Introduction to Programming

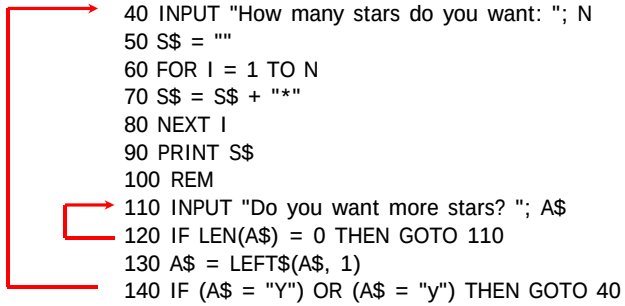
4: Program Control Structures:

Part 1 - Sequence & Selection

Program flow and controlling it

- Early programming languages used very simple flow control
  - GOTO – transferred control to a named or numbered statement – e.g. GOTO 1000
  - GOSUB – transferred control to a named or numbered subroutine – e.g. GOSUB 1200
  - RETURN – transferred control from a subroutine back to where it was called from
  - IF...THEN GOTO – transferred control conditionally
- Since these were the only tools to work with, they were the basis of all programming tasks
  - Assembly language is still the same

Example BASIC program



```
20 PRINT "Hello "; U$
30 REM
40 INPUT "How many stars do you want: "; N
50 S$ = ""
60 FOR I = 1 TO N
70 S$ = S$ + "*"
80 NEXT I
90 PRINT S$
100 REM
110 INPUT "Do you want more stars? "; A$
120 IF LEN(A$) = 0 THEN GOTO 110
130 A$ = LEFT$(A$, 1)
140 IF (A$ = "Y") OR (A$ = "y") THEN GOTO 40
150 PRINT "Goodbye ";
160 FOR I = 1 TO 200
170 PRINT U$; " ";
180 NEXT I
190 PRINT
```


The problem with GOTO

- It is too powerful
  - Programmer can transfer control to any other point in a program – either direction
  - As a program evolves, GOTO can be used to direct flow around bits of code – easy way to deal with bugs or odd circumstances
  - Led to programs with so many un-planned control transfers that the name used for this style of coding became ‘spaghetti programming’
- The lesson learned from this was that programmers should not have too much power – structured programming limits this

Typical use of GOTO

- In this example, GOTO forms the equivalent of a For..Next loop (For counter = 0 To 10...)

```
1000 COUNTER = 0
1010
1020    Code to be repeated goes here…
1030
1040 COUNTER = COUNTER + 1
1050 IF COUNTER < 10 THEN GOTO 1010
```

- Problems with this...
  - The GOTO statements are tied to specific lines
  - Programmer could add other GOTO statements to 'work around' code that needs changed

Structured programming

- By contrast, structured programming permits very few structures, following simple rules
  - Any section of code should have one entry point and one exit point
  - General flow of a structure (e.g. a loop) should always be top to bottom
  - Never use unconditional transfers (GOTO), or use them very carefully and sparingly (e.g. Error handling in BASIC programs)
  - Note that there is no GOTO statement in Java

Benefits

- Structured programming imposes structure on program code
  - Code can be analysed at any level of detail
  - Software can do the analysis
  - Consequent rigour makes software production much more like *engineering*
- Structured programming blocks can be diagrammatically represented
  - No longer only programmers who can understand software design
  - Analysts, Quality Assurance people, hardware specialists, Human Computer Interaction specialists etc.

Block – the simplest structure

- Groups a sequence of statements into a single statement
  - Entry point at top, exit point at bottom

```
{   // this block exchanges the values of x and y
    int temp;   // temporary variable within block
    temp = x;   // save a copy of x's value in temp
    x = y;      // copy the value of y into x
    y = temp;   // copy the value of temp into y
}
```

Empty block

- Often useful to have a block containing no statements
- Generally written on one line

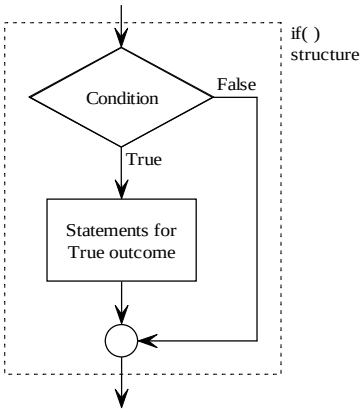
```
{}  // an empty block
```

- Will meet examples where empty blocks are used later

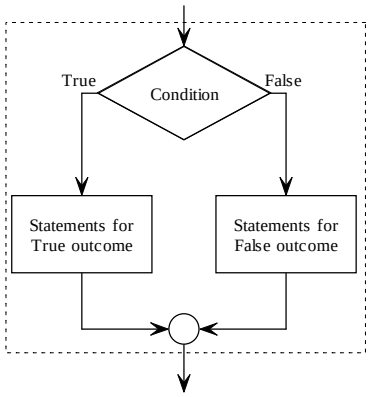
Selection

- Blocks just group a sequence of statements – no direct effect on control flow
- Selection (conditional statements)
  - These provide structures that choose which of a set of statements to execute depending on some condition or value
    - if statement
    - switch statement

Selection I



if( ) structure



if( )

if( )...else

if()...

- An if statement is used to determine whether to execute a statement based on a condition (a **boolean** expression).
- When the computer executes the statement it evaluates the expression
  - If the expression evaluates to **true** it executes the statement.
  - If the expression evaluates to **false** it skips the statement.

if()...

- For example

```
if (age >= 18)
      sendOutVotingCard();
```

- If the condition is false (age is less than 18) this if statement does *not* execute the statement but skips it
- Note that a block is necessary if you want to execute more than one statement if the condition is true
  - It is good practice to always include a block even if there is just one statement

if()...

- This example from the book (page 67) exchanges the values of x and y (same code as in the block example we saw earlier), but only if x is greater than y to begin with
  - After this if statement executes it is guaranteed that x is less than or equal to y

```
if (x > y) { // Note the block.
   int temp; // A temporary local variable.
   temp = x; // Save a copy of the value of x.
   x = y;    // Copy the value of y into x.
   y = temp; // Copy the value of temp into y.
}
```


if()...else

- An if statement with an else part is used if there are different actions to perform when the condition is true than when it is false

```
if  (mark >= passMark) {
      pass = true;
      credits += 20;
} else {
      pass = false;
      enrolForResit();
}
```


if()...else if()...

- Can string together any number of if else statements to get multi-way selection

```
if (value == 0) {
      System.out.println("value is zero");
} else if (value % 2 == 0) {
      System.out.println("value is even");
} else {
      System.out.println("value is odd");
}
```

if()...else if()...

- Once an else-if condition is reached that is true no other else-ifs/else parts are evaluated or executed
  - Code below outputs `value < 5`
  - Does not output `value < 10` even though that is also true

```
int value = 1;
if (value == 0) {
      System.out.println("value is zero");
} else if (value < 5) {
      System.out.println("value < 5");
} else if (value < 10) {
      System.out.println("value < 10");
} else {
      System.out.println("else part");
}
```


The “dangling else” problem

- As an if statement is a statement, an if statement may contain another if statement as the statement to execute when its condition is true
- Potential problem for an if-else statement when the if statement it contains has no else

“dangling else” problem cont’d

- Consider example below
  - Intention is that student takes the next test if the number of tests they have taken does not equal the number of tests to take
  - Not what happens in this example as an **else** always belongs to the closest preceding **if** (the compiler ignores indentation)!

```
if (testsTaken == numberOfTests)
      if  (mark >= passMark)
            pass = true;
else
      takeNextTest();
```

... goes with this if

this else...

Avoiding the problem

- Using blocks avoids the problem
  - one reason why it is good practice to always use blocks with if statements

```
if (testsTaken == numberOfTests) {
      if  (mark >= passMark) {
            pass = true;
      }
} else {
      takeNextTest();
}
```

Compound Conditions

- Two possible approaches
  - Nested if() statements...

```
if (age >= 17) {
    if (hasCurrentLicence) {
        if(drivingBans==0) {
            // A potential driver
        }
    }
}
```

  - Use logical operators...

```
if ((age >= 17) && hasCurrentLicence && (drivingBans==0)) {
      // A potential driver
}
```


Logical operators

- Can be used to combine conditions according to rules of Boolean logic...

Condition A	Condition B	A && B (AND)	A || B (OR)		Condition X	!X (NOT)
TRUE	TRUE	TRUE	TRUE		TRUE	FALSE
TRUE	FALSE	FALSE	TRUE		FALSE	TRUE
FALSE	TRUE	FALSE	TRUE			
FALSE	FALSE	FALSE	FALSE			

"Short circuit" operators

- The operators && and || are evaluated left to right
- Evaluation stops as soon as the value of the whole expression can be determined
- A && B
  - If A is **false** condition B is not evaluated as the whole condition must be **false**
- A || B
  - If A is **true** condition B is not evaluated as the whole condition must be **true**

Compound conditions and parentheses

- When using && and || in compound conditions make sure you include parentheses to indicate precedence

```
boolean a = false;
boolean b = true;
boolean c = true;

if ((a && b) || c) TextIO.putln("One");
// Output: One (as c is true)

if (a && (b || c))   TextIO.putln("Two");
// No output: (as a is false) - nothing to the
// right of && is evaluated
```

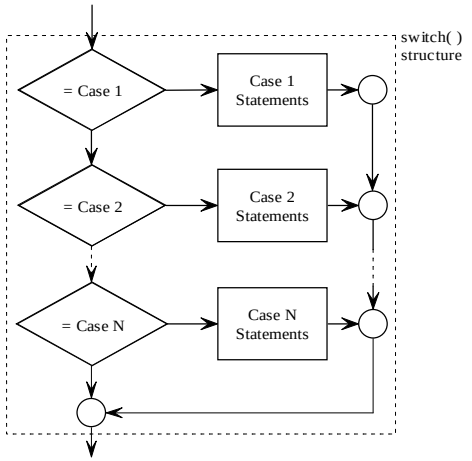
De Morgan's Rules

- Negating compound conditions can be confusing.
- De Morgan's rules:

!(A && B) is the same as (!A || !B)

!(A || B) is the same (!A && !B)

Selection II



This flowchart shows idealised switch without fall-through (see later)

switch( )

switch()

- This allows multiple selection of alternatives, based on an integer (though not **long**), character, or enumeration value and, from Java 7, also a String value
- e.g.

```
int choice = TextIO.getlnInt(); // from a menu
switch (choice) {
   case 0 : doChoice0(); break;
   case 1 : doChoice1(); break;
   case 2 : doChoice2(); break;
   default : TextIO.putln("Invalid choice");
}
```


switch() statement continued

- Behaviour identical to switch statement in C and C++
  - Some regard this as a mistake by the designers of Java (see the textbook, page 100, for example).
- The cases can be listed in any order
  - Unless there is good reason, list them in ascending order
- Duplicate cases are not allowed (won't compile)
- Each case must be a constant (the compiler must be able to work out what the case is)
- Including **default** is optional, if present it matches any value not explicitly listed in the cases
- If a case does not include a break statement execution continues to the next case (*fall-through*)!
  - This leads to programming errors so use switch with care

switch() fall-through

```
switch ( N ) { // N is an integer variable, from page 99 of book
        case 1 :
        case 5 :
                System.out.println("N is 1 or 5");
                break;
        case 2 :
        case 4 :
        case 8 :
                System.out.println("N is 2, 4, or 8");
                System.out.println("That's a power of 2!");
                break;
        case 3 :
        case 6 :
        case 9 :
                System.out.println("N is 3, 6, or 9");
                System.out.println("That's a multiple of 3!");
                break;
        default : System.out.println("N is 7 or not in range 1..9");
}
```


Fall-through continued...

- Fall-through is sometimes useful. To judge when it is appropriate follow this rule:
  - Each **case** should include a **break** statement *unless* the **case** is completely empty (as on previous slide)
    - In C# the compiler enforces this rule, in Java you will need to do it yourself

switch() *versus* if()...else if()

- You can think of a switch statement as simulating the effect of an if()...else if() statement, but in a clearer format

```
if (N==1 || N==5) {
   System.out.println("N is 1 or 5");
} else if (N==2 || N==4 || N==8) {
   System.out.println("N is 2, 4, or 8");
   System.out.println("That's a power of 2!");
} else if (N==3 || N==6 || N==9) {
   System.out.println("N is 3, 6, or 9");
   System.out.println("That's a multiple of 3!");
} else {
   System.out.println("N is 7 or not in range 1 .. 9");
}
```


In this lecture...

- Control flow
  - Structured programming imposes rules on control flow in order to manage it effectively
  - Statements are executed in order, top to bottom, in the absence of any other structure
  - Selection statements choose which of a set of statements are executed based on a condition or value (the book refers to this as *branching*)
    - In Java, if statements (including if..else and if..else if) and switch statements

Next week

- More exercises on what we have covered so far
- Reading over next two weeks:
  - Eck, chapter 3
    - Sections 3.1 to 3.7 cover the material in this week's lecture and the next lecture
    - 3.8 includes brief introduction to arrays
    - You can ignore section 3.9