



# Programming and Software Development

## COMP90041

### Lecture 2

# Console IO

NOTE: Some of the Material in these slides are adopted from

- \* Lectures Notes prepared by Dr. Peter Schachte and
- \* the Textbook resources



- Object Oriented Programming (Class vs Object)
- “Hello World” Java program
- Javac
- Java
- Variables
  - Primitive data types
  - Declaration and Assignment

# Review

- Operations for primitive data types & type conversions
- String class and operations for String
- Formatted console output
- Handling command line inputs/arguments
- Reading console input using Scanner class



- Operations for primitive data types & type conversions
- String class and operations for String
- Formatted console output
- Handling command line inputs/arguments
- Reading console input using Scanner class

## Outline



Type	Bytes	Values
boolean	1	false, true
char	2	All unicode characters (e.g., 'a')
byte	1	$-2^7$ to $2^7 - 1$ (-128 to 127)
short	2	$-2^{15}$ to $2^{15} - 1$ (-32768 to 32767)
int	4	$-2^{31}$ to $2^{31} - 1$ ( $\approx \pm 2 \times 10^9$ )
long	8	$-2^{63}$ to $2^{63} - 1$ ( $\approx \pm 10^{19}$ )
float	4	$\approx \pm 3 \times 10^{38}$ (limited precision)
double	8	$\approx \pm 10^{308}$ (limited precision)

## Primitive Data Types



- Each type has certain operations that apply to it
- Common operations for primitive number types:  $+$   $-$   $*$   $/$  (division)  $\%$  (modulus / remainder)
  - Type of result is same as type of operands
- Use operations to construct expressions, which have values that can be assigned or used as operands
  - E.g., `answer = (2 + 4) * 7;`
  - `count = count + 1;`

## Operations for Number Types



- Comparison operations also work for number type
  - $<$  : less than
  - $\leq$  : less than or equal to
  - $>$  : greater than
  - $\geq$  : greater than or equal to
  - $==$  : equal to
  - $!=$  : not equal to
- Comparisons return boolean values
  - E.g.,  $5 \neq 4$  returns true

6 < 7  
is 7 == 4 ?  
==

## Operations for Number Types (cont)



- **&& (AND)** is true if both operands are true
  - E.g., `int x = 5;` then `(x != 4) && (x > 3)` is true
- **|| (OR)** is true if either operand is true
  - E.g., `int x = 5;` then `(x != 4) || (x < 3)` is true
- Both are short-circuit operations: they only evaluate the second operand if necessary
  - E.g., `int x = 5;` then `(x != 4) || expr` is true no matter what `expr` is because `x != 4` is true
- **! (NOT)** is true if its operand is false
  - E.g., `int x = 5;` `!(x == 4)` is true

Handwritten truth table for AND and OR operations:

A	B	A & B	A    B
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	1

## Operations for booleans





- **++x** is a special expression that increments  $x$  (for any variable  $x$ ) and returns the incremented value
  - E.g., if  $x$  is 7,  $++x$  is 8, and after that,  $x$  is 8
  - Called “pre-increment” because it increments variables before returning
- **--x** (pre-decrement) is similar: it decrements  $x$  and returns it
- **x++** (post-increment) returns  $x$  and then increments it
  - E.g., if  $x$  is 7,  $x++$  is 7, and after that,  $x$  is 8
- **x--** (post-decrement) returns  $x$  and then decrements it

## Pre/Post Increment/Decrement



## Pre/Post Increment/Decrement

What will this code print?

```
int x = 10; int y = 5;  
System.out.println(x++ - ++y);
```

10 - 6

A. 3

B. 4

C. 5

D. 6

E. 7

# Quick Poll



## Pre/Post Increment/Decrement

What will this code print?

```
int x = 10; int y = 5;  
System.out.println(x++ - ++y);
```

A. 3

☒ B. 4

C. 5

D. 6

E. 7

# Quick Poll



- Pre/post increment/decrement can be confusing (like the last example)
- They can also be used as statements rather than expressions
  - E.g., `++x;` or `x++;`
  - Used as statements, these both just increment `x`

## Pre/Post Increment/Decrement Use



- Primitive operations work on operands of the same type
- But Java can convert types automatically
- A widening conversion converts a number to a wider type (so the value can always be converted successfully)
- Automatic conversions in Java

byte → short → int → long → float → double  
                  ↑  
                char

## Type Conversions

- Narrowing conversions are also possible. But they must be performed explicitly using a **cast**
- Cast is specified by writing the name of the type to convert to in parentheses before the value to be converted
  - E.g., `short x; int y = 50; x = (short) y;`
- Cast can also be used to explicitly ask for a widening conversion

```
int sum;  
int count;  
//compute sum and count ...  
double average = (double) sum / count;
```

## Casting



- Precedence of 2 operators, say  $\odot$  and  $\oplus$  determines whether  $a \odot b \oplus c$  is read as:
  - ▶  $(a \odot b) \oplus c$  ( $\odot$  has higher precedence), or
  - ▶  $a \odot (b \oplus c)$  ( $\odot$  has lower precedence)
  - ▶ E.g.,  $2+3*4$  ( $*$  has higher precedence)
- Associativity determines whether  $a \odot b \odot c$  is read as:
  - ▶  $(a \odot b) \odot c$  (left associativity), or
  - ▶  $a \odot (b \odot c)$  (right associativity)
  - ▶ E.g.,  $3-2-1$  ( $-$  associates left)
- Java's rules are mostly as you would expect
- When in doubt, just put in parentheses

## Precedence and Associativity

Symbol	Associativity
. (method invocation)	
++ --	
- (unary negation)	
( <i>type</i> ) casts	
* / %	Left
+ -	Left
< > <= >=	Left
== !=	Left
&&	Left
	Left
= += *= ...	Right

# Operators, High to Low Precedence





- After running the following piece of code, what will be the value of x, y, and z?

```
int x = 10, y = 5;  
int z;  
z = --x - y * 5 + x * (y++ - 4);
```

- A.  $x = 10, y = 5, z = 5$
- B.  $x = 9, y = 5, z = -7$
- C.  $x = 9, y = 5, z = 5$
- D.  $x = 9, y = 6, z = -7$

## Quick Poll

- After running the following piece of code, what will be the value of x, y, and z?

```
int x = 10, y = 5;  
int z;  
z = --x - y * 5 + x * (y++ - 4);
```

- A.  $x = 10, y = 5, z = 5$
- B.  $x = 9, y = 5, z = -7$
- C.  $x = 9, y = 5, z = 5$
- D.  $x = 9, y = 6, z = -7$

## Quick Poll



- Operations for primitive data types & type conversions
- **String class and operations for String**
- Formatted console output
- Handling command line inputs/arguments
- Reading console input using Scanner class

## Outline



- **String** is a class type, not a *primitive* type, so strings are *objects*
- Specify a string constant by enclosing in double-quotes ("")
  - E.g., `String s = "Hello, World!";`
- Use backslash (\) to include double-quote and other special characters (e.g., % and \) in a string
  - E.g., `println("He said \"backslash (\\) is special!\"")`
  - Prints "He said "backslash (\) is special!"
- Certain letters after backslash are treated specially
  - Eg., `\n` - new line, `\t` - tab character

## Strings



- You can use `+` to append two strings
  - E.g., `System.out.println("Hello " + "World");`
  - Prints "Hello World"
- If either operand is string, `+` operation will turn the other into a string
  - E.g., `System.out.println("a = " + a);`
  - If `a` is 1, this prints "a = 1"
- Beware:
  - `System.out.println("a + a = " + a + a);`
  - Actually prints "a + a = 11", not "1 + 1 = 2"  
corrected

## String Operations



- String class has many more operations, e.g.:
- Assume String s, s2; int i, j;
  - s.length() returns length of the string
  - s.toUpperCase() returns ALL UPPER CASE version of the string
  - s.substring(i, j) returns the substring of s from character i through j-1, counting the first char at 0
  - s.equals(s2) returns true if s and s2 are identical
  - s.indexOf(s2) returns the first position of s2 in s
- Don't use ==, <, >= etc. to compare strings
- See String class in documentation for more operations
  - Java API 8: <https://docs.oracle.com/javase/8/docs/api/>

## More String Operations



- Operations for primitive data types & type conversions
- String class and operations for String
- **Formatted console output**
- Handling command line inputs/arguments
- Reading console input using Scanner class

## Outline



- **printf** is like **print**, but it lets you control how data is formatted
- Method form:
  - System.out.printf(format-string, args ...);
- Example:
  - System.out.printf("Average: %5.2f", average);
- *Format-string* is an ordinary string, but can contain format specifiers, one for each of the arguments (args)
  - A format specifier begins with %
  - It may have a number specifying how to format the next value in the args list
  - It ends with a letter specifying the type of the value
- Ordinary text in format-string is printed as is

## Formatted Output





%X.Y

- The (optional) number(s) following % is/are interpreted:
  - **X** (before decimal point) specifies the minimum number of characters to be printed
    - The **full** number will be printed, even if it takes more characters than X
    - If X is omitted, the value will be printed in its minimum width
    - If X is negative, the value will be left-justified, otherwise right-justified
  - **Y** (after decimal point) specifies the number of digits of the value to print after the decimal point
    - If Y is omitted, Java decides how to format

## Format specifiers

The final letter in a format specifier can be:

<u>d</u>	format an <u>integer</u> (no fractional part)
s	format a string (no fractional part)
c	format a character (no fractional part)
<u>f</u>	format a float or double
e	format a float or double in exponential notation
g	like either %f or %e, Java chooses
%	output a percent sign (no argument)
n	end the line (no argument)

- Good format for money: \$%.2f

## Format Letters

```
public class PrintExample
{
    public static void main(String [] args)
    {
        String s = "string";
        double pi = 3.1415926535;
        System.out.printf("\"%s\" has %d characters %n", s, s.length());
        System.out.printf("pi to 4 places: %.4f%n", pi);
        System.out.printf("Right>>%9.4f<<", pi);
        System.out.printf(" Left >>%-9.4f<<%n", pi);
    }
}
```

### Generated Output

```
"string" has 6 characters
Pi to 4 places: 3.1416
Right>>    3.1416<< Left>>3.1416    <<
```

## Formatted Output Example



How many characters appear before the decimal point in a number `x` printed with `printf("%6.2f", x)`?

- ☐ A I don't know
- ☐ B 2
- ☐ C 3
- ☐ D 4
- ☐ E 6

**Quick Poll**



How many characters appear before the decimal point in a number ~~x~~ printed with `printf("%6.2f", x)`?

☒ A I don't know

☐ B 2

☐ C 3

☐ D 4

☐ E 6

Quick Poll



- Operations for primitive data types & type conversions
- String class and operations for String
- Formatted console output
- Handling command line inputs/arguments
- Reading console input using Scanner class

## Outline



- When your program is run, it can be given arguments on the command line
  - E.g., `javac Hello.java` ("Hello.java" is an argument for `javac`)
- Allows the user to give information to the program
- For the boilerplate we've been using, the command line arguments can be referred to as
  - First argument: `args[0]`
  - Second argument: `args[1]`, etc..
- Each of these arguments is a string

## Handling Command Line Inputs



```
// print out a friendly greeting
public class Hello2 {
    public static void main(String[] args) {
        System.out.println("Hello, " + args[0] + "!");
    }
}
```

## Program Use

```
frege% java Hello2 Peter
Hello, Peter!
frege% java Hello2
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0
    at Hello2.main(Hello2.java:4)
```

# Command Line Input Example





- To convert string to int:

`Integer.parseInt(string)`

```
// Print double the command line integer
public class Hello3 {
    public static void main(String[] args) {
        System.out.println("Twice your number is "
            + 2 * Integer.parseInt(args[0]));
    }
}
```

## Program Use

```
frege% java Hello3 4
Twice your number is 8
```

# Handling Command Line Inputs



- Operations for primitive data types & type conversions
- String class and operations for String
- Formatted console output
- Handling command line inputs/arguments
- Reading console input using Scanner class

## Outline



- Interactive programs get input while running
- Java 5 introduces the **Scanner** class for this
- To use **Scanner**:
  - ▶ Add this near top of source file:

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

- ▶ Create scanner: add this in **main** before reading input:

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

- ▶ Use **keyboard** as needed to read input
- ▶ *E.g.*, this reads (rest of) current line as a string:

```
String line = keyboard.nextLine();
```

## Reading Console Input

- Other methods to read from a **Scanner** variable **keyboard**:

What	Type	Expression
One word	<b>String</b>	<code>keyboard.next()</code>
One integer	<b>int</b>	<code>keyboard.nextInt()</code>
One double	<b>double</b>	<code>keyboard.nextDouble()</code>

- Similar methods to read other types; see documentation
- These all skip over whitespace and read one “word”
- Whitespace includes spaces, tabs, and newlines
- Error if text is not of expected type

## Reading Console Input

```
import java.util.Scanner;
public class ScannerExample {
    public static void main(String[] args) {
        int num1 = Integer.parseInt(args[0]);
        Scanner kbd = new Scanner(System.in);
        System.out.print("Enter second number: ");
        int num2 = kbd.nextInt();
        System.out.println(num1 + " * " + num2 +
                           " = " + num1*num2);
    }
}
```

```
frege% java ScannerExample 6
Enter second number: 7
6 * 7 = 42
```

## Command Line and Scanner Example



- `nextLine()` reads up to and including newline
- Others do not read after the next word
- After `next`, `nextInt`, or `nextDouble`, `nextLine` just reads rest of current line (maybe nothing!)
- To read a number on one line followed by the next whole line:

```
int num = keyboard.nextInt();  
keyboard.nextLine(); // throw away rest of line  
String line = keyboard.nextLine();
```

- Ideally, avoid mixing `nextLine` with the others

## Pitfall: Mixing with `nextLine`



```
Scanner kbd = new Scanner(System.in);  
int n       = kbd.nextInt();  
String s1   = kbd.nextLine();  
String s2   = kbd.nextLine();
```

Console input (on 3 lines):

```
1  
+ 2  
= 3
```

- A s1 = "+ 2", s2 = "= 3"
- B s1 = "", s2 = "+ 2"
- C s1 = "= 3", s2 = ""

**Quick Poll: What are s1 and s2 after**



```
Scanner kbd = new Scanner(System.in);  
int n      = kbd.nextInt();  
String s1   = kbd.nextLine();  
String s2   = kbd.nextLine();
```

Console input (on 3 lines):

1  
+ 2  
= 3

A s1 = "+ 2", s2 = "= 3"

B s1 = "", s2 = "+ 2"

C s1 = "= 3", s2 = ""

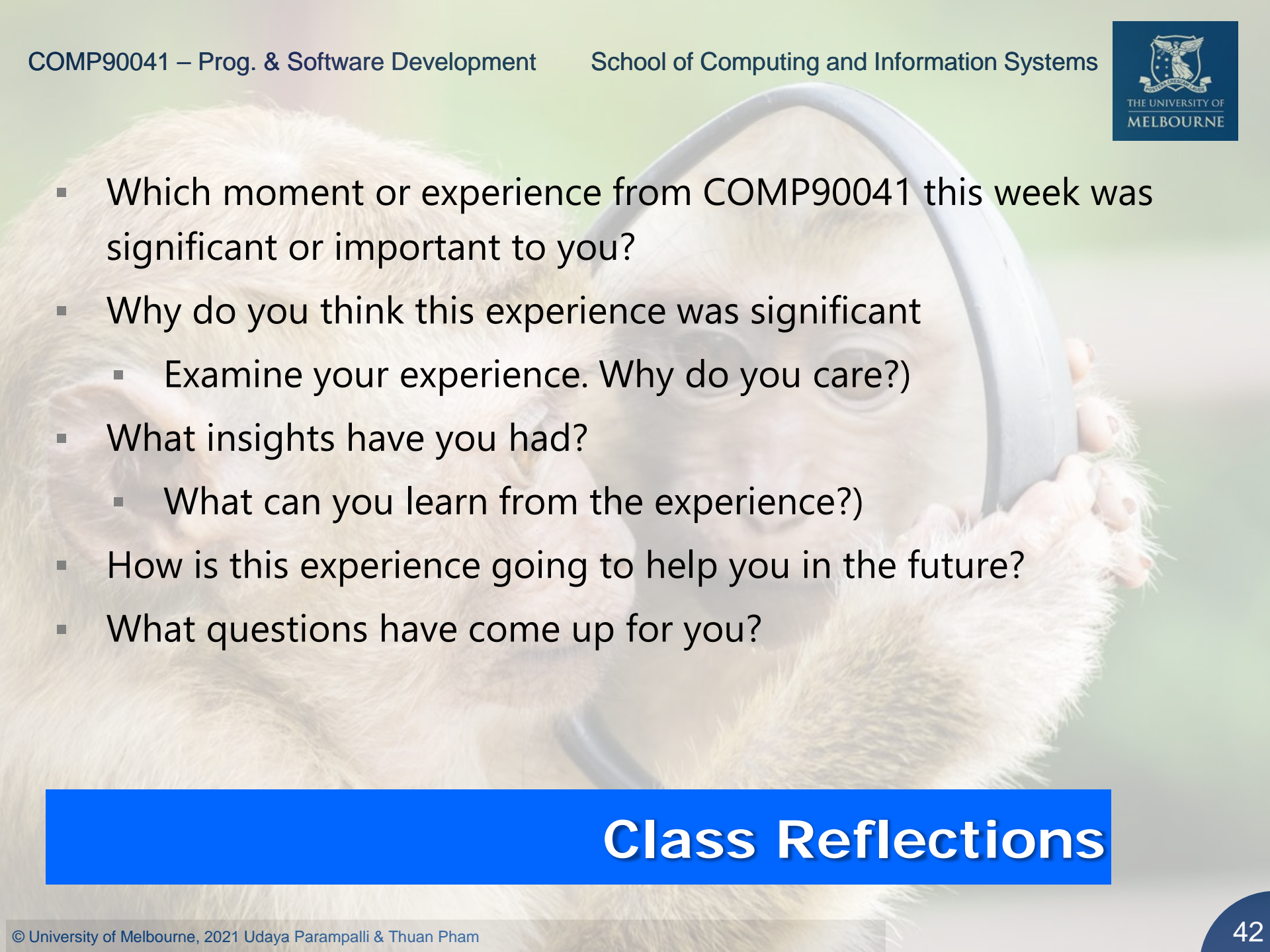
**Quick Poll: What are s1 and s2 after**



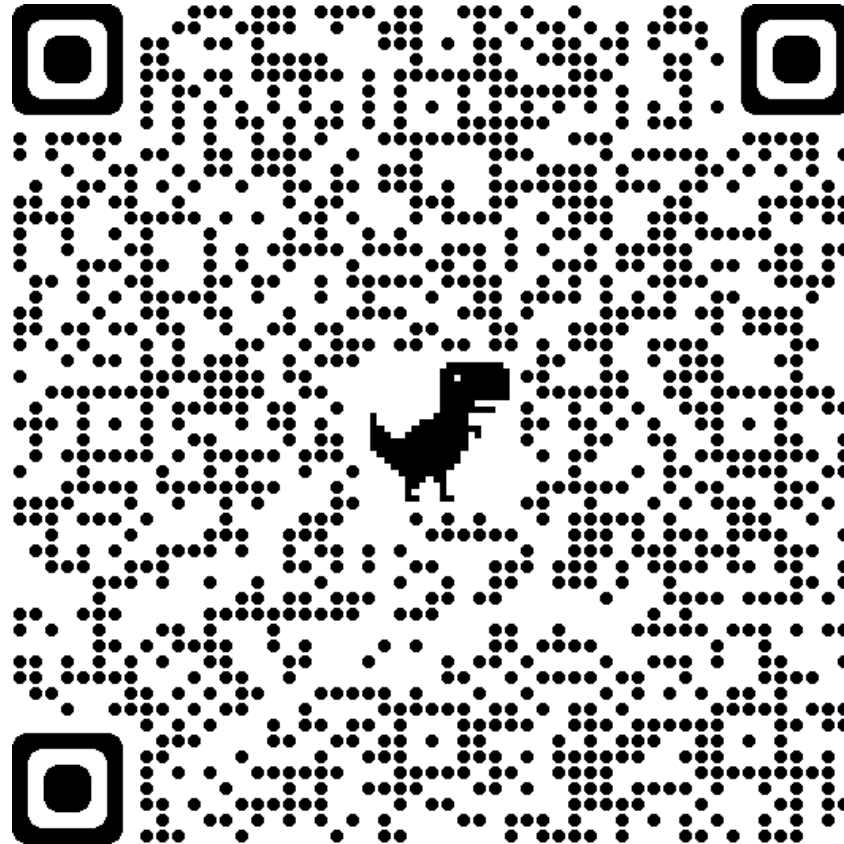


- Operations for primitive data types & type conversions
  - How to use different operators
  - How to identify/specify the precedence of the operators in an expression/a statement
  - How to convert between data types
- String class and operations for String
- Formatted console output
- Handling command line inputs/arguments
- Reading console input using Scanner class

## Summary

- 
- Which moment or experience from COMP90041 this week was significant or important to you?
  - Why do you think this experience was significant
    - Examine your experience. Why do you care?)
  - What insights have you had?
    - What can you learn from the experience?)
  - How is this experience going to help you in the future?
  - What questions have come up for you?

## Class Reflections



<http://go.unimelb.edu.au/5o8>

i.