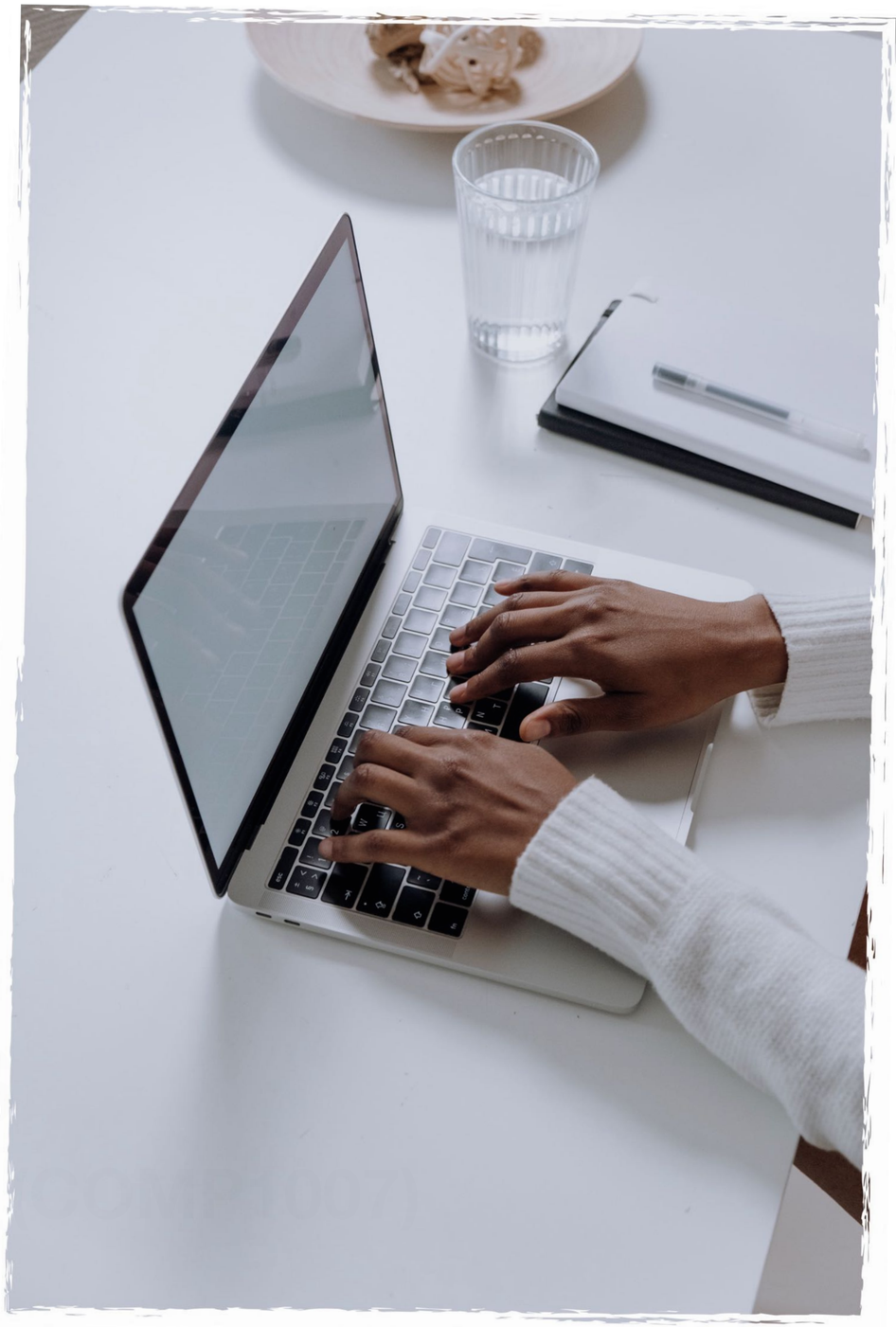
Dr David A. McMeekin

Programming Basics

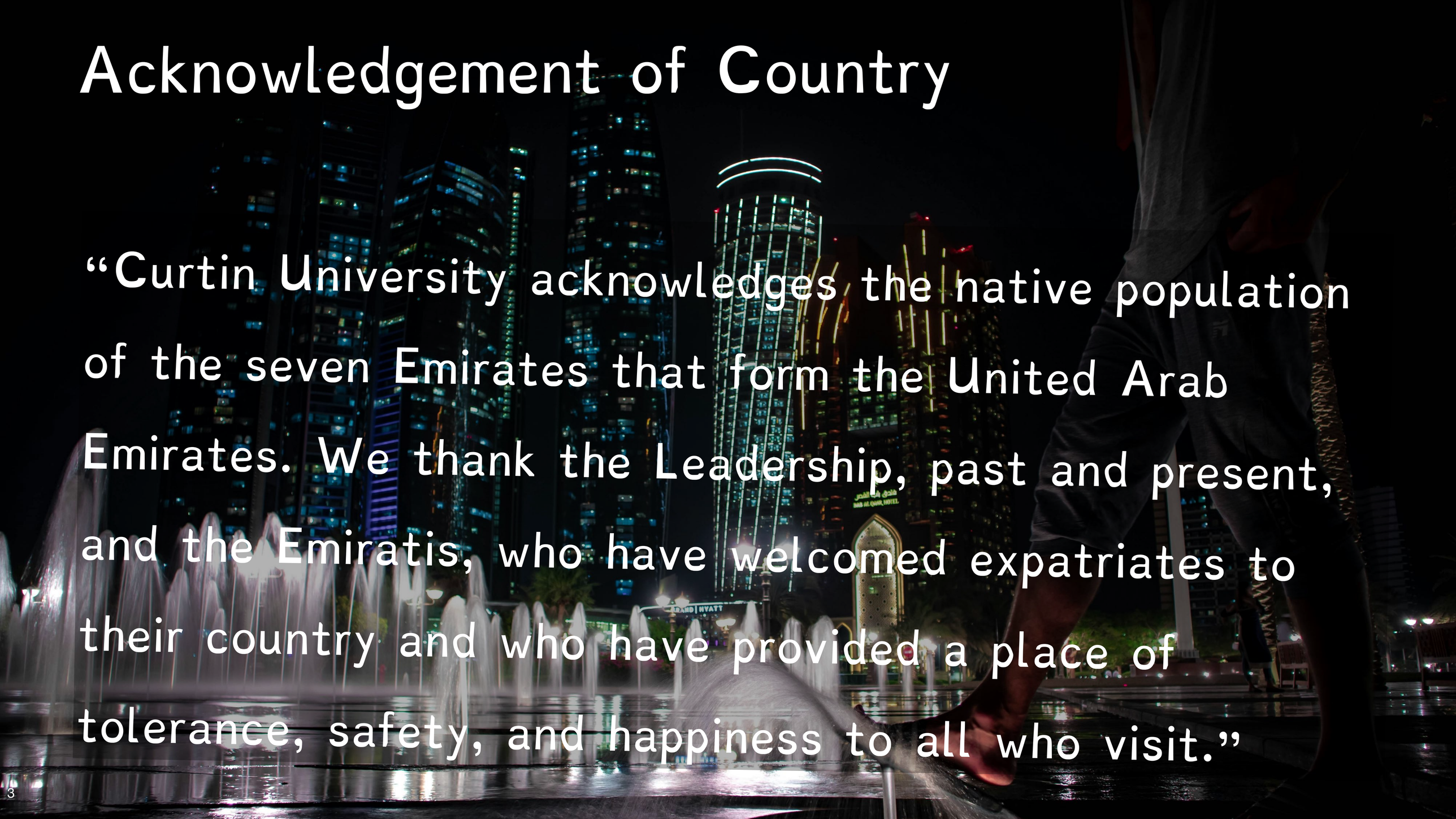
Programming Design and Implementation



Acknowledgement of Country

“I acknowledge the Wadjuk people of the Noongar nation on which Curtin University’s Boorloo Campus sits, and the Wangkatja people where the Karlkurla Campus sits, as the traditional custodians of the lands and waterways, and pay my respect to elders past, present and emerging.”

Acknowledgement of Country



“Curtin University acknowledges the native population of the seven Emirates that form the United Arab Emirates. We thank the Leadership, past and present, and the Emiratis, who have welcomed expatriates to their country and who have provided a place of tolerance, safety, and happiness to all who visit.”

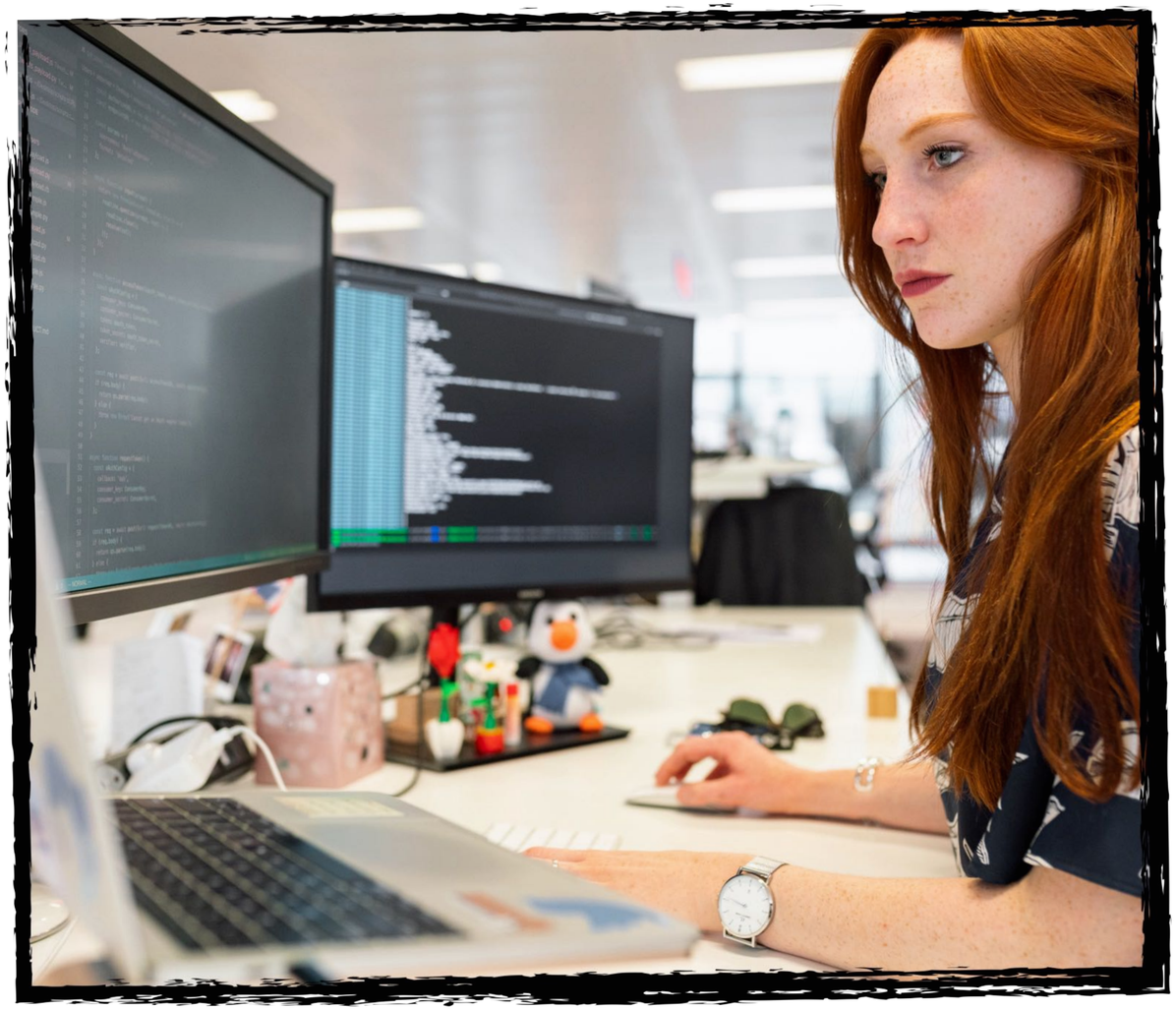
COMP1007 - Unit Learning Outcomes

- 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 aLinux 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.

Outline

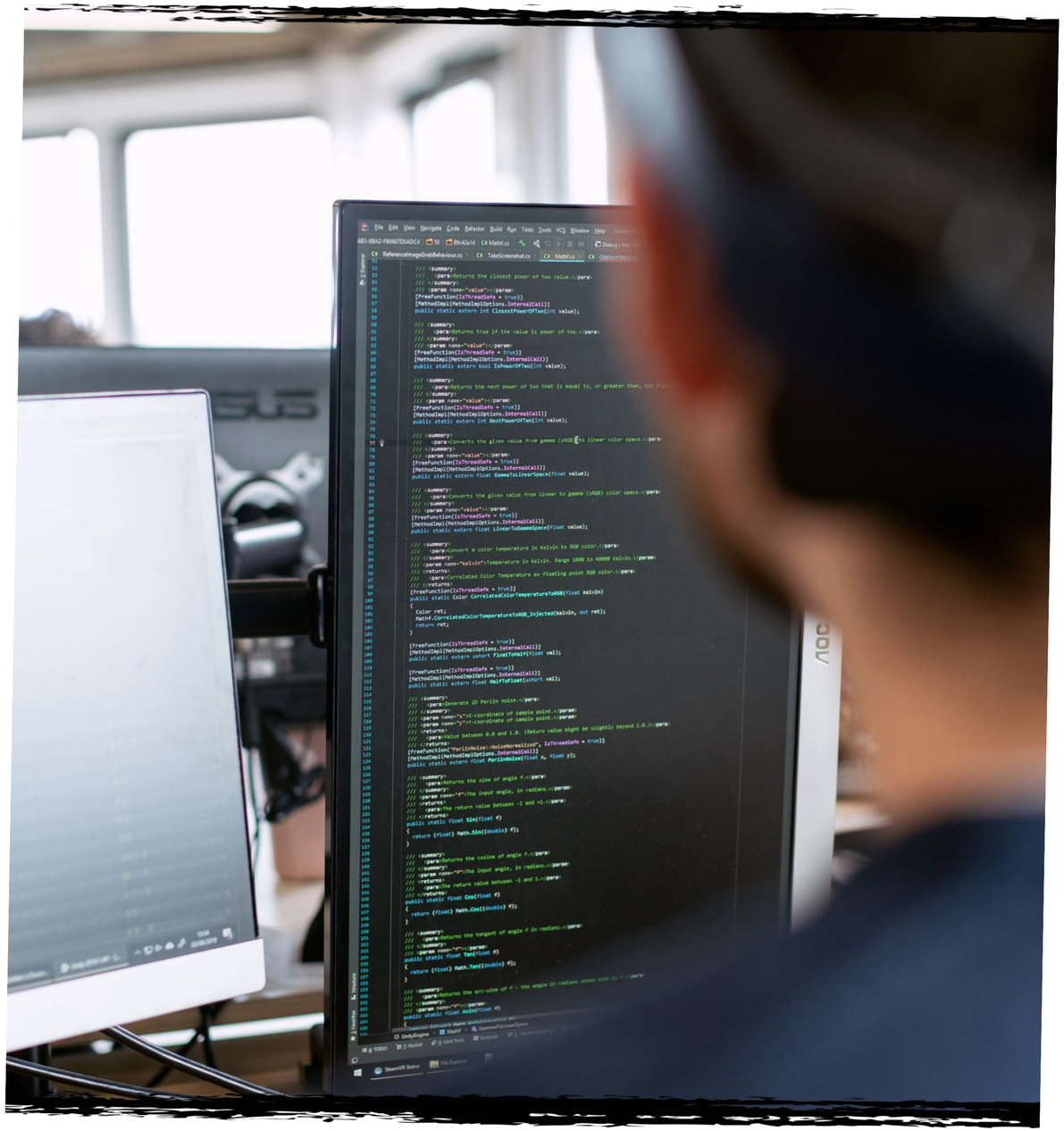
- Development
- Java
- Writing Java
- Variables
- Primitive Types
- String
- User Input

Development



Software Development

- Understand the Problem:
  - Define the problem;
  - Think/consult/revise.
- Design a solution:
  - Specify steps involved;
  - Create algorithms.
- Test the solution:
  - Does the design work (in theory).
- Fix the solution:
  - Correct the design where it failed.
- Write and document the code:
  - Test the code works;
  - Ensure 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.

Design a Solution – Algorithm

- Algorithm: set of detailed, unambiguous, ordered steps specifying a solution to a problem;
- State steps precisely, without ambiguity (pseudocode, next slide);
- English description independent of any programming language;
- Non trivial problem will need several refinement stages; and
- Various methodologies available.

Algorithm – Pseudocode

- Algorithms are expressed in pseudocode:
  - English like phrases describing the algorithm steps;
- Pseudocode is **NOT** a programming language;
- It is **PSEUDO**;
- Pseudocode 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

- Clear;
- Logical;
- Understandable;
- Consistent; and
- Correct.

Pseudocode - 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 the 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.
- That is: code the solution;
- 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 (OO):
  - 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.

Introduction to Java

- Java is an Object Oriented (OO) language;
- James Gosling designed it 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

- The source code file is translated line by line into machine code instructions that the machine executes:
  - If syntax errors exist, program partially executes;
  - As soon as syntax errors are encountered, execution halts;
- Python and Ruby are interpreted languages;
- Interpreting is slower than compiled code, as syntax checking & translation was already done when compiling the code.

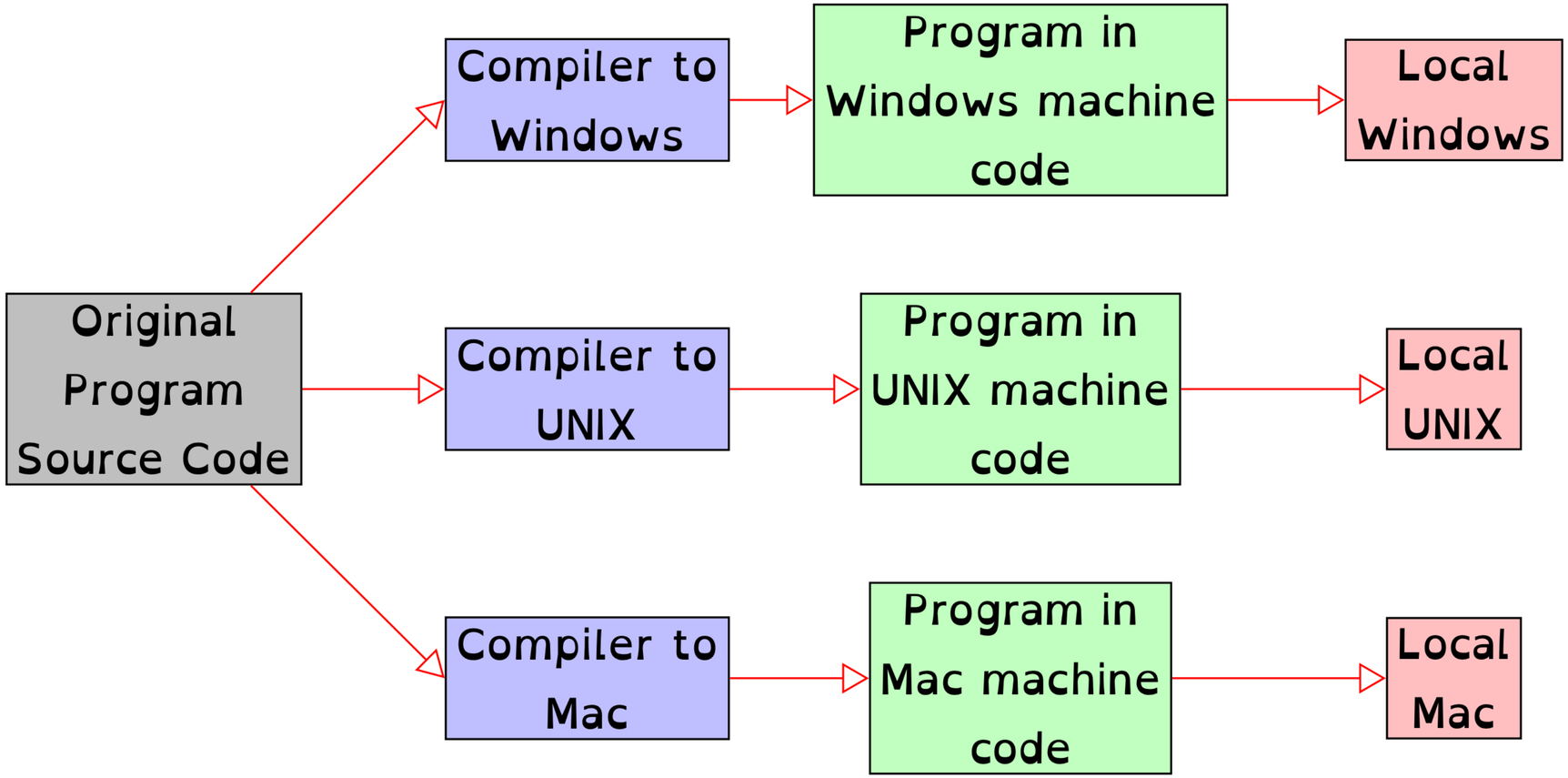
Compiling

- Compilation: the process whereby the source code file is translated into machine code;
- Source code is checked ensuring it conforms with grammar (syntax and semantics) rules;
- If no syntax errors found, then machine code file is created ready for execution;
- With just one error the machine code file is not created.

Java is Platform Independent

- Achieved by running byte code on Java Virtual Machine (JVM);
- Byte code is JVM machine code;
- Java compiler compiles from source code to byte code;
- JVM is a program whose job it is to interpret Java byte code;
- Each machine needs native machine language code: thus byte code is interpreted to local machine code to be executed locally;
- Byte code interpretation overheads is lower than traditional interpreters, conversion is from machine code (JVM) to machine code (native machine code).

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:
  - Source code is secure
  - Only one version of the byte code needs to exist

Java – the 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.

Java – the 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 the extension of **.java** must be created;
- This file stores the human readable Java program;
- The Java compiler (known as **javac**) is used to compile the source code into byte code;

Creating a Java Program

- For each class (see later) defined in source code the compiler creates a corresponding byte code file;
- Each byte code file has the same name as the **.java** file but is a **.class** file;
- The Java interpreter (known as **java**) then translates the byte code and executes the resulting native machine code.

Pseudocode Example

```
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);
      }
 }
```

Pseudocode to Java Code

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

```
import java.util.*;

public class MyFirstProgram
 {
     public static void main(String[] args)
     {
         String message = "Hello World!”;
         System.out.println(message);
     }
 }
```

Creating the Java File

- Enter the Java code into a text file (COMP1007, use vim or gVim);
- The **.java** file name **MUST** be exactly the same as the class name.

```
import java.util.*;

public class MyFirstProgram
 {
      public static void main(String[] args)
      {
           String message = "Hello World!”;
           System.out.println(message);
      }
 }
```

Compiling and Running the Java Code

- The .java file is then compiled into bytecode using the command:

javac MyFirstProgram.java

- If program contains errors, they're displayed, no bytecode created;
- If no errors, byte code is created;
- Bytecode in file called: **MyFirstProgram.class**

- Execute the program using the command:

java MyFirstProgram

The Import Statement

- Java comes with an extensive class library;
- The libraries make implementing algorithms much easier;
- Many organisations develop their own class libraries;
- Understanding the **import** statement is important;
- It 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

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 MAX = 99;
```

- A literal value is a literal value, as shown below:
  - Integer: 12, -15, 22
  - Real: 22.0987, -3.4561, 999.09854

Assigning Values to Variables

- The **assignment operator** '=' is used:

```
int thisIsAnInteger = 12;
```

- What's 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

```
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 declaring the variable;

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 – More Difficult Decisions

- What about more complex variables:

```
dateOfBirth    //6 or 8 digit Integer or 3 seperate Integers?
age            //Integer or Real?
phoneNo        //Integer or a String?
```

Data Types

Integer Data Types

- Integer: positive or negative value, whole number;
- Java primitive types:
  - **byte**, **short**, **int**, **long**, all integer abstractions;
- Integer range: determined by the amount of memory available for the 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 allocated number of bits;
- Every additional bit doubles the range size;
- For N bits: 1 bit for the sign, remaining N−1 bits represent 2^{N-1} different combinations directly related to the binary value;
- Lack of symmetry 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 using number's 2's compliment.
- Attempting to store a number larger/smaller than the maximum/minimum value results in Integer Overflow.

Real Numbers

- Positive or negative value consisting of a whole number plus a fractional part (expressed in floating point, or scientific notation);
- Java uses: **float** and **double;**
- Real numbers' range and accuracy are limited in computing systems:
  - How would you store or ? in a binary format?

Range and Accuracy of Real Numbers

- Determined by number of bits and the split up of the mantissa and exponent;
- The range is limited, by definition, an infinite number of bits needed to represent infinity (∞);
- Accuracy is limited:
  - The number of significant digits is limited:
    - An infinite number of real values exist between two points on the number line;
  - Irrational numbers;
  - Recurring decimals; and
  - IEEE 754 form (binary conversion).

Real and Integer Expressions

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

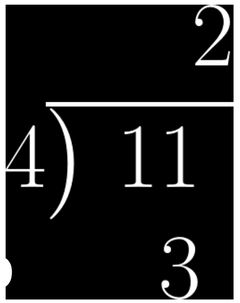
Expression	Result
27.3 + 8.4	35.7
7.0 - 10.0	-3.0
3.0 * 5.0	15.0
11.0 / 4.0	2.75

- 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

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



- Examples:
  - Assume year holds 4 digit year e.g., 1998:

```
(year / 100) + 1 //Evaluates to Century
numPages = (numLines / linesPerPage) + 1
hours = (hmm / 100) //hours from 24hr time
minutes = (hmm % 100) //minutes from 24hr time
```

Some Java Maths Operators

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 Examples

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 to modify the a variable's content;

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 the 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;`

- Also, `++x;` and `--x;`
  - These work differently in expressions;
  - Not used in COMP1007

Mixed Mode Arithmetic

- Occurs when mixing integer and reals in an expression:

```
y = 3 + 4.5;

z = 2 / 3.0;
```

- Programming languages always rules for evaluating mixed mode expressions, but:
  - It's different across different languages;
  - Compiler does not always it.

Mixed Mode Arithmetic (2)

- Mixed mode arithmetic errors are extremely hard to find;
- The rule: **NEVER** use mixed mode arithmetic.

Type Casting and Examples

- Is used to convert from one data type to another;
- The syntax is: `(NewDataType)(expression);`

```
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)
```

More Type Casting Examples

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

```
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;
```

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 complex expression readability;
- Use intermediate steps to break up complex expressions; and
- Don't over-parenthesise simple expressions.

Algebraic Simplicity

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

- In Java is written:

```
x = (y - p)/(z - q);
```

Character Data Type: char

- **char** stores a single character: `'a'`, `'A'`, `'6'`, `'&'`
- Stored in a Unicode, a standard that arbitrarily designates a bit pattern to represent a particular character symbol;
- Even if **char** represents a digit **'8'**, **'6'**, cannot do arithmetic:

```
'8' + '6';
```

More on `char`

- A `char` occupies `16` bits;
- Coded using the Unicode standard, hence `>32,000` different combinations to represent characters (enough);
- The lower (rightmost) `8` bits 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 $2^{63} - 1$	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	

Initialising Primitives

- What is stored in a variable when it is created?
- Java auto-initialises primitive variables:
  - Numeric: set to zero;
  - char: set to blank; and
  - boolean: set to false.

Java String Class (not a primitive, an Object)

- 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");
```

- Objects initialised to: null (represents an invalid memory address)

User Input

User Input - Scanner

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

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

Scanner Methods

- To process user input, `Scanner methods` are implemented:
  `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.

Example Code

```
import java.util.*;
public class MySecondProgram
 {
    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 + "!!");
    }
 }
```

References

Photo by ThisIsEngineering from Pexels
Photo by ThisIsEngineering from Pexels
Photo by Sofia Alejandra from Pexels
Photo by Markus Spiske on Unsplash
Photo by Rae Tian on Unsplash
Photo by Salman Hossain Saif on Unsplash
Photo by Karl Pawlowicz on Unsplash