MONASH University

Information Technology

Module 2:

Selection, Repetition, Strings and Console Input

FIT2034 Computer Programming 2
Faculty of Information Technology

Introduction to objects

Object-Oriented Programming

- This programming paradigm binds data and the code that accesses that data into self-contained objects that are analogues of real world objects or concepts
  - Because of this binding in self-contained objects, they are reusable from program to program
- A program becomes a “dance” of interacting objects
  - It interacts with other objects by presenting an interface of public methods that other objects can invoke (execute).
  - The effect of public methods is carefully controlled so that its internal data is never corrupted or assumes illegal values
- Java is designed as an object-oriented language
  - Java programs are largely concerned with the creation and interaction of Objects

Objects and Classes

- Objects are similar to Scribble's *Sprites*
- Objects are described by ***Classes***
  - A class is a type, just like int and double are types
- Unlike primitive types, class types do not use operators
  - Instead, we use ***methods***
  - ***String*** class is the only exception – it allows "+"
- Java comes with many pre-written classes for us to use
- We can make many objects of the same class type
  - Each is said to be an ***instance*** of the class

Object Reference Variable

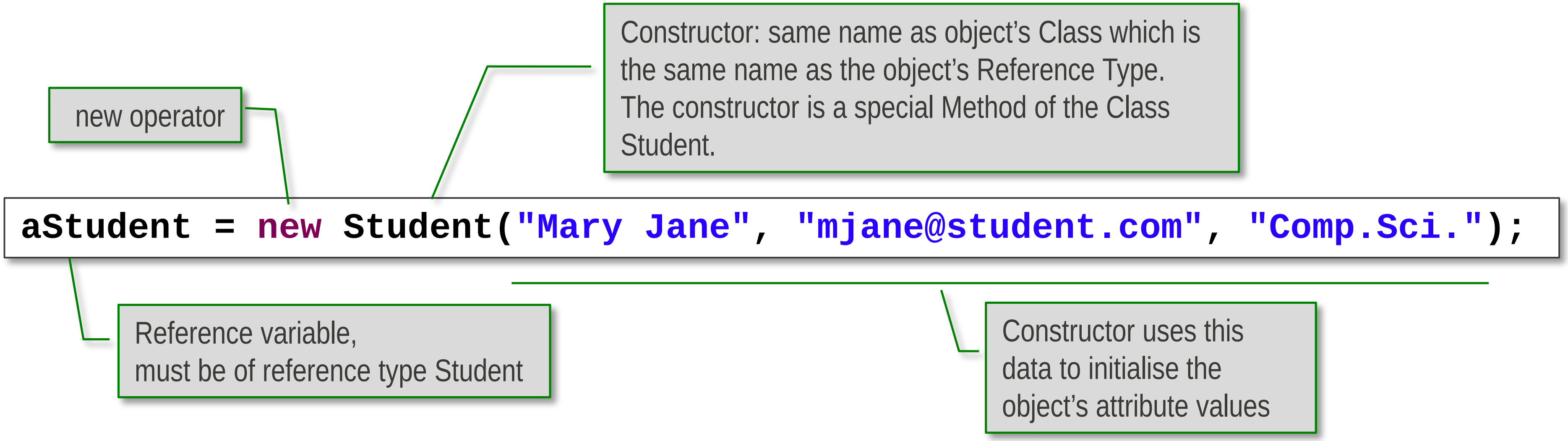
- We declare variables of class types in the same way we declare variables of primitive types
- The behaviour of the variables is different
  - Explained in a later week
- Example:
  - Assume a class type for Student. To create a variable for an object of that class, we write:

```
Student aStudent;
```

  - This merely declares a variable – no actual object is made

Constructing / Instantiating Objects

- The **new** operator is used to create an instance



new operator

Constructor: same name as object's Class which is the same name as the object's Reference Type. The constructor is a special Method of the Class Student.

```
aStudent = new Student("Mary Jane", "mjane@student.com", "Comp.Sci.");
```

Reference variable, must be of reference type Student

Constructor uses this data to initialise the object's attribute values

- One way to think of the semantics of this syntax is
  - The **new** operator allocates memory for all the objects' data (called ***attributes***)
    - And returns the address of this memory location to assign to the reference variable aStudent
  - The constructor initialises these attributes with values

Invoking Methods

- Applying a method to an object
  - Is called ***invoking*** or ***calling*** the Method
  - We can use the ***dot operator*** to invoke an object’s methods (behaviours)
  - A method invocation can be thought of as asking an object to perform a service
- e.g.

```
String temp = aStudent.getEmail();
```

- Some methods, such as getEmail(), return a value
  - We can use that value any valid way, e.g. assign it to a variable or print it.
  - In this case, it tells us the email address of the Student

Using Objects and Classes

- At this stage:
  - We will only use existing classes
  - You will only need to know how to declare reference variables (of these classes) and instantiate and use objects pointed at by these reference variables
- Later in the semester:
  - We will examine objects and classes further, including designing and writing our own classes
- Before we can use the existing classes, we need to know how to access them from our code

The String Class

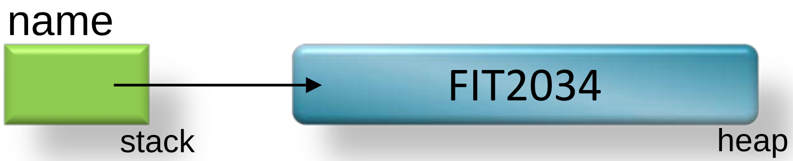
The String class type

- A String object is a sequence of 0 or more characters
  - String objects are “pointed at” by reference variables of type String which can have methods of the String class invoked on them (e.g. myString.length())
- Literal values of type String are enclosed in double quotes so the compiler does not mistake them for variable names
  - e.g. "FIT2034",
  - e.g. "FIT2034 Computer Programming 2"
- String literals *are* String objects
- String is in the java.lang package (no import needed)

The String Class

- We can instantiate a String object and point a reference variable at it in the normal way:
- e.g.

```
String name = new String("FIT2034");
```



- But in line with normal Java behaviour with respect to Strings you can also use the same syntax as used for primitive types
- e.g.

```
String name = "FIT1002";
```

Why does Java often (but not always) make it appear as if String is a primitive type? Perhaps for programmer convenience because Strings are used as much as primitives in most programs

Common Methods of the String Class

- In the following table
  - *Target string* is the string the method is invoked on
    - Note: None of the methods alter the Target string in any way

Method	Description
length()	Returns the number of characters in the target string as an integer
charAt(n)	Returns the character (type char) at the specified index of the target string Index is zero-based i.e. first character is at index 0
toUpperCase()	Returns a NEW string with all lower case letters in the target string converted to upper case
toLowerCase()	Returns a NEW string with all upper case letters in the target string converted to lower case
substring(n)	Returns a NEW string starting at index n (zero-based) of the target string and continuing to the end of the target string
substring(n1, n2)	Returns a NEW string starting at index n1 (zero-based) of the target string and continuing to index n2 - 1

2 Methods, same name, different “input data” ???

Length and Indexes in a String object

- Many methods require character indexes as input information to perform their actions
- This index is always zero-based
  - e.g. in the String “FIT2034” the “T” is at index 2 and the “F” at index 0
- The length of a string is how many characters there are
  - E.g. in the String “FIT2034”, the length is 7
  - The length() method will report the string’s length
- The indexes will be from 0, to one less than the length

String Concatenation

- Concatenation joins the character sequences of two strings together to make a new longer string character sequence
- e.g. with String literals
  - “cat and ” concatenated with “dog” evaluates to a new string “cat and dog”
- Now although this is performed by methods of a class from the Java Standard Class Library called StringBuilder, Java makes life easy for us:
  - When at least one operand of the + operator is a String object (literal or variable) the operation becomes concatenation not arithmetic addition (if the other operand is not a String object it’s converted if possible)

Which is why we managed to “+” String objects and integers/variables for display in some previous examples.

String example

```java
Scanner scan = new Scanner(System.in);
String name;

System.out.println("Please enter your name: ");
name = scan.nextLine();

System.out.println("Your name: " + name);
System.out.println("Your name (in uppercase): " +
   name.toUpperCase());
System.out.println("The length of your name: " +
   name.length());
```

```
Please enter your name:
Stephen
Your name: Stephen
Your name (in uppercase): STEPHEN
The length of your name: 7
```

Strings are immutable

- A String object is said to be Immutable.
  - This means that its value cannot be lengthened or shortened nor can the characters be changed once it has been created
- When a String object does need modification most commonly during a method invocation
  - Java automatically instantiates a new String object with the required modifications and returns that string object
  - e.g. myString.toUpperCase() does not alter myString in any way ***but returns a new string*** that is the same as myString but with all uppercase characters

MONASH University

Information Technology

Part 2

Control Flow in Java

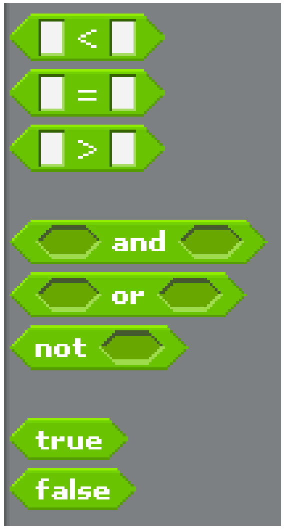
Control Structures

- The Flow of Control in a running program
  - Refers to the order in which the statements are executed
- The statements within a program are usually executed in textual sequence unless otherwise specified
- We use three code (text) **structures** to **control** the flow of control (they are called control structures):
  - Sequence
  - Selection (branch)
  - Repetition (loop)

Sequence is an implicit structure.
Statements are executed in the order of textual appearance, top to bottom.
Selection and Repetition are indicated by explicit textual structures one of which (Selection) we are about to learn the syntactic details of. Later today we will learn the syntactic details for Repetition.

Logical operators in *Scribble* and *Java*

- In Scribble we had a range of logical and relational operators:



- operators can be nested by snapping

- In Java we also have a range of logical and relational operators:

< <=

== !=

> >=

&&

||

!

true false

- operators can be nested using brackets

Relational Operators – Examples

==	equal to
!=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to

- Assuming that `x` is 3 and `y` is 6, these all evaluate to `true`

x != y	**(3 != 6)**
y > x	**(6 > 3)**
y >= x	**(6 >= 3)**
x < y	**(3 < 6)**

- ***Note difference between = and ==***

  = *means assignment to variable on left*

  == *means compare for equality*

- With chars: uppercase letters are earlier than their lowercase, so

  'T' < 'b' is true, but **‘t' < ‘B'** is false,

  'b' != 'B' is true

Common Mistake: using = instead of ==

Compiler (Syntax) Error

✗

```
if  (year = 2010) {
   System.out.println("South Africa hosted world cup");
}
```

✓

```
if  (year == 2010) {
   System.out.println("South Africa hosted world cup");
}
```

- If the assignment involved a boolean variable, and the RHS was a boolean expression, the compiler would accept it, but it would store the evaluated expression:

  Logic Error, meant: ==

  - E.g. `if ( changesAreSaved = true ) { ... }` ✗

Comparing Strings

- Strings are objects
- Objects do not allow operators to be used
  - Except the concatenation operator for Strings
- Must use the equals() method to compare
  - Additional reasons are explained in future week
- Example:

```
String input;
boolean sameWord;
…
sameWord =  input.equals("Octopus");
```

Logical Operators – Behaviour

X	Y	X && Y (logical AND)	X || Y (logical OR)	!X (logical NOT)
true	true	true	true	false
true	false	false	true	false
false	true	false	true	true
false	false	false	false	true

- Java performs **short curcuit evaluation**
  - If it can determine from the first operand what the outcome will be, it doesn't even bother checking the second operand
  - If first operand is <u>false</u> for && ➔ will be <u>false</u>
  - If first operand is <u>true</u> for || ➔ will be <u>true</u>

Revised Operator Order of Precedence

Highest

Operator Precedence
()
!
*, /, %
+, -
< <= > >=
== !=
&&
||
=

Lowest

All Mathematical operators have a higher precedence than all binary equality/relational and logical operators (Not is unary).
Therefore in boolean expressions involving both types of operand do the maths operations first.

This low precedence ensures RHS expressions in an assignment statement are fully evaluated before their value is assigned to the LHS variable.

Common Mistake: Java is not Maths

Compiler (Syntax) Error

```
if  (0 < x < 99) {
   System.out.println("x is in range");
}
```

✗

- Result of `0 < x` will be true or false
- What does `true < 99` mean?

```
if  (0 < x && x < 99) {
   System.out.println("x is in range");
}
```

✓

Logical Operators – Examples

X=true, Y=false, Z=true, A = 5, B = 9, answer = ‘y’

- What is the result, and how soon is it known?

```
X && Z
Y || (A > 3)
(answer == ‘Y’) || (answer == ‘n’)
X && Y || X && Z
(A <= B) && (answer == ‘N’)
X || Y && Z
! (X || Y) && Z
```

Click for each answer

Selection Structures

Java Selection Control Structures

- The Java `if` statement has three main variations and can be nested to deal with various selection scenarios
  - One-way selection
  - Two-way selection
  - One-from-many selection
    - Implemented by nesting two-way selection structures
- The Java `switch` statement is an additional specialised selection control structure
  - It’s appropriate for a restricted subcategory of selection scenarios

Syntax for if statements

- One-way selection

```
if (booleanExpression)
   block
```

These are syntax templates:
The blue text is required.
The black italicised text is to be supplied by the programmer and normally requires some explanation.
We already know what a boolean expression looks like.

- Two-way selection

```
if (booleanExpression)
   if/true block
else
   else/false block
```

A **block** in Java means
EITHER

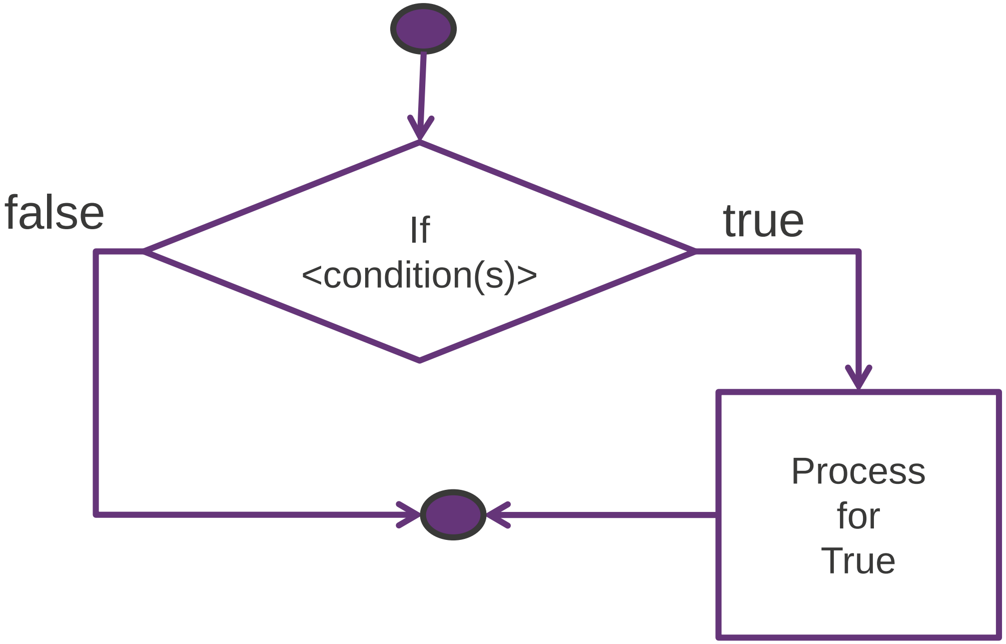
- A single statement terminated by a semi colon.

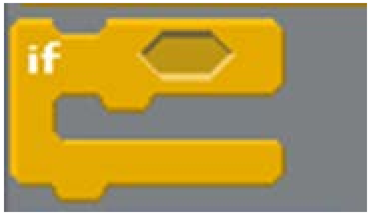
OR

- One or more statements each terminated by a semi colon and collectively enclosed by braces ({ … }).
  No semi colon is required after the terminating brace.

If using BlueJ, each block enclose by braces has a different background colouring.

One-way Decision Structure: simple if



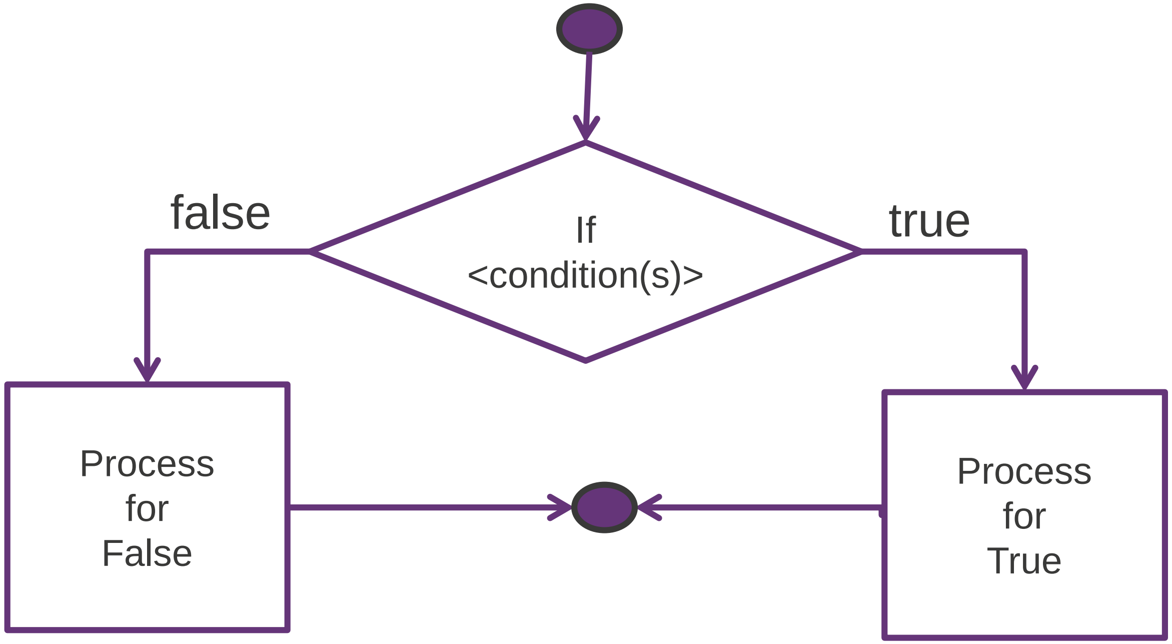


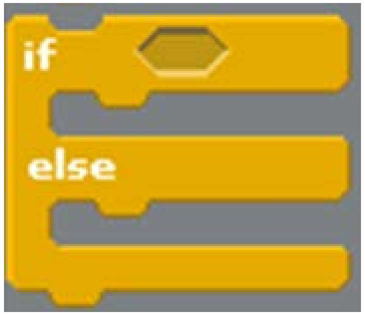
Scribble block

Java Example:

```
if (x % 2 == 1) {
   System.out.println("x is odd");
}
```

Two-way Decision Structure: if … else





Java Example:

```
if (x % 2 == 1)
   System.out.println("x is odd");
else
   System.out.println("x is even");
```

Time Out Question

- Q. Which (if any) of the following code fragments are equivalent in behavior to each other?

```java
if (x % 2 == 1){
   System.out.println(x);
}

System.out.println(" is odd");
```

```java
if (x % 2 == 1)
   System.out.println(x);
   System.out.println(" is odd");
```

```java
if (x % 2 == 1){
   System.out.println(x);
   System.out.println(" is odd");
}
```

Indentation, Brace Placement

Your IDE may indent automatically as you type. It may also have a correct existing indentation and formatting functions.

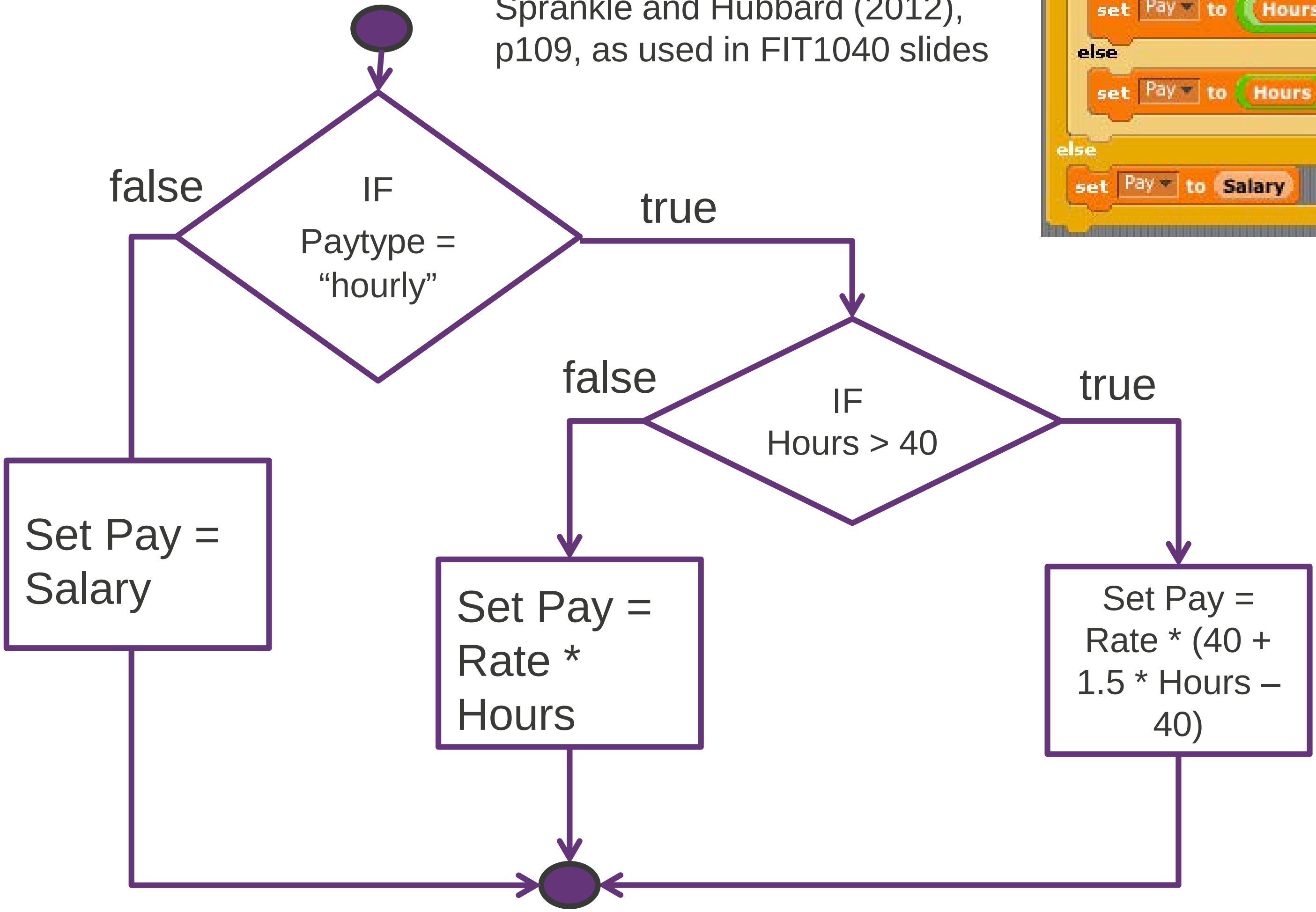
- Indentation
  - Indenting program statements with the tab key so that they reflect the control structures they belong to makes programs easier to read, understand and change
  - Beware! Poor indentation can mislead programmers but will never mislead the compiler
    - See the 2nd fragment on the previous slide
- Brace Placement
  - There are two popular and syntactically equivalent styles
  - Choose one of these styles and stick with it!

```java
if (x % 2 == 1)
{
    System.out.println(x);
    System.out.println(" is odd");
}
```

```java
if (x % 2 == 1) {
    System.out.println(x);
    System.out.println(" is odd");
}
```

If-Then structures are typically nested

Example adapted from Sprankle and Hubbard (2012), p109, as used in FIT1040 slides



IF
Paytype =
“hourly”

false

true

IF
Hours > 40

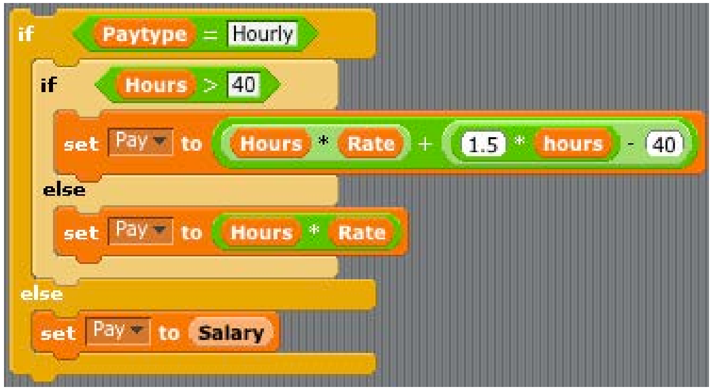
false

true

Set Pay = Salary

Set Pay = Rate * Hours

Set Pay = Rate * (40 + 1.5 * Hours – 40)



```
if Paytype = Hourly
  if Hours > 40
    set Pay to ((Hours * Rate) + ((1.5 * hours) - 40))
  else
    set Pay to (Hours * Rate)
else
  set Pay to Salary
```

MONASH University

If-Then structures are typically nested

- In Java:

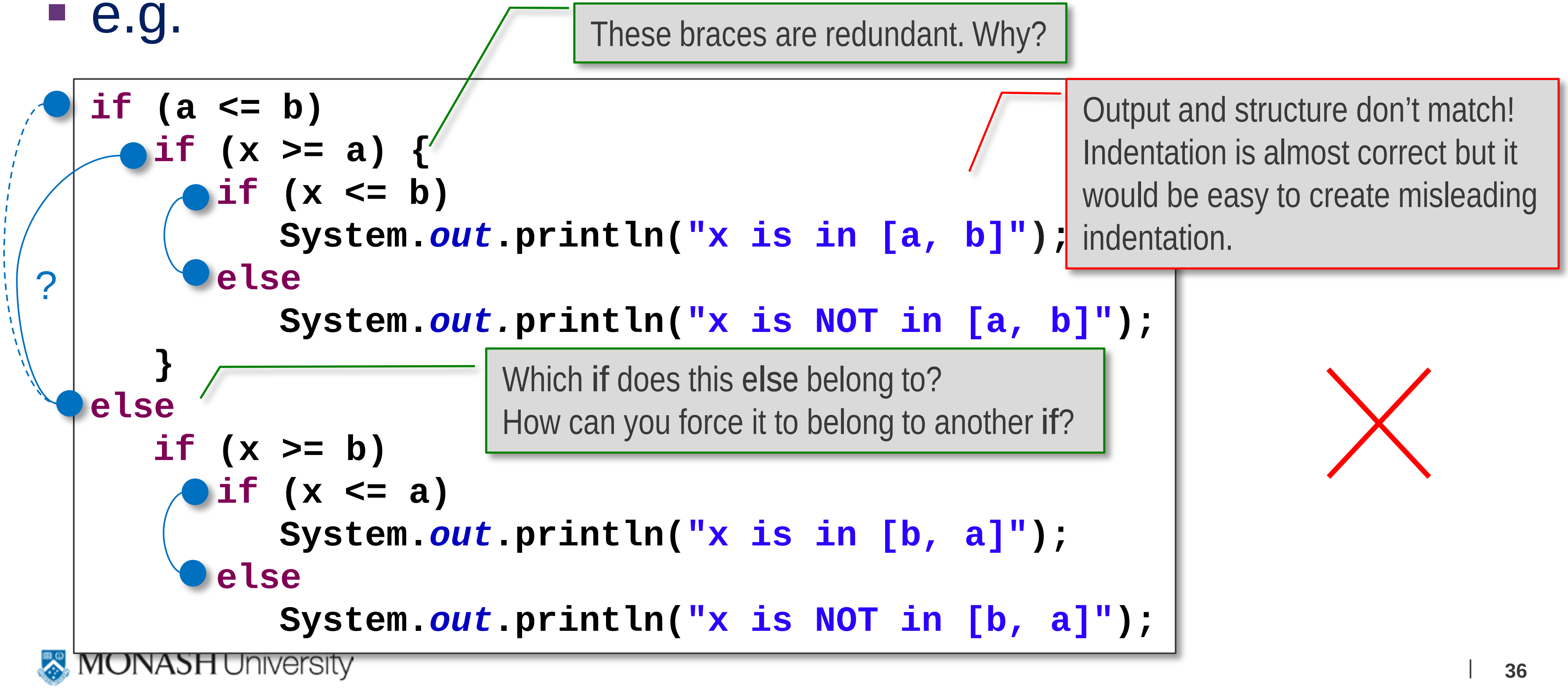
```
final int RATE = 22.71;
...

if (paytype.equalsIgnoreCase("Hourly"))
{
      if (hours > 40)
            pay = RATE * (40 + 1.5 * hours – 40);
      else
            pay = RATE * hours;
}
else
      pay = salary;

System.out.println("The pay amount is: " + pay);
```

Treats upper and lower case as equivalent when comparing

Take notice of indentation formatting

Dangling else Problem

- In complicated if structures:
  - The if that an else is paired with can appear to be ambiguous
    - So we call it a “dangling” else
  - The compiler always pairs an else with the closest (backward), syntactically valid if
- e.g.



```
if (a <= b)
   if (x >= a) {
      if (x <= b)
         System.out.println("x is in [a, b]");
      else
         System.out.println("x is NOT in [a, b]");
   }
else
   if (x >= b)
      if (x <= a)
         System.out.println("x is in [b, a]");
      else
         System.out.println("x is NOT in [b, a]");
```

These braces are redundant. Why?

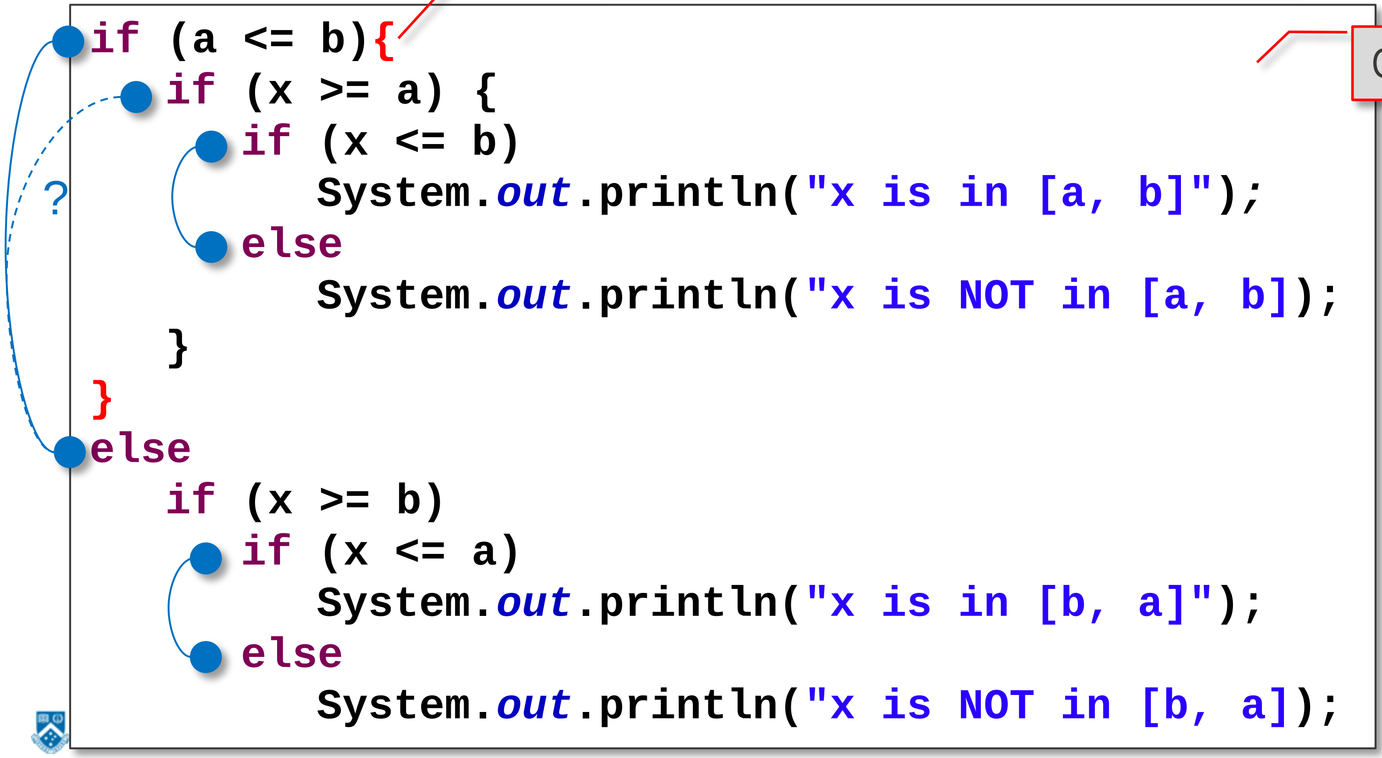
Output and structure don’t match! Indentation is almost correct but it would be easy to create misleading indentation.

Which if does this else belong to?
How can you force it to belong to another if?

Dangling else Problem

- The closest if to an else can be made syntactically invalid and therefore un-pairable
  - By introducing an additional set of braces
  - Now the next closest, syntactically valid if is paired
- e.g.

These braces change everything. Why?



```
if (a <= b){
   if (x >= a) {
      if (x <= b)
         System.out.println("x is in [a, b]");
      else
         System.out.println("x is NOT in [a, b]);
   }
}
else
   if (x >= b)
      if (x <= a)
         System.out.println("x is in [b, a]");
      else
         System.out.println("x is NOT in [b, a]);
```

Output and structure match!

✓

Selecting One-from-Many

- A control structure that selects one statement block from many can be built by:
  - Repeatedly nesting an if … else … structure in the else statement block of an enclosing if … else … structure
- e.g.

```
final int   JAN = 1, FEB = 2,MAR = 3, APR = 4,
            MAY = 5, JUN = 6,JUL = 7, AUG = 8,
            SEP = 9,OCT = 10,NOV = 11, DEC = 12;

int month = 0, days;
:
if (  month == JAN || month == MAR || month == MAY || month == JUL ||
      month == AUG || month == OCT || month == DEC)
   days = 31;
else if (month == SEP || month == APR || month == JUN || month == NOV)
   days = 30;
else if (month == FEB)
   days = 28;   // only correct for non-leap years
else
   System.out.println("invalid month number processed");
```

The indentation scheme employed here is common. It prevents the structure marching off the right of the page. It also reflects the underlying one-from-many flow-of-control structure.

Multi-way Decision Structure: Switch

- The switch statement is like a restricted one-from-many if statement
- Restrictions
  - All the conditions are equality conditions
  - They all test the value of the same expression
  - That expression’s type must be either char, byte, short or int
  - (From Java 7 onwards, it can also be a String object)
- Equivalence to one-from-many if structure
  - Any switch statement can be rewritten as a one-from-many if structure
  - The reverse is only true if the restrictions stated above apply
- Advantages over the equivalent one-from-many if structure
  - Syntactically more readable
  - Executes faster

Switch

- Syntax

```
switch (expression) {
   case expressionValue1:
      switchBlock1
      break;
   case expressionValue2:
      switchBlock2
      break;

   default:
      switchBlockDefault
}
```

Switch blocks are not like normal blocks. They are 0 or more statements all terminated with semi-colons. There are no braces.

- Example:

```
switch (menuChoice) {
   case 1:
      System.out.println("You Chose Menu Option 1");
      break;
   case 2:
      System.out.println("You Chose Menu Option 2");
      break;
   case 3:
      System.out.println("You Chose Menu Option 3");
      break;
   default:
      System.out.println("Error: Invalid Menu Option");
}
```

Tests for menuChoice == 1

Tests for menuChoice == 2

Tests for menuChoice == 3

menuChoice != 1
&& menuChoice != 2
&& menuChoice != 3

Repetition Structures

Repetition/Loops in Java

- Java provides three syntactical options for repetition:
  - For loop
  - While loop
  - Do .. While loop

For Loop

- The for loop is designed for when the number of iterations is known, and you want to increment or decrement a counter to know the current iteration

- Syntax:

```
for(initialisation; booleanexpression; inc/decrement)
   block
```

Loop header

Loop body

Loop condition

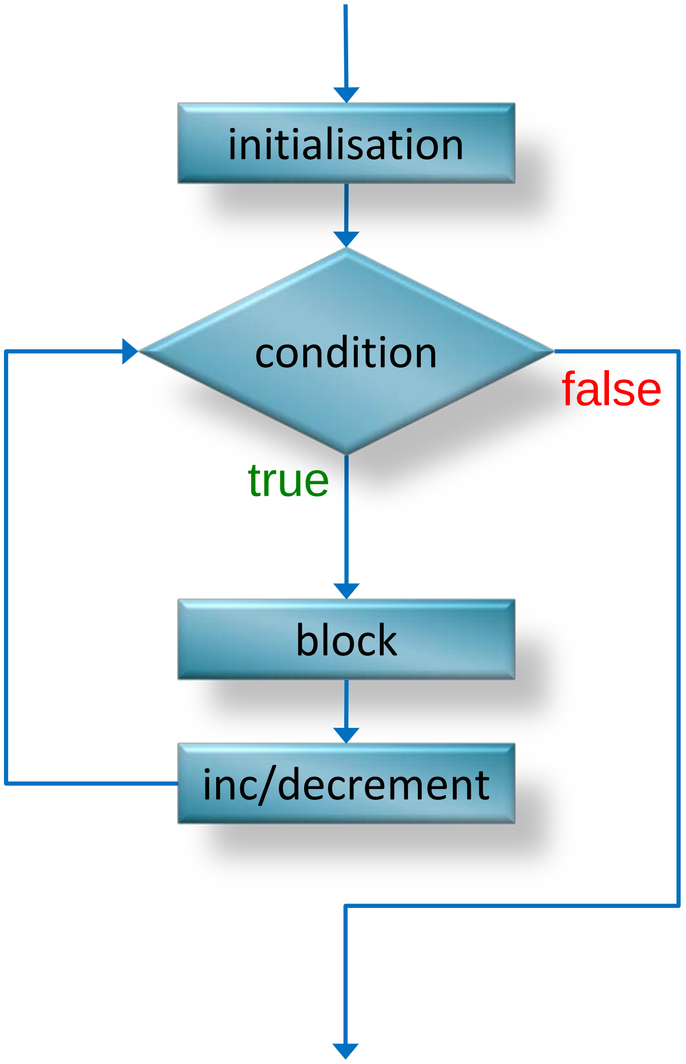
- Semantics
  - Execute the initialisation statement before anything else
  - Evaluate the boolean expression
    - If true execute the statement block
      - Execute the inc/dec statement
    - Else execute the next statement following after the entire loop control structure

repeat

exit

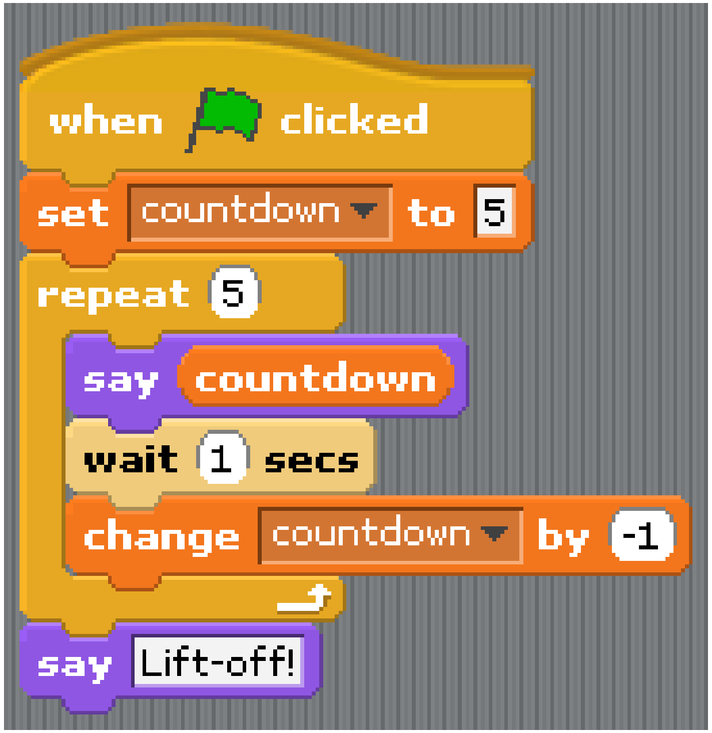
For Loop

- It's a pre-test loop and therefore can repeat 0 or more times
  - i.e. it's possible, but uncommon, to have no repetitions
- Infinite loops are possible if the loop condition is never false
- *initialisation* – executed once before first iteration
- *booleanExpression* – tested before each iteration (including 1st)
- *inc/decrement* – executed as last action in each iteration



For-Loop Example

- Counted loop in Scribble:



```
when [flag] clicked
set countdown to 5
repeat 5
    say countdown
    wait 1 secs
    change countdown by -1
say Lift-off!
```

- Counted loop in Java:

```
int counter;

for (counter = 5; counter > 0;
    counter--)
{
    System.out.println(counter);
}

System.out.println("Lift-off!");
```

Initialisation

Loop condition

Decrement

Loop body

```
5
4
3
2
1
Lift-off!
```

Counter Issues with `for` Loop

- The inc/decrement statement is not just limited to count = count + 1
  - e.g.
    - `count = count – 5` (or equivalently `count -= 5`)
    - `inputCount += 7`
- e.g.

```
for (int count = 20; count >= 0; count -= 7){
    System.out.println (count);
}
```

```
20
13
6
```

- Be careful, check
  - the initialisation statement allows the loop to start

```
for (int count = 20; count < 0; count -= 7){
    System.out.println (count);
}
```

✗ No Repetitions

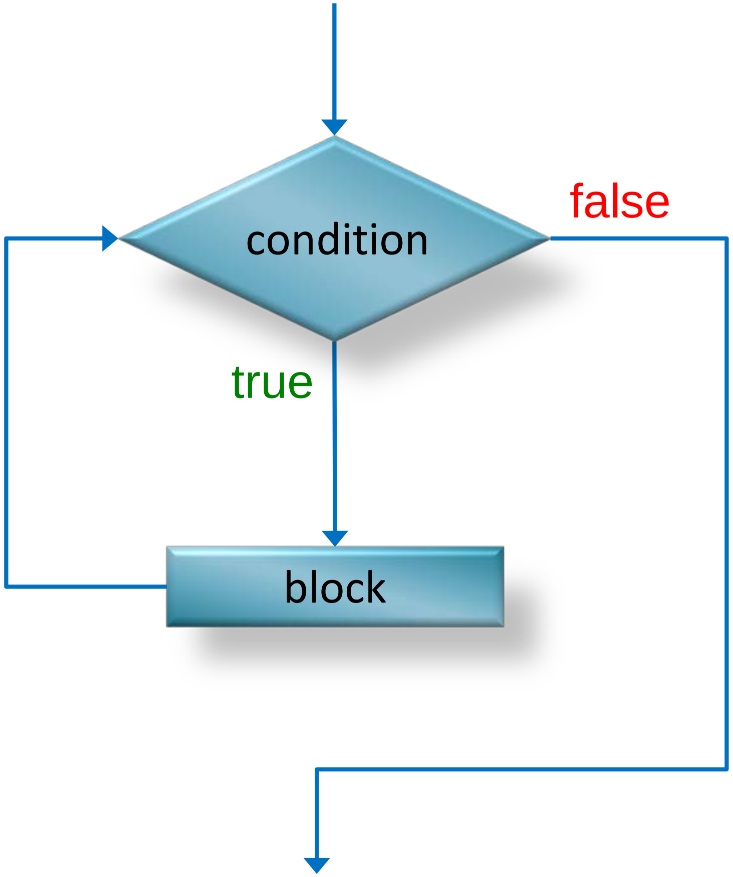
  - the inc/decrementing statement will eventually cause the loop condition to become false

```
for (int count = 20; count >= 0; count += 7){
    System.out.println (count);
}
```

✗ Infinite Loop

While Loop

- Another pre-test loop
  - the loop condition is tested before each loop execution including before the first repetition
  - It’s possible for a pre-test loop to occur 0 times
- If the loop is entered initially (i.e. loop condition evaluates to true initially) the statement block must eventually (during some repeat execution) cause the loop condition to evaluate to false
  - otherwise the loop will repeat infinitely



While Loop

- Syntax

```
while (booleanExpression)
   block
```

Loop header

Loop body

Loop condition

- Semantics
  - Evaluate the boolean expression
    - If true execute the statement/block
    - Otherwise execute the next statement after the entire loop control structure

repeat

exit

- Example:

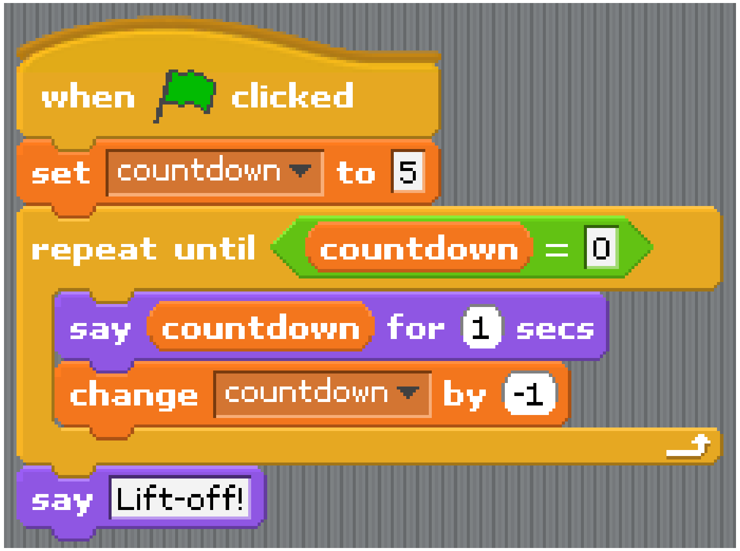
```
int count = 1;

while (count <= 5) {
   System.out.println (count);
   count++;
}
```

1
2
3
4
5

In this loop, this is the statement that eventually can change the loop condition's value from true to false to prevent an infinite loop

While Example – Comparison to Scribble

- Scribble example:
  - Repeat Until loops while the condition is false



```
when [flag] clicked
set countdown to 5
repeat until countdown = 0
    say countdown for 1 secs
    change countdown by -1
say Lift-off!
```

- Java Equivalent:
  - loops while the condition is true

```
int counter;

counter = 5;

while (counter != 0)
{
    System.out.println(counter);
    counter--;
}

System.out.println("Lift-off!");
```

Initialisation

Loop condition – logical opposite to Scribble condition

> 0 would be safer

Decrement

Loop body

While – Example

- A Guessing Game – Loop controlled by a boolean variable

```java
int guess;
int num= 5;
boolean done=false;
Scanner console = new Scanner(System.in);

while (!done){
   System.out.println("Guess the number I'm thinking of.");
   guess = console.nextInt( );

   if (guess == num){
      done=true;
   }
}
System.out.println("You guessed correctly!");
```

In this loop this is the statement that eventually can change the loop condition's value from true to false to prevent an infinite loop. Execution of this statement depends on the value of num which is input by the user.

```
Guess the number I'm thinking of.
2
Guess the number I'm thinking of.
9
Guess the number I'm thinking of.
5
You guessed correctly!
```

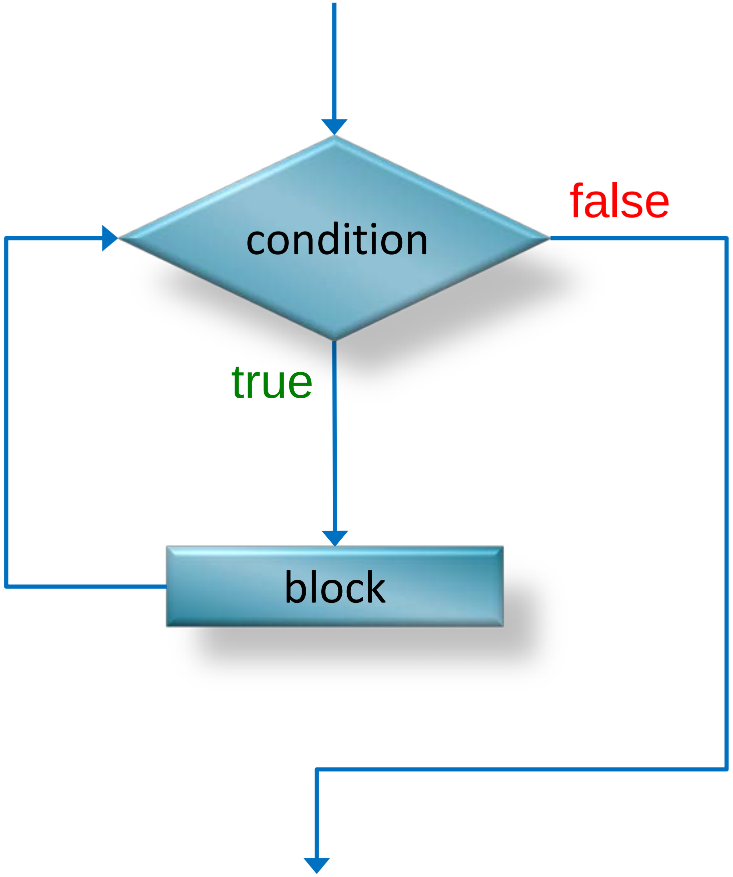
It's very important to understand that the statement that can potentially change the loop condition should be the last statement in the loop so the change is immediately detected on the next pre-test loop (top of the loop).
Any statements between this statement and the pre-loop test are still being executed when it is known the loop should be exited.

Do .. While

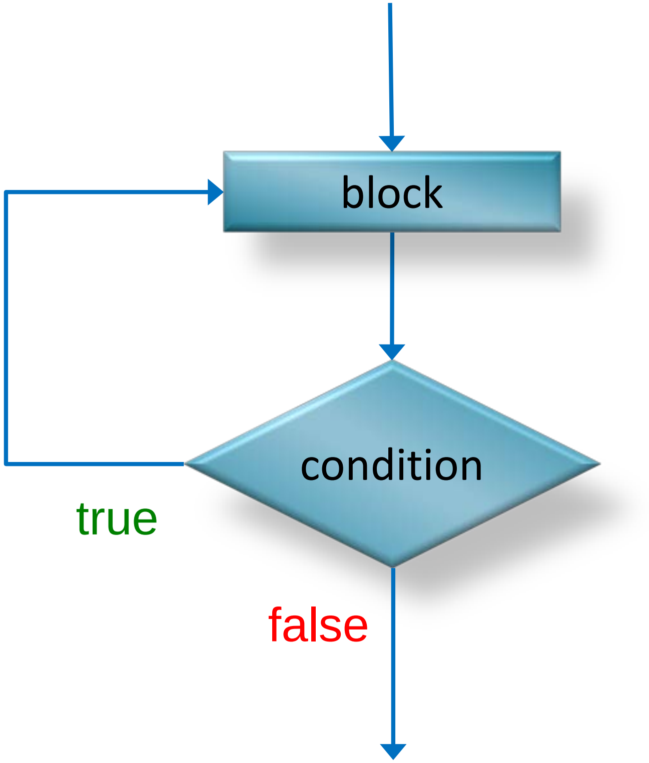
- Java’s post-test loop
  - the loop condition is tested after each loop execution including after the first repetition
  - A post-test loop’s body/statement must occur at least once
- Repeats the body, while the condition is still true
- Since the loop is entered initially regardless of the loop condition, the statement block must eventually (during some repeat execution) cause the loop condition to evaluate to false
  - otherwise the loop will repeat infinitely

While vs. Do .. While

- Java's While Loop



- Java's Do .. While loop



do ... while - Example

```
char response;  //variable to hold user response
Scanner scan = new Scanner(System.in);

do {
   System.out.println("play again(y/n)?");
   response = scan.next().charAt(0);    //get reply
} while(!(response == 'y' || response =='n'));
```

Potential loop condition value-changing statement

```
play again (y/n)?
t
play again (y/n)?
g
play again (y/n)?
y
```

- Notes
  - Initially the statement block is executed unconditionally
    - This is natural because we cannot validate user input without getting at least one response from them
    -
  - Repetitions:

Hand trace/Desk-check: variable and condition values at the bottom of each loop (post-test loop)

Repetition	Output	response	Loop condition
1	play again (y/n)	t	true
2	play again (y/n)	g	true
3	play again (y/n)	y	false

Time Out Question

- In the following code fragment how many times will the body of the loop be executed?
- What is the value of `count` when execution is complete?

```java
int max = 5;
int count = 0;

do{
    count++;
    System.out.println(count);
} while (count < max);
```

Time Out Question

- Q.
  - How many times will the body of the inner loop be executed in the following code fragment?

```
final int OUTER_REPS = 10;
final int INNER_REPS = 20;

int count1, count2;

count1 = 1;
while (count1 <= OUTER_REPS){
   count2 = 1;

   while (count2 <= INNER_REPS){
       System.out.print (((count1 - 1) * INNER_REPS) + count2);
       System.out.println (": " + count1 + ", " + count2);
       count2++;
   }

   count1++;
}
```

10 iterations

20 iterations

Note: there are 2 independent counters in use – one for each loop

not println

For each outer loop iteration the inner loop completes all its iterations.
Therefore 10 * 20 = 200 inner loop iterations

```
1: 1, 1
2: 1, 2
3: 1, 3
:
18: 1, 18
19: 1, 19
20: 1, 20
21: 2, 1
22: 2, 2
:
:
178: 9, 18
179: 9, 19
180: 9, 20
181: 10, 1
182: 10, 2
:
197: 10, 17
198: 10, 18
199: 10, 19
200: 10, 20
```

Which Loop?

- Use a for … loop
  - If the number of repetitions can be determined prior to executing the loop
    - i.e. it’s a counter controlled loop
- Use a while … loop
  - If the number of repetitions cannot be determined prior to executing AND
  - Zero or more repetitions are possible AND
  - The loop condition has to be checked at the beginning
- Use a do … while loop
  - If the number of repetitions cannot be determined prior to executing AND
  - One or more repetitions are possible
  - This is a much rarer case than zero or more repetitions

Using Loops to Validate Input

- Validating means ensuring a sensible value is given before proceeding
- Example:

```
int numerator, denominator;
Scanner scan = new Scanner(System.in);

System.out.println("Please enter a numerator:");
numerator = scan.nextInt();

do {
   System.out.println("Enter a positive denominator:");
   denominator = scan.nextInt();
} while (denominator < 1);     // achieves validation

System.out.println( numerator * 1.0 / denominator );
```