Program control: Decisions and operators

Generally speaking you aren't learning much when your lips are moving

Lecture objectives

To be able to understand the following fundamental concepts of the Java programming language:

- formatting output
- the flow of control
- decision-making statements
- operators for making complex decisions

Output

- Printing a double
  - Zero is always printed as 0.0
  - A whole number stored as a double is printed with one zero after the decimal place eg 999.0
- Other number are printed in their entirety eg.
  654321.347628736560918 or 0.3333333333333333

- How do we restrict output to 2 decimal places?
- What about leading and trailing zeros?
- Put in a comma for numbers > 1000?
- What about printing currency and percentages?

Formatting output

- Define a pattern that represents how the output should look

- # represents a character where no leading or trailing zeroes are printed, and decimals are rounded
  - for the pattern “###.##” 2.100000000000 will print as 2.1
- 0 represents a character where leading or trailing zeros are printed
  - for the pattern “000.00” 2.100000000000 will print as 002.10
- , represents a comma inserted for a number > 999
  - for the pattern “,###.00” 555212.0 will print as 555,212.00

Creating objects

- Last week you created objects of the `String` class

```
String title = “hello”;
```

- Now we need to learn to create other objects
- The usual way is:

```
Class object = new contructor(parameters)
```

- (note the constuctor has the same name as the Class)

Formatting Output

- DecimalFormat class is imported from *java.text*

DecimalFormat *object* = new DecimalFormat (*String pattern*)

eg.

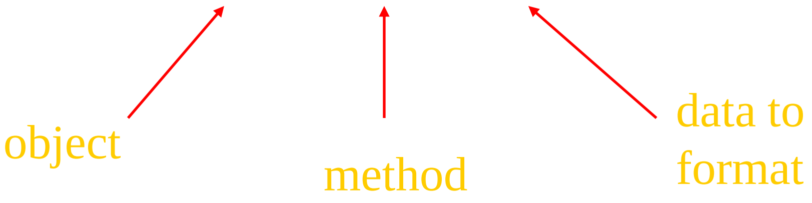
DecimalFormat fmt = new DecimalFormat(“0.00”);

The `format` method

- The `DecimalFormat` class has a method called `format()`
- This is applied to the data that we want to format eg *`area`* and returns a String containing the formatted number

```
    String format (double number)
```

```
DecimalFormat fmt = new DecimalFormat(“0.00”);
System.out.println ("The circle area: " +  fmt.format(area));
```



```
import java.text.DecimalFormat;

public class CircleStats
{
   public static void main (String[] args)
  {
     int radius = 5;
     double area, circumference;

     area = Math.PI * Math.pow(radius, 2);        // area = 78.53981633974483

     circumference = 2 * Math.PI * radius;        // circumference = 31.41592653589793

    DecimalFormat fmt = new DecimalFormat ("0.###");  // Round to three decimal places,
                                                       // no trailing zeros

     System.out.println ("The circle's area: " + fmt . format(area));
     System.out.println ("The circle's circumference: “ + fmt . format(circumference));
  }
}
```

```
The circle's area: 78.54
The circle's circumference: 31.416
```

Other formatting symbols

- $
  - inserts this symbol for currency printing
  - eg “$,###.00”

- %
  - Multiply by 100 and add a % sign
  - Eg “%”

```
import java.text.DecimalFormat;
public class Formatting
{
    public static void main (String[] args)
     {
                final double TAX_RATE = 0.06;
          int quantity = 10;
        double unitPrice = 5.0, subtotal, tax, totalCost;

         subtotal = quantity * unitPrice;
         tax = subtotal * TAX_RATE;
         totalCost = subtotal + tax;

        DecimalFormat money = new DecimalFormat(“$,###.00”);
         DecimalFormat percent = new DecimalFormat(“%”);

         System.out.println ("Tax: " + money . format(tax) + “ at “ + percent . format(TAX_RATE));
         System.out.println ("Total Price: " + money . format(totalCost));
     }
}
```

```
Tax: $3.00 at 6%
Total Price: $53.00
```

Flow of Control

- Unless indicated otherwise, the order of statement execution through a method is linear:
  - one after the other in the order they are written

- Some programming statements modify that order, allowing us to:
  - decide whether or not to execute a particular statement (this week)
  - perform a statement over and over repetitively (next week)

Boolean Expressions

- A condition often uses one of Java's *equality operators* or *relational operators*, which all return boolean results:

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

- Note the difference between the equality operator (==) and the assignment operator (=)

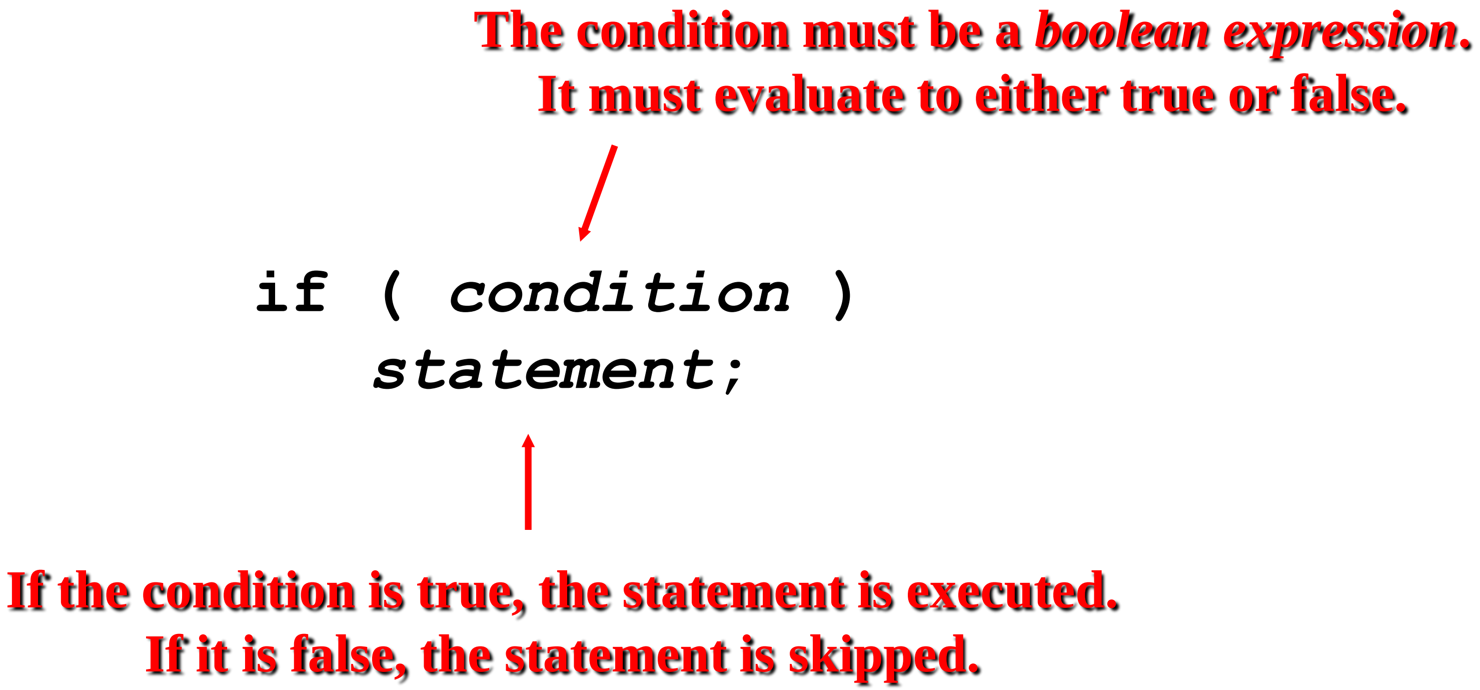
Conditional Statements

- A *conditional statement* lets us choose which statement will be executed next
  - ie give us the power to make decisions

- Java's conditional statements are
  - the `if` *statement*,
  - the `if-else` *statement*,
  - the `switch` *statement*

The if Statement

- The `if` *statement* has the following syntax:



The condition must be a ***boolean expression***. It must evaluate to either true or false.

```
if ( condition )
   statement;
```

If the condition is true, the statement is executed. If it is false, the statement is skipped.

The if Statement

- An example of an if statement:

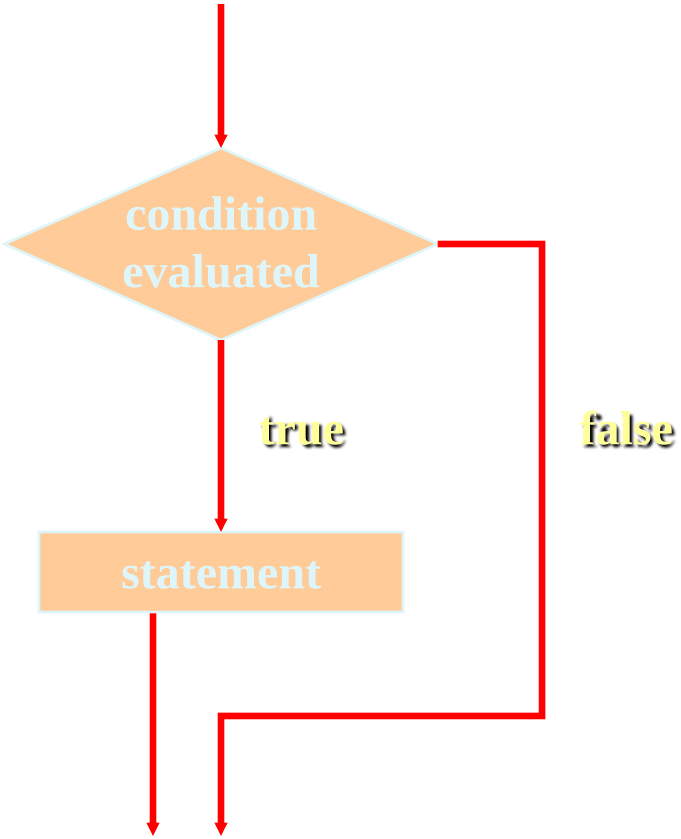
```
if (testResult > 0)
   total = total + testResult;
System.out.println ("The final mark  is " + total);
```

First, the condition is evaluated. It is either true or false

If the condition is true, the assignment statement is executed.
If it is not, the assignment statement is skipped.

Either way, the println is executed next.

Logic of an if statement



```
//  Age.java       Author: A.N. Oldie
//  Demonstrates the use of an if statement.
//***************************************************************
import javax.swing.JOptionPane;
public class Age
{
   public static void main (String[ ] args)
   {
            final int MINOR = 21;
            String ageStr = JOptionPane.showInputDialog ("Enter your age: ");
            int age = Integer.parseInt(ageStr);

            System.out.println ("You entered: " + age);

            if (age < MINOR)
                System.out.println ("Youth is a wonderful thing. Enjoy.");

            System.out.println ("Age is a state of mind.");
   }
}
```

Common Errors

- Assignment and equals

```
if (total = 100)
        System.out.println(“congratulations!”);
```

- This is not a condition
- It is an assignment and will not compile

Common Errors

- The empty statement

```
if (total > 100);
        System.out.println(“congratulations!”);
```

- In Java semicolons don’t go at the end of each line but
  - at the end of each complete declaration or statement
- This does not give a compiler error
- It is interpreted as “if true do nothing”

Common Errors

- What about multiple true statements?

```
if (total > 100)
        System.out.println(“congratulations!”);
        System.out.println(“you’re a genius!”);
```

- The second print line will always print
- For multiple statements we use a block

Block Statements

- Several statements can be grouped into a *block statement*
- A block is delimited by curly braces { … }
- A block statement can be used wherever a *statement* is called for in the Java syntax

```
if (temp > 40)
{
        System.out.println(“get a drink”);
        System.out.println(“get a book”);
        System.out.println(“relax!”);
}
```

- For consistency a block can be used for a single statement

The if-else Statement

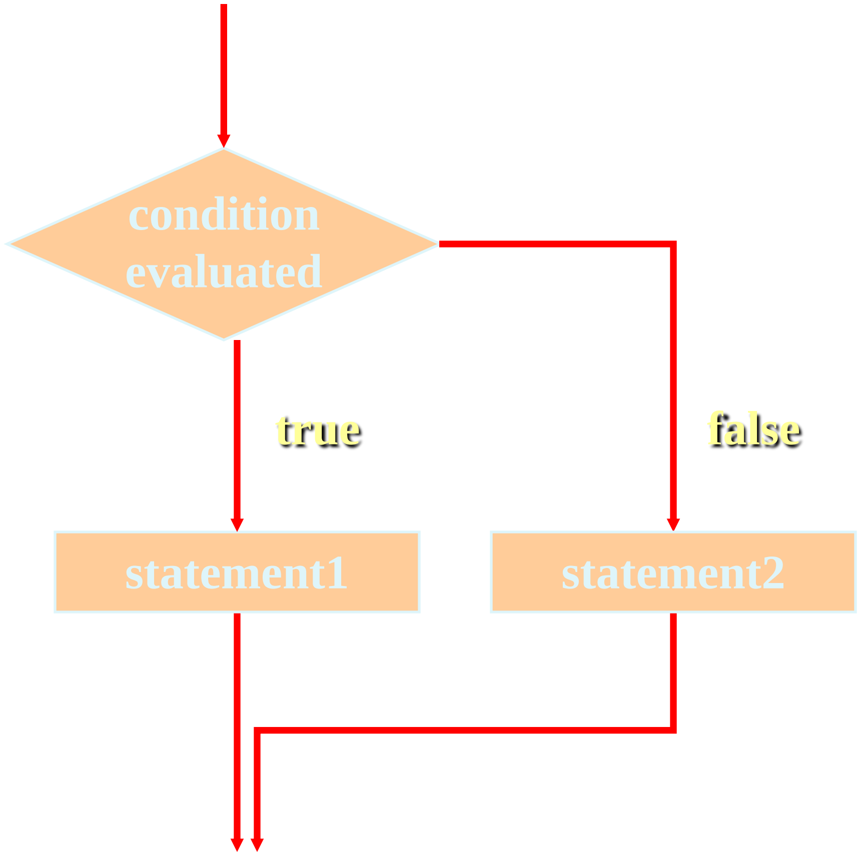
- An `else` *clause* can be added to an `if` statement to make it an `if-else` *statement*:

```
if ( condition )
    statement1;
else
    statement2;
```

- If the condition is true, *statement1* is executed; if the condition is false, *statement2* is executed

- One or the other will be executed, but not both
- Align the else under the if

Logic of an if-else statement



```
 if (hoursWorked > 40)
{
     regularPay = 40 * rate;
    overtimePay = (hoursWorked – 40)  * (1.5 * rate);
}
 else
 {
     regularPay = hours * rate;
     overtimePay = 0.0;
 }
System.out.println(“Regular pay is “ + regularPay”);

System.out.println(“Overtime pay is “ + overtimePay”);
```

Multiple independent `if`’s

- Consider a scale of commissions paid on salesTotal

Sales > 100,000	Commission = 10%
Sales > 50,000 and <= 100,000	Commission = 7.5%
Sales <= 50,000	Commission = 5%

```
if (salesTotal > 100000)
            commission = .1;
if (salesTotal > 50000)
            commission = .075;
if (salesTotal <= 50000)
            commission = .05;

System.out.println(“rate = “ + commission);
```

- Commission for sales of 110,000?

A solution

- Order matters

```
if (salesTotal <= 50000)
        commission = .05;
if (salesTotal > 50000)
        commission = .075;
if (salesTotal > 100000)
        commission = .1;
```

- This will work but it is not an elegant solution
- For a very long commission list, all statements must be processed with commission possibly being reassigned multiple times

A better Solution: nested `if`

- The statement executed as a result of an `if` statement or `else` clause could be another if statement
- This skips all remaining tests after an alternative has been selected

```
if (salesTotal > 100000)
    commission = .1;
else
        if (salesTotal > 50000)
         commission = .075;
   else
         commission = .05;

System.out.println(“rate = “ + commission);
```

A Problem with nested `if`

- Say we want to look at students who have got distinction (ie a mark >=70 and <80) or less in the unit

```
if  (examMark >= 70.0)
        if (examMark < 80.0)
                  System.out.println(“Distinction”);
else
               System.out.println(“not to distinction standard”);
```

The “Dangling else” Problem

- The problem is ambiguity
- Indentation is irrelevant to the compiler
- When if statements are nested, Java matches each else clause with the nearest unmatched if
  - As if the if-else were in a block

```
if  (examMark >= 70.0)
{
    if (examMark < 80.0)
         System.out.print(“Distinction”);
    else
         System.out.println(“not to distinction standard”);
 }
```

The “Dangling `else`” Problem

- To be absolutely clear about your logic – use blocks with nested if’s when there is a dangling else

```
if  (examMark >= 70.0)
{
    if (examMark < 80.0)
       System.out.print(“Distinction”);
}
else
    System.out.println(“not to distinction standard”);
```

Comparing Floating Point Values

- We also have to be careful when comparing two floating point values (`float` **or** `double`) for equality
- This is because of the way they are stored in memory
- 0.33 or even 0.1 in binary can only be an approximation of the true value of the number since they cannot be represented exactly with powers of 2

- This is true of floating points in all languages, not just Java

```
public class TestFloat
{
  public static void main(String[] args)
  {
      double result;

      result = 9.8 - 9.7;

      if (result == 0.1)
          System.out.println("they are the same");
      else
          System.out.println(“result = “ + result);
   }
}
```

```
result = 0.10000000000000142
```

- Be aware when comparing a floating point number stored as a variable to a floating point literal

Cascaded `if` Statements

- When we test a series of nested conditions we can encounter indentation creep

```
 if (day == 1)
       System.out.println("Sunday");
else
        if (day == 2)
             System.out.println("Monday");
      else
               if (day == 3)
                      System.out.println("Tuesday");
             else
                       if (day == 4)
                           System.out.println("Wednesday");
                     else . . .
```

An accepted solution

- To avoid this you can put each `else` under the original `if`

```
if (day == 1)
      System.out.println("Sunday");
else if (day == 2)
      System.out.println("Monday");
else if (day == 3)
      System.out.println("Tuesday");
else if (day == 4)
      System.out.println("Wednesday");
else if (day == 5)
      System.out.println("Thursday");
else if (day == 6)
      System.out.println("Friday");
else if (day == 7)
      System.out.println("Saturday");
```

Comparing Strings for equality

- As a String is an *object* not a primitive type
  - Use the `equals(String object)` method to compare for equality

```
String myName = “Rumplestiltskin”;
String yourName = JOptionPane.showInputDialog(“Enter your name”);
if (yourName.equals(myName))
    System.out.println(“Our names are the same”);
else
    System.out.println(“Our names are different”);
```

- Also see the method equalsIgnoreCase(String object)

```
String answer = JOptionPane.showInputDialog(“Do you want to continue? (y/n): ”);
if (answer.equalsIgnoreCase(“y”))
```

The switch Statement

- A better means of comparing a variable (or an expression) against a set of possible values

- Matches the result to one of several possible *cases*

The switch Statement

- The expression eg *day* must result in a data type *char, byte, short* or *int*;
- it cannot be a floating point value

```
switch (day)
{
   case 1: System.out.println("Sunday");
   case 2: System.out.println("Monday");
   case 3: System.out.println("Tuesday");
   case 4: System.out.println("Wednesday");
   case 5: System.out.println("Thursday");
   case 6: System.out.println("Friday");
   case 7: System.out.println("Saturday");
 }
```

The Break Statement

- A match is a start point for execution
- Processing will continue into the following cases
- A *break statement* causes control to transfer to the end of the switch statement

What if there is no match?

- A switch statement can have an optional `default` case
- Control will transfer to it if no case value matches
- If there is no `default` case, and no other value matches, control falls through to the statement after the switch

The switch Statement

```
switch (day)
{
    case 1:  System.out.println("Sunday");
             break;
    case 2:  System.out.println("Monday");
             break;
    case 3:  System.out.println("Tuesday");
             break;
    case 4:  System.out.println("Wednesday");
             break;
    case 5:  System.out.println("Thursday");
             break;
    case 6:  System.out.println("Friday");
             break;
    case 7:  System.out.println("Saturday");
             break;
    default: System.out.println(“day “ + day + “ out of range”);
}
```

Logical Operators

- Consider this nested `if` in *structured English* (pseudocode)

```
if (the time is 6.30 am)
    if (it is a weekday)
          Get out of bed
```

- The same logic could also be expressed as:

```
if (the time is 6.30 am AND it is a weekday)
    Get out of bed
```

- We form complex expressions by combining conditions with
  - Logical AND
  - Logical OR

Logical Operators

- Boolean expressions (ie expressions evaluating to true or false) can use the following *logical operators* :

&&	Logical AND
||	Logical OR
!	Logical NOT

Logical AND

- The logical *and* expression

  a && b

  is true if both **a** and **b** are true, and false otherwise

- It can make your code less error-prone and more readable than nested `if`’s

$$\text{if (}\underbrace{\text{amount >= 1000}}\text{ \&\& }\underbrace{\text{amount < 2000}}\text{)}$$

Logical OR

- The logical *or* expression

```
a || b
```

is true if a or b or both are true, and false otherwise

$$\text{if } \underbrace{(\text{score} > 0}\ \ ||\ \underbrace{\text{count} > 10})$$

Truth Tables

- A truth table shows the possible true/false combinations
- Since && and || each have two operands, there are four possible combinations of true and false

a	b	a && b	a || b
true	true	true	true
true	false	false	true
false	true	false	true
false	false	false	false

Logical NOT

- Logical NOT is a unary operator (it precedes one operand)

if `a` is false, then `!a` is true

== reversed is !=

< reversed is >=

a	**!a**
true	**false**
false	**true**

Precedence revisited

- Logical operators have precedence relationships between themselves and other operators
- Always use brackets to clearly show your intention

Highest	* / %
	+ -
	> < >= <=
	== !=
	&&
	||
Lowest	=

Lecture Outcomes

Have you understood:

- formatting output
- the flow of control through a method
- decision-making statements
- operators for making complex decisions

- Questions?