Programming Design and Implementation

Lecture 2: Programming Basics

Dr David McMeekin
Discipline of Computing
School of Electrical Engineering, Computing and Mathematical Sciences (EECMS)

COMP1007 - Unit Learning Outcomes

- Identify appropriate primitive data types required for the translation of pseudocode algorithms into Java;
- Design in pseudocode simple classes and implement them in Java in a Linux command-line environment;
- Design in pseudocode and implement in Java structured procedural algorithms in a Linux command-line environment;
- Apply design and programming skills to implement known algorithms in real world applications; and
- Reflect on design choices and communicate design and design decisions in a manner appropriate to the audience.

COMP5011 - Unit Learning Outcomes

- Develop and apply simple non-object oriented algorithms;
- Develop and implement simple classes in an object oriented language;
- Create object oriented designs consisting of classes connected by aggregation; and
- Communicate design and design decisions in a manner appropriate to the audience.

Outline

Development

Software Development

- Problem Definition

  Understand the problem

  Think/consult/revise

- Design a solution (flow chart and pseudocode, UML)

  Specify the steps involved

  Create the algorithm

- Test the solution

  Does your design work (in theory)

- Fix the solution

  Correct your design where it failed

- Write and document the code

  Testing that the code works

  Ensuring the code is maintainable

Problem Definition

- Understand the problem that needs to be solved
- Start with a "top Level" general English description of the problem
- State what data is required to solve the problem
  - The input data
- State what results will be calculated
  - The output data
- What actions need to be taken to generate the results
- Crucial part of solution is to know what the problem is, but this is often ignored by poor software developers
- Need to consider security, reliability and performance requirements

Design a Solution - Algorithm

- An algorithm is a set of detailed, unambiguous, ordered steps specifying a solution to a problem
  - Steps must be stated precisely, without ambiguity (pseudocode, next slide)
  - Enter at the start & exit at the bottom
  - English description independent of any programming language
  - Non trivial problem will need several stages of refinement
  - Various methodologies available

Algorithm - Pseudo Code

- Algorithms are expressed in Pseudo Code:
  - English like phrases which describe the algorithm steps
  - Pseudo code is NOT a programming language
  - It is PSEUDO
  - Pseudo code development is about refinement
    - Developing an algorithm is a journey (to enjoy :-)
  - Algorithm design is an art that takes a lot of practice

Using Pseudocode

- What is it with Psuedocode:
  - Clear;
  - Logical;
  - Understandable;
  - Consistent; and
  - Correct.

Pseudo Code - Simple Example

- Problem

  Write a program to sum 2 numbers input from the keyboard (user). Output the result to the screen (user).

- Algorithm:

```
MAIN:
   INPUT numOne
   INPUT numTwo
   sum = numOne + numTwo
   OUTPUT sum
END_MAIN
```

Test the Solution

- Desk check the algorithm
- Walk through the algorithm step by step
  - Was it complete?
  - Did you get all the way through the logic?
  - Did you get an answer?
  - Was the answer correct?
- Answered NO to any of the above? Error in your algorithm

Fix the Solution

- Return to “Understand the Problem”
  - What did you mis/not understand?
- Return to “Design a Solution”
  - With new understanding, fix your design
- Return to “Test the Solution”
  - Walk through your test cases again
  - Did you answer NO to any of those questions
  - Rinse and Repeat until your solution is correct

Write and Document the Code

- Convert algorithm description into implementation of HLL Higher Level Language. e.g., Java.
  - Known as coding
  - Programmer needs to know the semantics and the syntax of the language
  - The files of HLL statements are called the **source files**
  - Write the documentation so the code can be easily maintained

Java

To be OO or not to be OO?

- There are 2 basic paradigms for designing imperative algorithms:
  - Procedural:
    - Focus on the steps required to perform the task.
    - The design of the steps lead to the types of data structures that will be required.
  - Object Oriented:
    - Focus is on the entities required. (i.e., What do we need to represent in the algorithm?)
    - What functionality each thing will require.
    - How these things will communicate with each other.
    - Each entity will be represented as an object.
    - The design of each object leads to the steps required.
- Covered in detail later in the semester.

Introduction to Java

- Java is an Object Oriented (OO) language (its roots in C & C++, covered in another unit);

- James Gosling designed Java as a ”small” OO language in 1990-91:
  - Initially aimed at information appliances, but in 1993 adapted for animation & interaction on WWW;
  - Introduction of Netscape in 1995 allowing Java applets, led to its continued popularity.

Interpreting

- Process whereby the file of source code is translated a line at a time into machine code instructions that can be executed by the machine.
  - If syntax errors exist then the program will be partially executed
  - As soon as syntax errors are encountered, execution halts
- Python and Ruby are both interpreted HLL
- Interpreting code is slower than executing compiled code, because syntax checking & translation has already been done with compiled code
- If syntax errors can be eliminated before the source code is interpreted then the syntax error problem is avoided

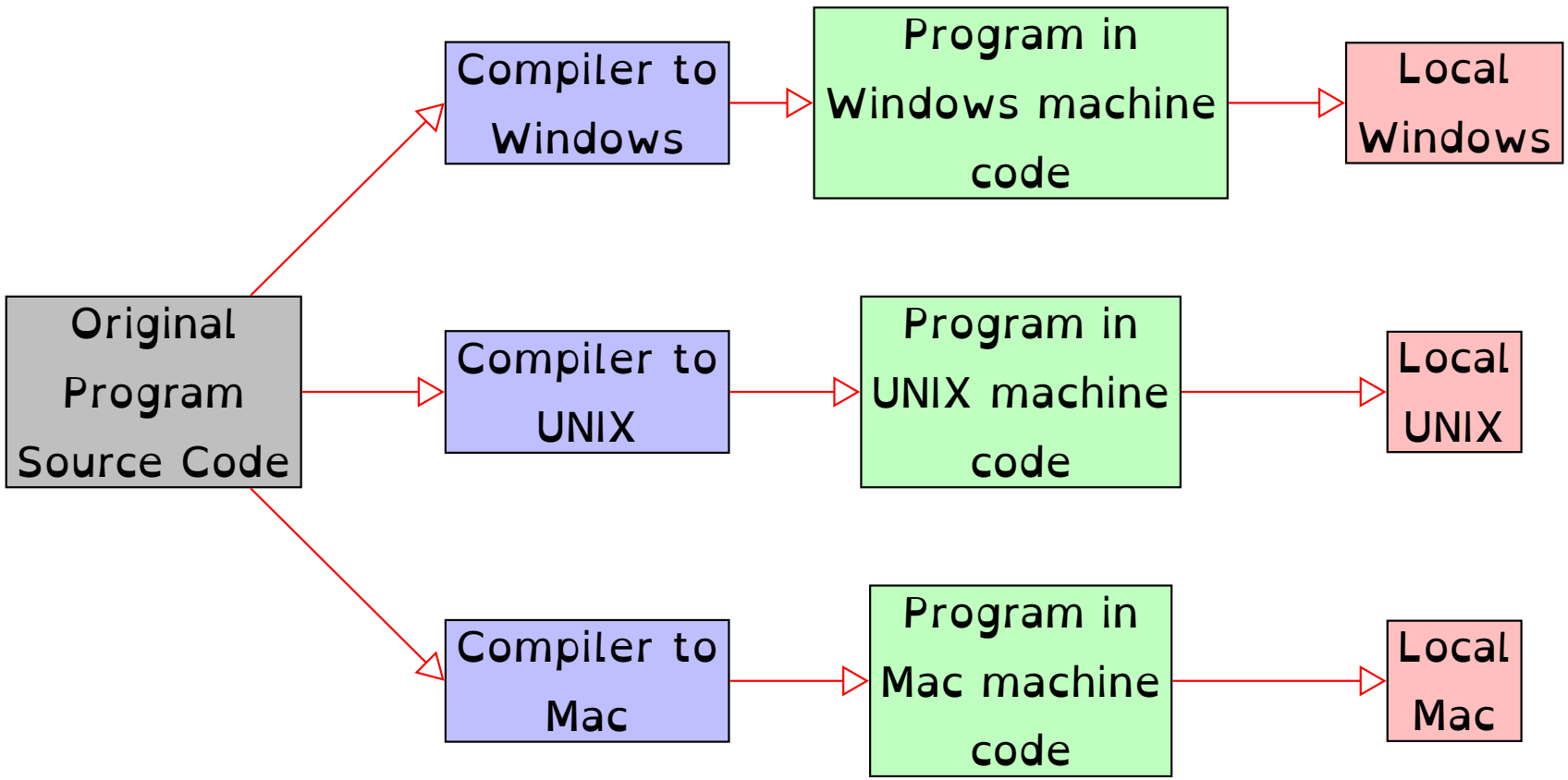
Compiling

- Compilation:
  - Process whereby the file of source code is translated into machine code
  - The source code is checked to ensure it conforms with grammar (syntax and semantics) rules
  - If no syntax errors found at compile time then machine code file is created and can be executed
  - If even just one error then machine code file does not exist and there is no machine code to run

Java Platform Independence

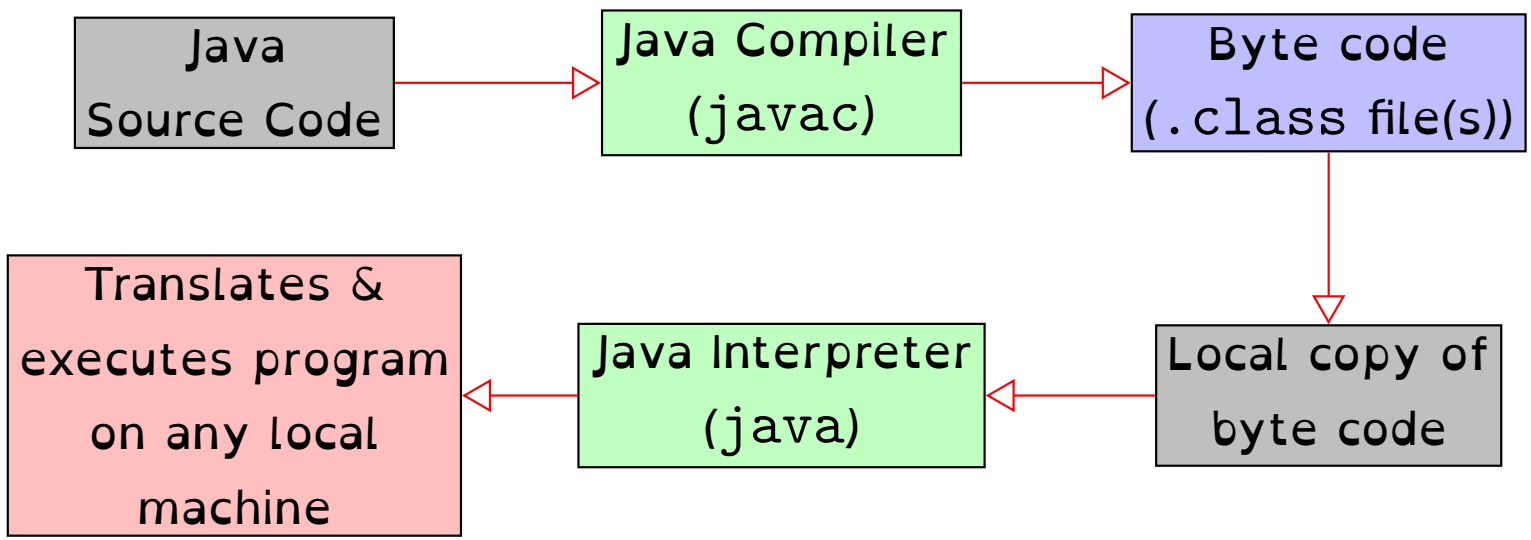
- Platform independence is achieved by running byte code on a Java Virtual Machine (JVM)
  - Byte code is machine code for the JVM. A Java compiler compiles from source code to byte code
- The JVM is itself a program whose job it is to interpret the Java byte code
- Each machine actually needs code in its own particular native machine language: thus byte code needs to be interpreted to the local machine code to be able to execute locally
- The overheads of interpreting byte code are lower than traditional interpreters because the conversion is from machine code (JVM) to machine code (native machine code)
  - Syntax checking has already been done at compile time

Traditional Methods: Compile -> Execute



Java Virtual Machine

- As long as local machine has the byte code interpreter it can download any Java program in byte code & execute it and yet:
  - Source code is secure
  - Only one version of the byte code needs to exist



Java: The Good and the Bad

- Good
  - Platform independent execution;
  - Platform independent binary data (files etc);
  - Robust;
  - Does not not allow operator overloading;
  - Comes with a huge class library facilitating:
    - File input/output;
    - Graphics;
    - Event trapping/handling; and
    - 3D modelling.
- Bad
  - Syntax adopted from C;
    - Therefore some control structures are primitive and unstructured;
- Relax, the good and bad will make more sense as you go.

Writing Java

Creating a Java program

- Design and write your algorithm first! (pseudo code)
- A text file with a file extension of `.java` must be created
  - In this file, you store the human readable Java program
- The Java compiler (known as `javac`) is then used to compile the source code into byte code
  - For each class (see later) defined in source code the compiler will create a corresponding file of byte code
  - The name of each byte code file will be the name of the class (as defined in the source code) with an extension of `.class`.
- The Java interpreter (known as `java`) is then used to translate the byte code and execute the resulting native machine code

Pseudocode

```
MAIN:
  message = "Hello World"
  OUTPUT message
END_MAIN
```

A Simple Java Application

```
import java.util.*;

public class MyFirstProgram
{
    public static void main(String[] args)
    {
        String message = "Hello World!";

        System.out.println(message);
    }
}
```

Creating, Compiling and Running

- The Java code on the previous slide is entered into a text file using a text editor (under Linux we recommend using `vim`).
- The name of the `.java` file **must** be exactly the same as the name of the class (i.e., `MyFirstProgram.java`)
- The `.java` file is then compiled into byte code
- The command would be:

```
[user@pc]$ javac MyFirstProgram.java
```

  - if the program contained errors, they will be displayed and **no** byte code is generated
  - else the byte code is produced
  - The byte code is stored in a file called `MyFirstProgram.class`

- To execute the program, use the command:

```
[user@pc]$ java MyFirstProgram
```

The Import Statement

- Java comes with an extensive library of classes
- The libraries make implementing algorithms much easier
- Many organisations develop their own class libraries
- In PDI we do not have our own class libraries
- We **must** understand the `import` statement
- The `import` statement tells the Java compiler that libraries are to be found by looking for the directory path specified

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

  - Means import all class library files in the `util` directory

Live Demo

- In this live demo we will look at:
  - Writing your very first Java program.

Variables

Variables, Constants and Literal Values

- Variable: a piece of memory in which data can be stored (and retrieved from).
- Has a name (also known as an identifier) associated with it;
- It must be declared:

```
int thisIsAnInteger;
```

- Constant, a variable whose initial value can *never* be modified.

```
public static final int MYCONST = 12;
```

- A literal value is a literal value, as shown below:

Integer	-12, 42, 0
Real	-10.2, 56.8, 0.0

Assigning Values to Variables

- The assignment operator '=' is used:

```
int thisIsAnInteger = 12;
```

  - what is on the right is **assigned** to what is on the left.
  - 12 is **assigned** to the variable **thisIsAnInteger**
- The value 12 is stored in **thisIsAnInteger**

Examples of Declaring and Assigning Values to Variables

```
int moSallah = 11;
int firmino = 9;

double latitude = 53.4308;
double longitude = 2.9608;

char codeLetter = 'a';
char initial = 'd';

String name = "Steven Gerrard";
String legend = "Valentina Tereshkova";

boolean theBest = true;
boolean theWorst = false;
```

Data Types

- The type of data to be stored in the variable;
- In Java, specified when the variable is declared;

```
//data type   name/identifier        value assigned
  double          latitude      =       53.4308;
  boolean         theWorst      =        false;
```

- Syntax of declarations is easy; the challenge is identifying the type
  - `triangleSideA` `//Probably a Real`
  - `yearOfBirth` `//This is obviously an Integer`
  - `studentName` `//This is a string of Characters`
    `//AKA: String`

Data Types (2)

- What about more complex variables:
  - `dateOfBirth` `//6 or 8 digit Integer`
    `//or 3 seperate Integers`
  - `age` `//Integer or Real`
  - `phoneNo` `//Integer or a String`

Live Demo

- In this live demo we will look at:
  - Variables; and
  - Assigning and changing variable values.

Primitive Types

Integer Data Types

- Integer: positive or negative value that is a whole number
- Java primitive types `byte`, `short`, `int` and `long`, all integer abstractions from the mathematical world
- The integer range is determined by the amount of storage available (memory) for a particular data type
- The accuracy is guaranteed
  - Stored as the exact base2 (Binary) equivalent of the base10 (Decimal) integer

Integer Range

- Determined by how many distinct base2 values can be stored in the given number of bits: every additional bit doubles the range size;
- For N bits: 1 bit for the sign, the remaining N-1 bits represent 2^{N-1} different combinations directly related to the binary value;
- **Note**: the lack of symmetry is due to representing zero (0) as one of the 2^{N-1} values:
  - {2^{N-1} negative, 0, 2^{N-1}-1 positive} values
  - Remember: negative values stored as the 2's compliment of the number.
- Attempting to store a number larger/smaller than the maximum/minimum value then Integer Overflow occurs

Real Numbers

- Positive or negative value that consists of a whole number plus a fractional part (expressed in floating point, or scientific notation);
- In Java: `float` and `double` are used;
- Real numbers' range and accuracy are limited in computing systems:
  - How would $\frac{1}{3}$ or $\sqrt{2}$ be stored in a binary format?

Range and Accuracy of Real Numbers

- Determined by number of bits and the split up of the **mantissa** and **exponent**
- There has to be a limit on the range, by definition, you need an infinite number of bits to represent infinity (∞)
- Accuracy is limited
  - The number of significant digits is limited
    - There are an infinite number of real values between any two points on the number line
  - Irrational numbers
  - Recurring decimals
  - IEEE 754 form (binary conversion)

Real and Integer Expressions

- Real operands used with + - * / produce Real results

Expression	Result
27.3 + 8.4	35.7
7.0 - 10.0	-3.0
3.0 * 5.0	15.0
11.0 / 4.0	2.75

- Integer operands used with + - * / % produce Integer results

Expression	Result
27 + 8	35
7 - 10	-3
3 * 5	15
11 / 4	2
11 % 4	3
10 % 2	0

Integer Arithmetic

- The integer truncation feature of / (DIV) and the remainder operator % (MOD) are very useful and powerful tools
  - Think of long division:

$$\begin{array}{rr} \texttt{DIV} & 2 \\ & 4\overline{)11} \\ \texttt{MOD} & 3 \end{array}$$

  - Assume year holds 4 digit year e.g., 1998
    - `(year / 100)+ 1  //Evaluates to Century`
  - Other examples:
    -
      ```
      numPages = (numLines / linesPerPage)+ 1   //Number of pages
      ```
    - `hours   = (hhmm / 100)   //Hours from 24hr time`
    - `minutes = (hhmm % 100)   //Minutes from 24hr time`

Some Java Maths Operators

- Below are some maths operators in Java.

Operator	Description
+	Addition - Adds values on either side of the operator
-	Subtraction - Subtracts right hand operand from left hand operand
*	Multiplication - Multiplies values on either side of the operator
/	Division - Divides left hand operand by right hand operand
%	Modulus - Divides left hand operand by right hand operand, returns remainder

Operator Precedence Example

Expression	Result
`7 + 23 * 6`	`= 7 + 138 =` **145**
`3 * 2 + 4 * 5`	`= 6 + 20 =` **26**
`-6 * 2`	`=` **-12**
`3 + 5 * 6 / 4 + 2`	`= 3 + 30 / 4 + 2 = 3 + 7 + 2 =` **12**
`3.0 + 5.0 * 6.0 / 4.0 + 2.0`	`= 3.0 + 30.0 / 4.0 + 2.0 = 3.0 + 7.5 + 2.0 =` **12.5**
`-6 * 2 + 3 / 4`	`= -12 + 0 =` **-12**
`2 * 5 % 2`	`= 10 % 2 =` **0**

Assignment Operators

- Short hand way of modifying the contents of a variable

Traditional	Alternative
`x = x + 5;`	`x += 5;`
`x = x - 32;`	`x -= 32;`
`fred = fred * 2;`	`fred *= 2;`
`ralf = ralf / 6;`	`ralf /= 6;`

- Must be careful though:
  - `y *= x - 2;`
- Is the same as:
  - `y = y * (x - 2);`
- But **not** the same as:
  - `y = y * x - 2;` or `y = (y * x) - 2;`

The Increment/Decrement Operator

- Increment (++) / Decrement (--)
  - `x++;` is the same as `x = x + 1;`
  - `x--;` is the same as `x = x - 1;`
- Be careful though:
  - `x = x++;` is *nonsense*
- Also, `++x;` and `--x;`
  - These work differently in expressions
  - In this unit, do not use them in an expression

Mixed Mode Arithmetic

- Mixed mode arithmetic occurs when a numeric expression contains a mixture of integer and reals

```
y = 3 + 4.5;
z = 2 / 3.0;
```

- Programming languages always have a set of rules for evaluating mixed mode expressions, but:
  - It's not the same across different languages
  - Not always supported by the compiler
- Errors caused by mixed mode arithmetic in program code are *extremely* hard to find
- The rule: **never** use mixed mode arithmetic

Type Casting

- To convert from one data type to another, a Type Cast is used
- The syntax is: `(NewDataType)(expression);`
- Examples:

```
int a, b, c;
double x, y, z, average;

... // Initialise variables

average = (double)(a + b + c + d) / 4.0;
    // a + b + c + d are added first, the value is converted to a double
    // then divided by 4.0, then assigned to average

z = (double)(a + b);
    // a and b are added, the value is converted to a double
    // then assigned to z

a = (int)y;
    // the value of y is truncated to an int, then assigned to a
    // (y is NOT changed)
```

Type Casting (2)

- Examples cont:

```
int a, b, c;
double x, y, z, average;

... // Initialise variables

x = (double)(a / b);
    // this is a div b, then converted to double and assigned to x
    // if a is 5 and b is 2, x is assigned 2.0
    // x = (double)(5/2);

y = (double)a / (double)b;
    // this is convert both the values of a and b to doubles
    // then normal division, same as y = 5.0/2.0;
```

- Note: conversion from a real data type to an integer data type involves truncating the real value (i.e., not rounding)

Expression Guidelines

- Never use mixed mode arithmetic
  - Use type casting to prevent mixed mode arithmetic
- Precedence rules are the same as in mathematics
- Use parentheses to simplify readability of complex expressions
- Use intermediate steps to split complex expressions into explicitly seperate steps
- Don't over-parenthesise simple expressions
- Beware or algebraic simplicity:

$$x = \frac{y - p}{z - q}$$

- This is written in Java as `x = (y - p) / (z - q)`

Character Data Types

- A `char` stores a single Character e.g., 'a', 'A', '6', '&', etc.
- Stored in a Unicode, a standard that arbitrarily designates a bit pattern to represent a particular character symbol
- If the character is a decimal digit e.g., '8' can't do arithmetic with it: `'8'+'6';` cant possibly be expected to be meaningful
- A character occupies 16 bits & is coded according to the Unicode standard, thus there are >32,000 different possible combinations to represent characters, more than enough
  - The lower (rightmost) 8 bits is identical to the ASCII system
- Order of the characters is determined by the codes:
  - 'A' < 'B' ... < 'Z' < ... < 'a' < 'b' < ... < 'z'

Java's Primitive Data Types

- Java defines 8 primitive types:

Java Type	Memory Format	Range/Domain	Range/Domain
byte	8 bit integer	-2^7 to 2^7-1	-128 to 127
short	16 bit integer	-2^{15} to $2^{15}-1$	-32768 to 32767
int	32 bit integer	-2^{31} to $2^{31}-1$	-2147483648 to 2147483647
long	64 bit integer	-2^{63} to $2^{63}-1$	$\pm$9.22337E+18
float	32 bit floating point	± 6 sig. digits (10^{-46},10^{38})	
double	64 bit floating point	± 15 sig. digits (10^{-324},10^{308})	
char	16 bit character	All Characters	
boolean	boolean	true, false	

Live Demo

- In this live demo we will look at:
  - Using primitive data types.

String Type

The Java String Class

- A string is a collection of 0 (empty) or more characters;
- The Java String class provides the facility to handle strings;
- String variables are objects but can be used like primitives:
  - `String unitName ="Programming";`
- Can also be treated like an object:
  - `String unitName =new String("Programming");`

Initialising Variables

- What is stored in a variable when it is created?
- Java auto-initialises variables:
  - Primitive Variables:
    - `Numeric:` `set to zero`
    - `char:` `set to blank`
    - `boolean:` `set to false`
  - Object variables:
    - `set to null`
    - `null` represents an invalid memory address
- Not all programming languages auto-initialise so it is extremely poor programming style to rely on auto-initialisation
- **Always** explicitly initialise your variables

Live Demo

- In this live demo we will look at:
  - Strings; and
  - Using Strings in applications.

User Input

User Input - Scanner

- Receiving and processing user input is fundamental;
- In Java, the **Scanner** `class` is used;
- Create a **Scanner** object and then use any of its **methods**;
- Close the **Scanner** object when finished with it.
- Found in the **java.util** package;

```
import java.util.*;
public class MyFirstProgram
{
    public static void main(String[] args)
    {
        Scanner input = new Scanner(System.in);
        ...
        input.close();
    }
}
```

Scanner Methods

- To process user input, `Scanner methods` are implemented;
- Some `Scanner methods` using the previous slide's code:
  - `input.nextLine()` read a line of text entered by the user;
  - `input.nextInt()` reads an int value entered by the user;
  - `input.nextDouble()` reads a double value entered by the user;
  - `input.nextFloat()` reads a float value entered by the user.
- Reading a single character is slightly different:
  - `input.next().charAt(0)` reads the first character from the user.
- Close `Scanner` object when finished using it:
  - `input.close()` closes the `Scanner` object.

Example Code

```
import java.util.*;

public class MyFirstProgram
{
    public static void main(String[] args)
    {
        String name = "";

        Scanner input = new Scanner(System.in);
        System.out.print("What is your name? ");
        name = input.nextLine()
        System.out.println("Hello " + name + "!!");
        input.close();
    }
}
```

Live Demo

- In this live demo we will look at:
  - Declaring a Scanner object;
  - Using the Scanner to read a String;
  - Using the Scanner to read an integer;
  - Using the Scanner to read a double;
  - Using the Scanner to read a float; and
  - Using the Scanner to read a character.