program*

Executing High-level Source Code

- Often the target program produced is in the machine language of a particular computer (CPU), which is then run to produce results
- The Java approach is somewhat different from the above approach

Java Translation and Execution

- The Java compiler translates Java source code into a special representation called ***byte-code***
- Java byte-code is not the machine language of any computer - it is platform independent
  - As in the previous approach, a ***linking*** step (**class loader**) is involved, in which other pieces of code (usually compiled) are fetched from elsewhere (in current or the another directory or even across a network from another machine) and put together with the current program to make code which can be run immediately

Java Translation and Execution

- An **interpreter** (another program) translates each byte-code instruction into machine language and executes it
- This interpreter is called **Java Virtual Machine** (JVM) – a part of the JDK and the foundation of the Java platform
- If the same JVM is available on many platforms, applications that it executes can be used on all those platforms

Java Translation and Execution

- Thus the Java compiler is not tied to any particular computer
- Java is considered to be ***architecture neutral***

Why Learn Java?

- Java is a popular choice for implementing Internet-based applications and software for devices that communicate over a network
- It is an **object-oriented (OO) language** so, in designing and implementing Java software, we will be using objects and modelling with concepts such as classes
  - OOP is today's key programming methodology

Why Learn Java?

- There will be benefits for re-use of software and clean design of larger applications
- We will learn the basics of OOP in this unit but there will be much more (such as **inheritance**, **polymorphism** and **dynamic binding**) to leave for later
- The basics of OOP also apply to the another popular C++ language

Why Learn Java?

- Java was developed from a language which could be used on many different machines (like toasters, VCRs, televisions and smart phones) and is *architecturally neutral*
  - A program can run in a standard way on a variety of different types of machines and under a variety of operating systems

Why Learn Java?

- As well as being architecturally neutral, Java has lots of other facilities to allow ease of use with Internet-based applications
  - These include applets, classes for URLs and networking, servlets, JSPs, JSFs and very convenient GUI (graphical user interface) tools
- We will not study these applications in any depth in this unit but all such applications need basic knowledge of Java programming

Compiling, Interpreting and JVM

- The standard JDK compiler is called "**javac**"
  - Use it to compile a text file containing java source code
- The result is a file (with a **.class** extension) called the Java **byte-code** and is analogous to executable code
- To execute the byte-code file, you need an **interpreter**
  - Which one to use depends on the environment and the operating system

Compiling, Interpreting and JVM

- For command line running of ordinary programs (called ‘application programs’), we can use the interpreter “**java**” (appropriate versions of this are supplied for Windows, Linux, Mac OS and Solaris)

Compiling, Interpreting and JVM

- IDEs use "**java**" or their own versions
- Most modern browsers can run compiled **applets**
- The compiled Java program, in byte-code, should run in (almost) exactly the same way under different operating systems (Java is architecture neutral)
- We say that the byte-code is being run on a **Java Virtual Machine** (JVM) and that various interpreters are implementations of the JVM

Compiling, Interpreting and JVM

- So, Java source code can be compiled on any machine (with JDK), the compiled code can be sent to any other machine (eg: over the Internet) and that program will run as planned provided the second machine has an implementation of the JVM (for example, in a browser
- This makes Java suitable for Internet applications

Differences Between Java and C

- The syntax of both languages is very similar but with a number of minor (but important) differences
  - The selection (`if, if-else, switch-case`) and iteration (`for, while, do-while`) constructs you have used in C are virtually identical in Java except that Java has a `boolean` data type (and keywords `false/true` for use in boolean expressions)

Differences Between Java and C

- C is a compiled language meaning a C program compiled for one computer system will not run on a system with a different type of CPU and different type of operating system
  - Java runs on the JVM and is architecture neutral
- C (like most programming languages) uses the ASCII character set whereas Java uses the *Unicode* character set
  - The Unicode character set includes the ASCII character set and characters from many different alphabets (but you probably won't use them)

Differences Between Java and C

- The I/O is different:
  - C uses functions like `scanf()` and `printf()` for input and output respectively
  - Java uses methods of library classes like `Scanner` and `System.out` for input and output respectively
- Parameter passing is different:
  - In C, programmer specifies whether they want to pass a parameter by value or by reference
  - In Java, the language decides automatically
    - Any primitive data types are automatically passed by value
    - Any complex data types (objects) are passed by reference

Differences Between Java and C

- Java is an object-oriented language where as C is a procedural language
  - ***This is the biggest difference between the two languages***
- OOP is a very powerful way of developing software and focuses on abstraction, information hiding, composition and inheritance among other things (more of this later)

An Example Java Program

```
// HelloClass.java

public class HelloClass
{
  public static void main(String[] args)
  {
     System.out.println("Hello Class");
  } // end main
} // HelloClass
```

An Example Java Program

- Note:
  - There is a **class** containing a **main** method
  - You might not know what "**public** ..." means in two of the lines, but you must put them in
  - The name of the file **must** be exactly the same as the name of the class, with an extension "`.java`"
    - Java is case sensitive, so be careful when naming the class and the file

An Example Java Program

- Compiling on command line:

```
javac HelloClass.java
```

- Executing on command line:

```
java HelloClass
```

- This program should result in the following output on the command line:

```
Hello Class
```

- There will be an information sheet on using Java and NetBeans IDE in the labs

Standard IO in Java

- Many operating systems allow programs to mention a standard source of input (called standard input) and a standard place for output to go (called standard output)
- By default, keyboard activity goes into the standard input and standard output goes to (a window on) the screen (unless you redirect it elsewhere)

Standard IO in Java

- It is easy to command the operating system to get standard input for a program from somewhere else (like a file) or to send standard output somewhere else
- Input/Output (IO) is quite a complicated business (with ends of lines and invisible characters, etc.) but programming languages usually provide simple facilities for standard IO

Standard IO in Java

- To send a line to the standard output, Java allows you to just write:

```
System.out.println("a line");
```

  - **System.out** is an object for sending output to the screen
  - **println()** is a method to print whatever is in parentheses
- If you do not want to finish with a new line, use:

```
System.out.print("a line");
```

Standard IO in Java

- The string of characters contained between the double quotation marks is called a **character string** or a **string literal**
- White-space characters in strings are **not** ignored by the compiler
- Strings cannot span multiple lines of code

Standard Input

- Standard input will come into a program from the keyboard (unless you redirect it)
- Java is unusually inconvenient when it comes to standard input
- This is because Java has strict rules for error handling
- Inputs can cause errors (like a letter arriving when a number is expected) and so these errors should be handled

Standard Input

- Java expects to be told what to do if an input error arises
  - The programmer should tell the program what to do
- Anyway, these days, most serious programs use GUIs for IO (and this is quite easy in Java)
- Later in the unit you will see how to handle standard input yourself and how to set up simple GUI input

The Scanner Class

- The `Scanner` class (in `java.util` package) is available as part of the standard Java library
- It provides convenient methods for reading input values of various types
- The input values can come from various sources including standard input (keyboard) or a file

The Scanner Class

- When reading data from the keyboard, we are reading from the standard input stream, which is represented by the `System.in` object in a Java program
- To create a Scanner class variable (object), use

```
Scanner input = new Scanner(System.in);
```

- Here identifier `input` is the programmer defined variable name and Scanner is the type of this variable
  - We say `input` is a Scanner object

The Scanner Class

- After the above Java statement, methods of Scanner class (such as `nextInt()`, `nextDouble()`, `next()`, and `nextLine()`) can be used with the object **input** to read data (of a particular type) from the keyboard

The Scanner Class

- Note that in order to make the Scanner class available to your program, the following line must be inserted in the beginning of the program:

```
import java.util.Scanner;
```

or

```
import java.util.*;
```

An Example

```
// File name: SmallIO.java
import java.util.Scanner;

public class SmallIO{
   public static void main(String[] args){
    Scanner keyboard = new Scanner(System.in);
    String a = "";    // initialise to empty string
    while (true){
      //an infinite loop, use Ctrl-C to quit
      System.out.println("Enter a line:");
      a = keyboard.nextLine();
      System.out.println("Your line: " + a);
      System.out.println();  // print a blank line
    } //end of while
   } //end of main
} //end of class
```

The Scanner Class

- It is NOT a good design to trap a user in an infinite loop but as a user (and debugger) of your own programs, remember to use Ctrl-C (i.e. press the Ctrl and C keys together) to quit such a program when in command prompt mode
- If running from NetBeans IDE, use:

```
Run | Stop Build/Run
```

Some Scanner Class Methods

- `Scanner_object_name.`**`nextInt()`**
  - Returns next int value
- `Scanner_object_name.`**`nextFloat()`**
- `Scanner_object_name.`**`nextLong()`**
- `Scanner_object_name.`**`nextDouble()`**

Some Scanner Class Methods

- `Scanner_object_name.next()`
  - Returns next keyboard characters up to, **but not including**, the first delimiter character
  - Unless specified otherwise, white spaces, tabs and newlines are used to separate the elements of input from each other - these characters are called ***default delimiters***
- `Scanner_object_name.nextLine()`
  - Returns the rest of the input line as a string
  - The end-of-line character '\n' is read and discarded, it is not included in the string returned

Standard Output: printf

- In addition to the `System.out.print()` and `System.out.println()` methods for standard output, Java 5.0 introduced a `System.out.printf()` method similar to C's `printf()` function
- Eg: *(what will the following print on the screen?)*

```
System.out.printf( "%n%s%n%s%n",
"Welcome to ICT167 !",
"The Unit Coordinator is Kevin
 Wong");
```

Primitive Java

- Java Program Structure:
  - A Java program is made up of one or more **classes;** each class is normally in a separate file
  - A class contains one or more **methods** which perform tasks in the program
    - The item(s) inside parentheses are called **argument(s)** and provide the information needed by methods
  - A method contains program **statements** that perform the method's tasks
    - Each statement ends with a semicolon
  - A Java application always executes the **main** method

Java API

- The Java *Application Programming Interface (API)* is a collection of classes (*class libraries*) that can be used as needed to support program development
- The classes in a class hierarchy are often related by inheritance
- The classes in the Java API are separated into packages which can be nested

Java API

- The `System` class, for example, is in package `java.lang`
- Each package contains a set of classes that relate in some way
- For example, the `print` and `println` methods are part of the Java API; they are not part of the Java language itself

Java API

- Using a class from the Java API can be accomplished by using its fully qualified name: `java.lang.System.out.println();`
- Or, the package can be imported using an *import* statement, which has two forms:

```
import javax.swing.*;
import java.util.Random;
```

- The `java.lang` package is automatically imported into every Java program

Comments

- Used to document programs and improve their readability

```
// indicates that the line is a
comment
```

- A comment that begins with '//' is an end-of-line comment - it terminates at the end of the line on which it appears
- Traditional comment, can be spread over several lines as in

```
  /* This is a traditional comment.
It  can be split over multiple lines
*/
```

Comments

- Blank lines, spaces and tabs are known as whitespace and make programs easier to read
- Compiler ignores comments, blank lines and whitespaces

Class Declaration

- Every Java program consists of at least one class that you define
- `class` keyword introduces a class declaration and is immediately followed by the class name
- Keywords are reserved for use by Java and are always spelled with all lowercase letters
- By convention, class names begin with a capital letter and capitalize the first letter of each word they include (e.g., HelloClass)

Class Declaration

- Java is **case sensitive** - uppercase and lowercase letters are distinct - so n1 and N1 are different (but both valid) identifiers
- A left brace ‘{‘ begins the body of every class declaration and a corresponding right brace ‘}’ must end each class declaration
- The code between braces should be indented

The `main` Method Declaration

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

- The main method is the starting point of every Java application and must be defined as shown, otherwise the JVM will not run the application
- Java class declarations normally contain one or more methods
- Methods perform tasks and can return information when they complete their tasks

The `main` Method Declaration

- The keyword `void` indicates that this method will not return any information
- The body of a method must be enclosed in left and right braces

Primitive Data Types

- A data type is defined by a set of values and the operators that you can perform on them
- Each value stored in memory is associated with a particular data type
- The Java language has several predefined primitive types
- The following reserved words represent seven different primitive data types:
  - `byte, short, int, float, double, boolean, char`

Primitive Data Types

Type Name	Default value	Memory used	Default value	Range of values
byte	integer	1 byte	0	-128 to 127
short	integer	2 bytes	0	-32,768 to 32,767
int	integer	4 bytes	0	-2.147,483,648 to 2.147,483,648
long	integer	8 bytes	0	-2^{63} to $(2^{63}-1)$
float	Floating-point	4 bytes	0.0	$+-3.40282347 \times 10^{+38}$ to $+-1.40282347 \times 10^{-45}$
double	Floating-point	8 bytes	0.0	$+-1.7976931348623157 \times 10^{+308}$ to $+-4.9406564584124654 \times 10^{-324}$
char	Single char (Unicode)	2 bytes	'\0'	Each values from 0 to 65535 represents a character in the Unicode character set
boolean		1 bit	false	true or false

Variables

- Each variable in a Java program has to be declared to be of a particular type
- This is so that the compiler (and the reader of code) can know what kind of values a variable can have. The compiler can allocate storage and check for stupid errors
- The variable may be of a *primitive type* like int, boolean, double, char etc. The variable can hold one of these simple values directly

Variables

- Declare via:

```
int count;
double sum, average;
```

- Declare and initialize via:

```
boolean flag= true;
```

- All other variables in Java are of a ***Class type***

Naming Conventions

- Class types begin with an uppercase letter (e.g. String)
- Primitive types begin with a lowercase letter (e.g. int)
- Variables of both class and primitive types begin with a lowercase letter (eg: studentName, studentNumber)
- Multi-word names are "punctuated" using uppercase letters

Named Constants

- Java provides a mechanism to define a variable, initialise it, and fix the value so it cannot be changed
- Eg:

```
public static final double PI =
3.14159;

public static final int MAX_COUNT =
100;
```

Assignment Statements

- An **assignment statement** takes the form:

  `variable = expression;`

- The `expression` is evaluated and the result is stored in the variable, overwriting the value currently stored
- The `expression` can be a single value or a more complicated calculation involving operators and operands

Assignment Capabilities

- Java is a *strongly typed* language
  - That is, you can not assign a floating point value to a variable declared to store an integer
- Sometimes conversions between numbers are possible
  - Eg: you can assign a value of an integer type to a variable of a floating-point type

```
double interestRate = 7;
```

Note the value of `interestRate` is 7.0

Assignment Capabilities

- A value of one type can be assigned to a variable of any type further to the right below:

```
byte -> short -> int -> long -> double
```

but not to a variable of any type further to the left. For example:

```
int todays_rate;
todays_rate = interestRate;
```

is illegal because `interestRate` is of type double

Type Casting

- A ***type cast*** can be used to change the data type of a value from its declared type to some other type
- Eg: the above example can be written as:

```
todays_rate = (int)interestRate;
```

  which is now legal
- Any nonzero value to the right of the decimal point is *truncated* rather than *rounded*

Java Arithmetic Operators

- Arithmetic expressions can be formed using the +, -, *, and / operators together with variables or numbers referred to as *operands*
  - When both operands are of the same type, the result is of that type
  - When one of the operands is a floating-point type and the other is an integer, the result is a floating point type

Java Arithmetic Operators

- The division operator (/) behaves as expected if one of the operands is a floating-point type
  - When both operands are integer types, the result is truncated, not rounded, eg: 60/100 gives a zero value
- The modulus (mod) operator (%) is used with operators of integer type to obtain the remainder after integer division (example next slide)

Java Arithmetic Operators

- Eg: 18 % 4 gives 2

  That is, 18 / 4 = 4, with 2 left over
- The mod operator has many uses, including determining
  - if an integer is odd or even
    - Eg: if ((num % 2) == 1) // true if odd
  - if one integer is evenly divisible by another integer
    - Eg: if ((num1 % num2) == 0) // true if even

Parentheses and Precedence

- Parentheses can communicate the order in which arithmetic operations are performed

```
eg (itemPrice – discount) *
numberOfItems
```

- Without parentheses, an expressions is evaluated according to the *rules of precedence*

Parentheses and Precedence

Precedence level	Operators
1st Highest precedence	Unary operators +, -, !, ++, and --
2nd Highest precedence	Binary arithmetic operators *, /, and %
Lowest precedence	Binary arithmetic operators + and -

- When binary operators have equal precedence, the operator on the left acts before the operator(s) on the right

Specialized Assignment Operators

- Assignment operators can be combined with arithmetic operators (+, -, *, /, and %)
- Eg:

```
sum = sum + number;
```

can be written as

```
sum += number;
```

giving the same result

Increment / Decrement Operators

- Used to increase (or decrease) the value of a variable by 1
- The increment operator count++ or ++count
- The decrement operator count-- or –count
- Easy to use – however it is important to recognize the difference:
- Eg:

```
int m = 5;  int n = 7;
int result= m *(n++); // post increment
of n
```

Increment / Decrement Operators

- What are the values of m, n and result after execution of the above code?

```
int m = 5; int n = 7;
int result= m *(++n);// pre increment of n
```

- Now what are the values of m, n and result?

Java Comparison Operators

- The comparison operators:
  - equivalent to (==)
  - not equivalent to (!=)
  - greater than (>)
  - greater than or equal to (>=)
  - less than (<)
  - less than or equal to (<=)

  are available for use in boolean expressions

Java Selection Statements

- An **if statement** has the form:

```
if(expression)
   statement;
```

- The expression is a boolean expression which must evaluate to a true or false result
- If the expression is true, the statement is executed. Otherwise the statement is skipped

Java Selection Statements

- An **if-else statement** has the form:

```
if(expression)
   statement1;
else
   statement2;
```

- If the `expression` is true, statement1 is executed
- If the `expression` is false, statement2 is executed

Java Selection Statements

- Several statements can be grouped together into a **block statement** by enclosing them in braces '{' and '}'
- The body of an 'if' statement or 'else' clause can be another if statement - called **nested if statements**

Java Selection Statements

- A **switch statement** has the general form:

```
switch(expression)
{
   case value1:
      statement-list1;
      break;
   case value2:
      statement-list2;
      break;
    .
    .
```

Java Selection Statements

```
    .
    .
    case valuen:
       statement-listn;
       break;
    default: // code to handle other
     cases
       break;
} // end switch
```

Java Selection Statements

- The `expression` must evaluate to an integral value such as an integer or character
- Java 8 (jdk 1.8) will allow strings in expressions

Java Repetition Statements

- A **`while`** statement has the form:

```
while(boolean-expression){
   statement(s);  // loop body
}
```

- If the `boolean-expression` is true, the statement (called body of the loop) is executed and then this `expression` is evaluated again

Java Repetition Statements

- The above step is repeated until the `expression` becomes false
- If the `expression` is false initially, the statement is never executed

Java Repetition Statements

- A **`for`** statement has the form:

```
for(initialisation;test-expression;update-expression){
    statements;   // loop body
}
```

- Like a `while` loop, the `test-expression` of a `for` loop is tested prior to executing the loop body
- Like a `while` loop, a `for` loop executes zero or more times

Java Repetition Statements

- The initialisation is performed only once, but the `update-expression` is executed after each iteration
- Note that each expression in a `for` statement is optional:
  - If `initialisation` is left out, none is performed
  - If the `test-expression` is left out, it is always considered to be true - results in infinite loop!!
  - If the `update-expression` is left out, no update is performed

Java Repetition Statements

- A **`do-while`** statement has the form:

```
do{
   statement(s);  // loop body
} while(boolean-expression);
```

- The loop body is executed at least once
- The above process is repeated until the `boolean-expression` becomes false

Nested Statements

- Note that selection and repetition statements can contain any sort of statements within them including other `if`, `switch-case` and looping (`while`, `for`, `do-while`) statements

Java Logical Operators

- Boolean expressions can be combined using the java "and" (&&) operator and “or” (||) operator
- Eg:

```
if ((score >= 0) && (score <= 100))
 System.out.println(score);
if ((n1 < 0) || (n2 <= 0))
 System.out.println(“Error – invalid data”);
```

- Note: the following is **not** allowed:

```
if (0 < score <= 100)  ….
```

Java Logical Operators

- A boolean expression can be negated using the "not" (!) operator
- Eg:

```
boolean isValid = (score >= 0) && (score <=
100);
if (! isValid)
System.out.println("Invalid score");
```

The Conditional Operator

- The following code:

```
if (n1 > n2)
  max = n1;
else
  max = n2;
```

- can be written as

```
max = (n1 > n2) ? n1 : n2;
```

- The conditional operator is useful with print and println statements

```
System.out.print("You worked " +
                        ((hours > 1) ? "hours" :
"hour"));
```

Examples:

// output numbers 0 to 9 : uses **while** loop

```
int count;
count = 0;                  //
   initialisation
while(count < 10)           // test
{
  System.out.println(count);
  count = count + 1; //
   incrementation
} // end while
```

Examples:

// output numbers 0 to 9 : uses **for** loop

```
for(int i = 0;i < 10; i++)
{
  System.out.println(i);
} // end for
```

Examples:

// output numbers 0 to 9 : uses **do-while** loop

```
int count;
count = 0;                    //
   initialisation
do {
  System.out.println(count);
  count = count + 1; //
   incrementation
} while(count < 10); // test
```

Murdoch UNIVERSITY

Another Example:

// File: ScannerDemo.java

```
import java.util.*;     // for Scanner class
public class ScannerDemo
{
  public static void main(String[] args)
  {
     Scanner keyboard = new Scanner(System.in);
     System.out.print("Enter two whole numbers ");
     System.out.println("separated by one or more
                                          spaces:");
     int n1, n2;
     n1 = keyboard.nextInt( );
     n2 = keyboard.nextInt( );
     System.out.println("You entered "+n1+" and "+n2);
```

Another Example:

```
  System.out.println("Next enter two numbers.");
System.out.println("A decimal point is OK.");

double d1, d2;
d1 = keyboard.nextDouble( );
d2 = keyboard.nextDouble( );
System.out.println("You entered "+d1+" and "+d2);
System.out.println("Next enter two words:");
String s1, s2;
s1 = keyboard.next( );
s2 = keyboard.next( );
System.out.println("You entered \""+s1+"\" and \""
                                        +s2+"\"");
s1 = keyboard.nextLine( ); // To get rid of
// newline char '\n' - this is important !!!!!
```

Another Example:

```
        System.out.println("Next enter a line of text:");
        s1 = keyboard.nextLine( );
        System.out.println("You entered: \"" + s1 + "\"");
    } // end main
} // end class
```

End of Topic 1 – Part 1

Murdoch UNIVERSITY