

Revision

- To perform multi-way branching, one can use the nested if/else statement.
- Since the else part may be omitted from an if/else statement, it may happen that an else statement is connected to an if statement that the programmer does not expect.
- This problem, called the dangling else problem, can be solved by using compound statements, or using else-part with a null statement.
- When designing more complicated programs, we employ a technique called top-down design to write programs.
- By considering the big problem first and by testing early, top-down design allows problems to be discovered and resolved quickly.

0911B L09-1

Switch: another way for multi-way branching

At times you want to make a multi-way branch based on the value on a single integer or character type expression.

At such time, the if/else statement construct cannot express your intentions clearly. We need to look at every statements to be sure that the choice is based on the same expression. Example:

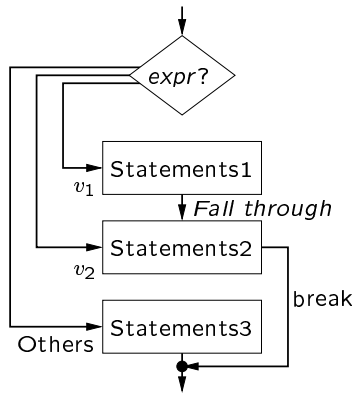
```
import chapman.io.*;
public class KiloOrPound {
    public static void main(String[] args) {
        StdIn in = new StdIn();
        double weight = in.readDouble();
        System.out.print("In [p]ounds or [k]ilograms? ");
        char ch = in.readChar();
        if (ch == 'p' || ch == 'P') {
            weight /= 2.2;
        } else if (ch == 'k' || ch == 'K') {
            // Do nothing
        } else {
            System.out.println("Wrong input!");
        }
    }
}
```

0911B L09-2

General form of switch

```
switch (expr) {
    case v1:
        Statements1
        // FALLTHROUGH
    case v2:
        Statements2
        break;
    default:
        Statements3
}
```

Note: *expr* must be of char, byte, short or int type; *v1* and *v2* must be constants.



0911B L09-3

Example usage

In most cases, switch is used to realize multi-way branches. If used correctly, it reveals your intentions much clearer than nested if/else statements.

Fall-throughs are usually used to implement multiple case labels.

```
import chapman.io.*;
public class KiloOrPound {
    public static void main(String[] args) {
        StdIn in = new StdIn();
        double weight = in.readDouble();
        System.out.print("In [p]ounds or [k]ilograms? ");
        char ch = in.readChar();
        switch (ch) {
            case 'p': case 'P':
                weight /= 2.2;
                break;
            case 'k': case 'K':
                break;
            default:
                System.out.println("Wrong input!");
        }
    }
}
```

0911B L09-4

Fall through: example

Most of the time, you want to put "break" after each non-empty case. When you intentionally not to put one, you usually write a comment like "// FALLTHROUGH" to signify that it is intentional and not a bug.

```
import chapman.io.*;
public class KiloOrPound {
    public static void main(String[] args) {
        StdIn in = new StdIn();
        double weight = in.readDouble();
        System.out.print("In [p]ounds or [k]ilograms? ");
        char ch = in.readChar();
        switch (ch) {
            case 'p': case 'P':
                weight /= 2.2;
                // FALLTHROUGH
            case 'k': case 'K':
                System.out.println("Weight = " + weight + " kg.");
                break;
            default:
                System.out.println("Wrong input!");
        }
    }
}
```

0911B L09-5

The need of looping

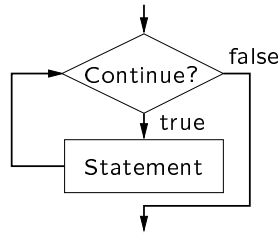
- None of the statements we learnt allow repetitions.
- That means, to run 30 statements you must write at least 30 statements. (You need more if some are if/else statements.) Each statement is executed at most once.
- With "loops", one statement can be executed multiple times in your program. It stops only when certain condition is met.
- Most programs have something repetitive, and thus require loops.
- By having different values of variables, different things may happen each time the statement is executed. Writing such loops requires careful thinking about what to do each time the loop is executed.

0911B L09-6

The while loop: general form

while (*continue*)
Statement

If *continue* evaluates to true, the *Statement* is executed. This is repeated until a time when *continue* evaluates to false.



0911B L09-7

Example while loop

```
/* Prints a square table */
import chapman.io.*;
public class SqTable {
    public static void main(String[] args) {
        StdIn in = new StdIn();
        System.out.print("Largest number: ");
        int largest_num = in.readInt();
        int i = 1;
        while (i <= largest_num) {
            System.out.println(i + " square is " + i*i);
            i += 1;
        }
    }
}
```

Key feature: Each time the loop content is executed (i.e., in each iteration), it performs the required operation, and modify *i*. This allows the condition *i* <= *largest_num* to check whether the job is done.

0911B L09-8

More complicated example

Finding 5-days average of a stock (content of program only).

```
StdIn in = new StdIn();
double day1 = 0, day2 = 0, day3 = 0, day4 = 0, day5 = 0;
int current_day = 1;
System.out.print("Day " + current_day + " value? (0 stops program) ");
day5 = in.readDouble();
while (day5 != 0.0) {
    // Find average from the 5th day
    if (current_day > 4) {
        System.out.print("Day " + current_day + " average: ");
        System.out.println((day1 + day2 + day3 + day4 + day5) / 5);
    }
    // Fill in the value of all days
    current_day += 1;
    day1 = day2;
    day2 = day3;
    day3 = day4;
    day4 = day5;
    System.out.print("Day " + current_day + " value? (0 stops program) ");
    day5 = in.readDouble();
}
```

0911B L09-9

Some more shorthands: increment and decrement

When dealing with loops, frequently it requires adding or subtracting one from some integer variables. It occurs so often that there are operators for such manipulations.

- ++*var*, *var*++: add one to *var* (increment).
- --*var*, *var*--: subtract one from *var* (decrement).

Note that there are two variants of each form. Their differences is in the value of the expression (++ and -- are operators!). In both cases the value is that of *var*.

When ++ and -- are placed before *var* (prefix operation), the increment or decrement is done before the value of *var* is fetched.

When ++ and -- are placed after *var* (postfix operation), the increment or decrement is done after the value of *var* is fetched.

0911B L09-10

Example

Most of the time the value of the increment/decrement is not used. In this case prefix and postfix operators behaves the same. But there are some situations when the values are actually used.

```
public class TestIncDec {
    public static void main(String[] args) {
        int a = 10, b = 20;
        System.out.println("Before pre-increment: " + a);
        System.out.println(++a);
        System.out.println("After pre-increment: " + a);
        System.out.println(--a);
        System.out.println("After pre-decrement: " + a);
        System.out.println("Before post-increment: " + b);
        System.out.println(b++);
        System.out.println("After post-increment: " + b);
        System.out.println(b--);
        System.out.println("After post-decrement: " + b);
        // How about these?
        ++ ++a;
        System.out.println(a--b);
    }
}
```

0911B L09-11

Precedence table again

We have learnt most of the operators that we will ever learn in this course. The three remaining ones are [], ?:, and new.

Operators	Associativity
., method (), postfix ++, postfix --	left
unary +, unary -, !, prefix ++, prefix --	right
casting ()	right
*, /, %	left
binary +, binary -	left
<, <=, >, >=	left
==, !=	left
&	left
^	left
	left
&&	left
	left
=, +=, -=, *=, /=, %=	right

0911B L09-12