4. Programming – Conditional Statements

***Note**: some example programs are from, or based on, examples from Java for Everyone (C Horstmann), the course text.*

All the programs we've seen so far do *one* thing – they don't make any *decisions.* This is not a property of most real programs – they need to make choices based on inputs and/or stored data, and do different things depending on the value(s) of those inputs (or that data). There are two basic ways of doing this in Java – *if statements,* and *switch statements.* You should know about both but in practice if statements are much more commonly used, and we'll do those first. There is some material later on switch statements – but you should make sure you fully understand (and can use) if statements before you go on to switch statements.

If Statements

The basic structure of the if statement is:

```
If (condition) {
     statements
}
```

Where *condition* is a Boolean expression (i.e. something that is either true or false and *statements* is a list of Java statements. If the condition is true, then the statements are executed; if the condition is false, then the statements are skipped. Here's are some actual examples:

```
if (x == 0) {
     System.out.println("x is zero");
}
```

Assuming `x` is an integer variable, the text "`x is zero`" is printed out if `x` has the value 0.

```
if (x > y) {
     int temp = x;
     x = y;
     y = temp;
}
```

This example is more complex – if `x` is greater than `y`, the code in the if statement swaps over the values of `x` and `y`. Notice that we've declared a new variable called temp inside the {..} after the if statement – this variable

can only be used between those {..} – this is the *scope* of the variable (and there's a chapter on it later).

Aside: Code is Executed In Order

The code above to swap two variables is a 'standard' example and something that you do in practice quite often. A common mistake is to not understand that you need the 'intermediate' variable temp – *code is executed in order*, so if we did not save the value in `x`, as soon as we execute the line `x = y`; the value stored in `x` is lost (but not for us because we saved it in `temp`).

Here's another more complex example:

```
if (language.equals("English")) {
     System.out.println("Hello!");
}
if (language.equals("Greek")) {
     System.out.println("γεια σας!");
}
```

Two important points here:

- We now have *two* if statements – first we test if the variable `language` contains the value "English" (and say "Hello" if it does); then we check if `language` contains the value "Greek" (and say "γεια σας" if it does).
- Secondly, we have used the funny notation `language.equals("Greek")` instead of just `language == Greek` – this is a very important point, and we'll explain in detail later. For now, just remember that whenever you want to check if two strings `A` and `B` are equal you need to write `A.equals(B)` – and not `A == B`

In this case you might spot something – if language was English then it obviously cannot also be Greek. So if the first test (`language.equals("English")`) is true, then the second one cannot be. We can join the two if statements together:

```
if (language.equals("English")) {
     System.out.println("Hello!");
} else if (language.equals("Greek")) {
     System.out.println("γεια σας!");
}
```

The word **`else`** (in bold only for visibility – it won't be bold in your code) *links* the two separate if statements together. Now the second condition will only be checked if the first one is false. (Notice also I've slightly changed the indenting – now the closing } appears on the same line as else. You can also write if statements using this style of indenting:

```
if (language.equals("English"))
{
     System.out.println("Hello!");
}
else if (language.equals("Greek"))
{
     System.out.println("γεια σας!");
}
```

Both are equally acceptable – I use the first style but you are free to choose either – just stick to the one you pick!)

You might wonder if we can add more conditions to this example – and we can:

```
if (language.equals("English")) {
     System.out.println("Hello!");
}
else if (language.equals("Greek")) {
     System.out.println("γεια σας!");
} else if (language.equals("Swedish")) {
     System.out.println("Hallå!");
}
```

In practice, a very common case we have to deal with is a default value – that is, if one or more conditions are not true, then the we do not need to check a final case because it must be true. For example:

```
if (x > 0) {
     System.out.println("positive");
} else if (x < 0) {
     System.out.println("negative");
} else if (x == 0) {
     System.out.println("zero");
}
```

But if x is not less than zero and not greater than zero, it must be zero – there is no need to check. We can simplify this to:

```
if (x > 0) {
     System.out.println("positive");
} else if (x < 0) {
     System.out.println("negative");
} else {
     System.out.println("zero");
}
```

by removing the final if test.

Summary

You can see that the if statement has a number of possible forms.

- A simple single test without an else case.
- A simple single test with an else case.
- Two or more tests without an else case.
- Two or more tests with an else case.

This allows us to create a very wide range of possible tests.

Here's a simple complete program based on the examples we've seen above.

```
import java.util.Scanner;

class SayHello {
     public static void main(String[] args) {
          Scanner in = new Scanner(System.in);

          System.out.print
               ("What language do you speak? ");
          String language = in.nextLine();
          if (language.equals("English")) {
               System.out.println("Hello!");
          }
          else if (language.equals("Greek")) {
               System.out.println("γεια σας!");
          } else if (language.equals("Swedish")) {
               System.out.println("Hallå!");
          } else {
               System.out.println
                  ("Sorry, don't know it…");
          }
     }
}
```

An Easy Mistake to Make – Be Careful

Here's something that you might accidentally do, particularly if you use this type of bracketing

```
if (language.equals("English"))
{
     System.out.println("Hello!");
}
```

That is, where the opening { is below the if instead of on the same line. Because most other Java statements end with a ";", you might accidentally write this:

```
if (language.equals("English"));//<- note the ";"
{
     System.out.println("Hello!");
}
```

The (wrong) ";" is in red. The trouble with this is that it's legal Java. The ";" on its own is an *empty statement* – **it does nothing**. So this is an if statement that does nothing if it's true. In this case, the code in the {…} is not related to the if and will always be executed – the fact that we've put it in the {...} does not stop that – so this code will *always* print "Hello!" regardless of the value of the variable `language`.

KEY POINT: No ; in If

Do not put a ";" at the end of an if statement

Strings and Equals Danger: Skip on First Reading if New to Programming

Above we said that you have to use a different way to test if two strings are equal in Java than simply using `==`. In this section we will explain why. In the case of variables of simple data types (`integers, doubles, Booleans`) Java stores the value in that variable in a particular space in memory. It can do this because the space in memory taken up by simple data types is always the same (e.g. an `int` always takes four bytes). The trouble with more complex data is that this is not always true. For example:
"lemon"
takes up less space than:
"time flies like an arrow, fruit flies like a banana."
Without moving other data around, Java cannot easily accommodate data that can be variable in size in the same parts of memory it stores other data.
So, instead of storing the actual data in a particular word in memory, it stores a *reference* (sometimes called a *pointer* or *memory address*) to some *other* location in memory where the actual data is stored:

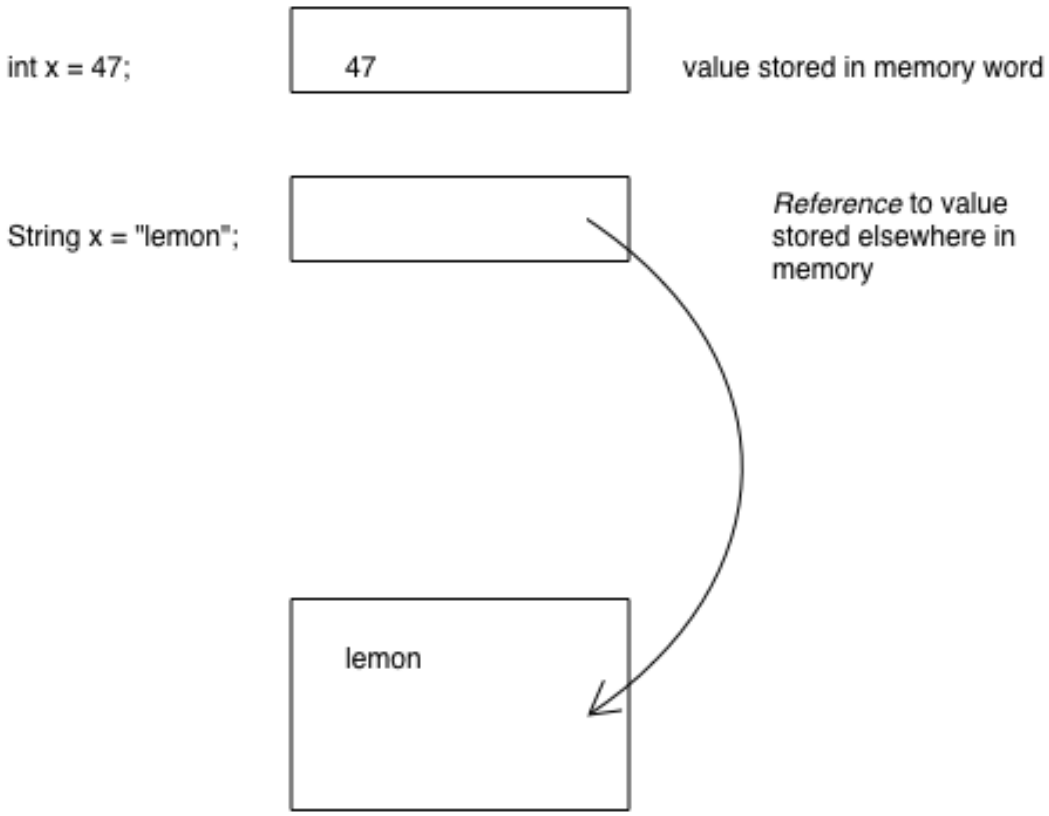

Figure 1: Storing simple and complex data

This means that if you use == to compare strings you are not comparing the actual data, but the value of the location they are stored in memory. This means that if the two strings have the same value (say they are both “lemon”) but are stored in different places in memory, Java will say they are not equal – and this is not usually what you want to happen. So the notation `A.equals(B)` is used to compare the actual values stored in the strings.
You might be wondering – why not just change the meaning of == so it works properly with Strings? The trouble is Strings are a particular case in Java of something called *objects*, and in practice making changes to == just for strings would mess up a lot of other behavior around == that currently works perfectly well. So in practice we just have to remember not to use == for Strings.

Strings and Equals More Danger: Skip on First Reading if New to Programming

Unfortunately, things are (potentially) a bit worse than this. It would not be so bad if every time we used == to compare Strings by mistake (and we do – and you probably will too) then our programs did not work properly. We would almost certainly find this out in testing. The trouble is that often our programs do work with == and Strings even though it’s wrong!
The reason for this is that Java, for efficiency reasons, keeps track of what is in Strings and, if you create one that is the same as another one you already

have, instead of making a complete new one, Java just reuses the one you already have:

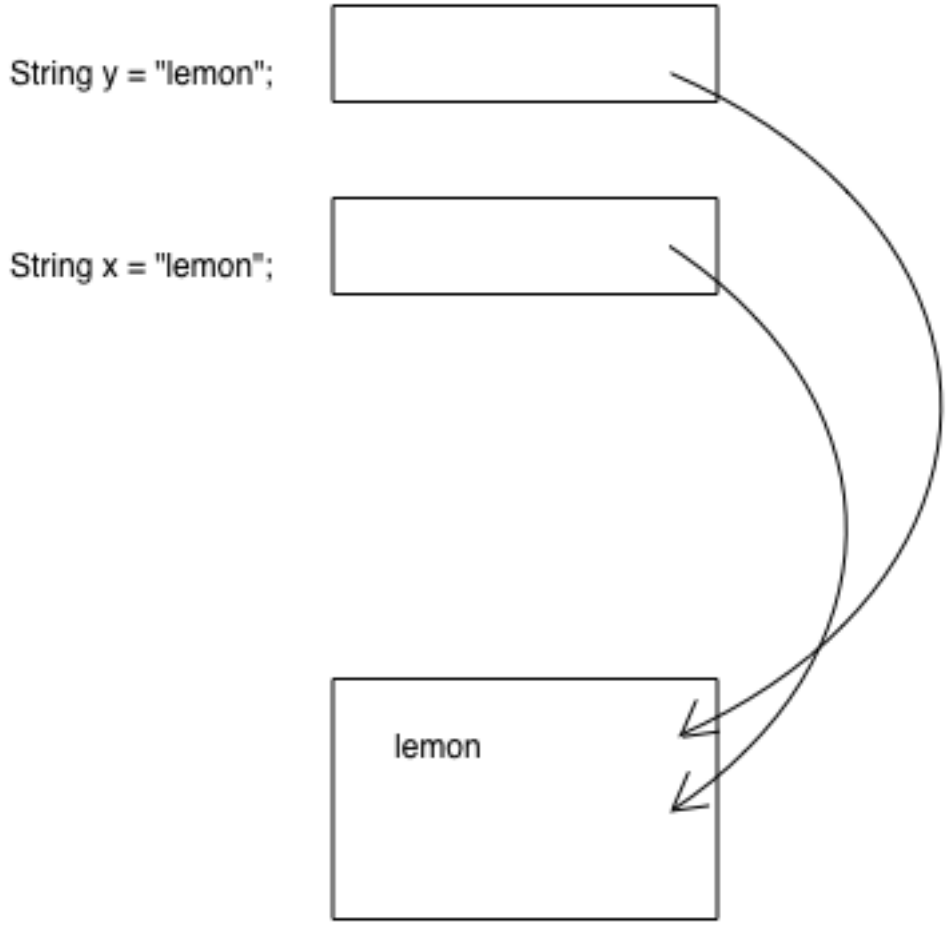

Fig 2: Sharing Strings

(In case you are wondering: "what happens if I change one of the strings?" the answer is *you can't* – Strings in Java are *immutable* and cannot be changed. When you *think* you are changing one you are *really* making a new one.)

Because Fig 2 means that `x` and `y` 'point' to the same bit of memory, they *are* equal and so if you check if **`x == y`** you will find that it's true.

So why worry then? If this is true, then surely there's no problem? But the problem is code like this:

```
String fruit = "Banana";
String dessert = "Banana Split";

String devious = dessert.substring(0,6);

if (fruit == devious) { // Should be true but isn't…
     System.out.println("Fruit & Dessert...");
}
```

What we've done here is create two strings – "Banana" and "Banana Split" which are not equal (so Java cannot 'share' them), and then extract a substring from "Banana Split" consisting of the first six letters (which is of course "Banana" – that's what `dessert.substring(0,6)` does). Now the value of the strings is the same ("Banana") but they are not 'shared' by Java – so == does not work.

Bottom Line: Always use `A.equals(B)` when comparing Strings in Java (and don't get too cross with yourself if you forget occasionally – everyone does.)

Conditions and Booleans

The tests we use in if statements are called *conditions* and they evaluate to either *true* or *false.* Previously we say Boolean variables which also have the value *true* or *false.* Are these related? Obviously they are. Remember this example from above:

```
if (x > 0) {
     System.out.println("positive");
} else if (x < 0) {
     System.out.println("negative");
} else {
     System.out.println("zero");
}
```

We can rewrite this as follows:

```
boolean isPositive = x > 0;
boolean isNegative = x < 0;
if (isPositive) {
     System.out.println("positive");
} else if (isNegative) {
     System.out.println("negative");
} else {
     System.out.println("zero");
}
```

Instead of directly using the actual conditions we have created two Boolean variables and assigned the values of the tests to them. Now in this case it doesn't seem obviously a good idea to do this (and it isn't – it makes it longer but not any easier to read; maybe worse). But there *are* cases when we are dealing with more complex conditions (we'll look at this next) where it can make the code more readable and easier to understand if we use some Boolean variables. Notice we've started the names of the variables with "`is`" – this is a common thing to do with Boolean variables and good practice. The reason is that something either "is" or "isn't" – for example, in the case above, the value of `isPositive` is true if x actually is positive.

Complex Conditions Part 1: Logical Operators

The tests we've done so far are very simple – in each case we are testing for just one thing. However, in practice, we are often faced with cases where we have to test for more than one thing. For example:
"You get a discount of 10% when buying more than 5Kg of any fruit except for lemons"
You need to check that both conditions are true before giving the discount. One way to do this is to use the Boolean operators from chapter 3.:

```
if ((weight > 5) && !fruit.equals("lemon")) {
     price *= 0.9;
}
```

This example uses two Boolean operators:

- `&&` - to check that the `weight` is > 5 (assume units are Kg) and that the second condition is also true; which is…
- `!` – to negate the test that the fruit are lemons - `!fruit.equals("lemon")` is true only if fruit does not equal "lemon".
- (Notice I've bracketed `weight > 5` – you do not have to do this but I think it helps readability.)

We can also use the other Boolean operators – for example:
"Shipping costs are 10% higher for peaches, strawberries and raspberries"

We can express this like this:

```
if (fruit.equals("peach") ||
          fruit.equals("strawberries") ||
          fruit.equals("raspberry")) {
     shipping *= 1.10;
}
```

This is now getting a bit long and complicated – what if there were even *more* parts to the condition? It could become unreadable. This is one of those cases where, as terms become more complex, you might think about breaking them up and introducing some Boolean variables. For example:
"There's a 20% discount on Kiwis if you buy more than 10kg, or if they are within 1 day of their 'sell by' date"

This becomes:

```
if (fruit.equals("kiwi") &&
          ((weight > 10.0) || (daysToSellBy < 1.0))) {
     shipping *= 0.8;
}
```

This is now getting pretty tricky to follow, with lots or brackets (some of which we need to ensure that the or operations are performed before the and operation). Maybe this is better:

```
boolean isDiscountable = ((weight > 10.0) ||
                          (daysToSellBuy < 1.0));
if (fruit.equals("kiwi") && isDiscountable) {
     shipping *= 0.8;
}
```

We have combined a Boolean variable and a complex condition to try to break the logic into more readable parts.

Dangerous Issue – Skip of First Reading if New to Programming

Here's something to worry about (a bit). Suppose you want to calculate and print the average of a group of numbers, but only if the average is above some threshold – you've already calculated the total of the numbers (`total`); you have the count of the numbers too (`count`); and you know the threshold above which you will print the average (`threshold`). But you are aware of the possibility of the count of numbers being zero, and want to avoid a *divide by zero error*. You might write something like this:

```
if ((count != 0) && (total / count > threshold)) {
     System.out.println("Average is " + total/count);
}
```

This will work in Java but not in lots of other languages. The reason is a different approach to the logic.

- Given an expression like `A && B`, or `A || B`, Java guarantees to evaluate `A` first and then `B`.
- Furthermore, if in the case of `A && B` the value of `A` is false, it does not bother evaluating `B` – because `A && B` cannot possibly be true if `A` is false. Similarly, in the case of `A || B`, if `A` is true it does not bother evaluating `B` because in that case `A || B` cannot possibly be false.

This means, in Java, if count is zero in the case above, it does not try to evaluate `total/count > threshold` – and just as well because `total/count` would be a *divide by zero error*.

There are other languages though that do not do this *– they will evaluate the whole expression (and do not guarantee to do them in the order you write them either).* Neither approach is 'good' or 'bad' – you just need to be aware that languages differ, particularly when you move on from Java.

(How do we handle the example above in such languages? The answer is in the next section.)

Complex Conditions Part 1: Nested If Statements

You can use Boolean operators - &&, ll, ! etc. – to create conditions that are as complex as you want – but you probably don't want to because they can

get hard to read. For example, consider *leap years.* The common perception is that leap years happen every four years – but this is not quite true. The actual rules are:

- Years that are divisible by 400 **are** leap years.
- Years that are divisible by 100 but not 400 **are not** leap years
- All other years divisible by four not covered by the two cases above **are** leap years

As you can see, it's a bit more complex – so here's some code to tell if a year is a leap year or not:

```
if ((year % 4 == 0) && (year % 100 != 0) ||
          (year % 400 == 0)) {
     System.out.println("Is a leap year");
} else {
     System.out.println("Is not a leap year");
}
```

This works but it's a bit 'dense' and hard to read. We might want to break it up a bit to make it clearer. One way to do that would be to use some Boolean variables; another would be to use *nested* if statements – one if statement inside another.

```
//All leap years must be divisible by 4…
if (year % 4 == 0) {
     //Ones divisible by 400 are leap years…
     if (year % 400 == 0) {
          System.out.println("Is a leap year");
     //But ones divisible by 100 and not 400 are not
     } else if (year % 100 == 0) {
          System.out.println("Is not a leap year");
     //All others divisible by 4 are leap years
     } else {
          System.out.println("Is a leap year");
     }
//Anything not divisible by 4 is not a leap year
} else {
     System.out.println("Is not a leap year");
}
```

We've broken the logic into parts, and that's allowed us to add comments to help explain it – in the previous case it would have been harder because the comments could not go 'within' the logic. In this particular example, one thing I don't like is the repeated messages – we have to put each one in twice which violates the DRY (*don't repeat yourself*) principle. But we can fix that:

```
final String NOT_LEAP_YEAR = “Is not a leap year”;
final String LEAP_YEAR = “Is a leap year”;

//All leap years must be divisible by 4…
if (year % 4 == 0) {
     //Ones divisible by 400 are leap years…
     if (year % 400 == 0) {
          System.out.println (LEAP_YEAR);
     //But ones divisible by 100 and not 400 are not
     } else if (year % 100 == 0) {
          System.out.println (NOT_LEAP_YEAR);
     //All others divisible by 4 are leap years
     } else {
          System.out.println (LEAP_YEAR);
     }
//Anything not divisible by 4 is not a leap year
} else {
     System.out.println(NOT_LEAP_YEAR);
}
```

In this particular case, this might all be going a bit far – the nested code is probably easier to understand for new programmers in this case; but the first example is not *that* hard. But there are examples where simply trying to build one large condition would be very complex and hard to understand.

KEY POINT: Using Boolean Logic, Boolean Variables and Nesting for Complex Conditions

If you have complex conditions that need to be evaluated for if statements, consider Boolean logic, Boolean variables, or nested if statements – of a combination of these. What should you choose in each case? That really depends on the example and your judgment – you will get better at judging what’s the best option as you gain experience.

Dangerous Issue Solution – Skip of First Reading if New to Programming

It should now be obvious how you solve the problem we showed above with a possible divide by zero error in languages other than Java – you would write:

```
if (count != 0) {
     if (total / count > threshold) {
          System.out.println(“Average is “ +
                total/count);
     }
}
```

Dangerous Thing – Don’t Do It!

If you read text books on Java programming they will tell you that you can write an if statement like this:

```
if (condition)
     statement
```

where ‘statement’ is a single statement – notice we’ve left out the {..}. *But you must never do this!* Here’s why. Suppose you start off with code like this:

```
if (fruit.equals(“Lemon”))
     System.out.println(“My favourite!”);
else
     System.out.println(“Not as good as Lemons…”);
```

This is syntactically correct and works – but if you change it to:

```
if (fruit.equals(“Lemon”))
     System.out.println(“My favourite!”);
     System.out.println(“Better than oranges.”);
else
     System.out.println(“Not as good as Lemons…”);
```

It won’t compile – because Java does not recognize that the two statements

```
     System.out.println(“My favourite!”);
     System.out.println(“Better than oranges.”)
```

are grouped together – you’ll get something like this:

```
Fruit.java:9: error: 'else' without 'if'
Else
```

To fix it you need to bracket the two lines after the if:

```
if (fruit.equals(“Lemon”)) {
     System.out.println(“My favourite!”);
     System.out.println(“Better than oranges.”);
} else
     System.out.println(“Not as good as Lemons…”);
```

You may think this is not so bad – the compile catches the error. But if you don’t have an else but wrote this:

```
if (fruit.equals(“Lemon”))
     System.out.println(“My favourite!”);
```

and then changed it to this:

```
if (fruit.equals(“Lemon”))
     System.out.println(“My favourite!”);
     System.out.println(“Better than oranges.”);
```

You don't get the intended behavior *even though it compiles* – Java will execute the second line and print "Better than oranges" *regardless* of the value of `fruit`. In this case the indenting is incorrect and misleading us – this code should really be written:

```
if (fruit.equals("Lemon"))
     System.out.println("My favourite!");
System.out.println("Better than oranges.");
```

KEY POINT: Always put the {…} in If statements

Even if you only have one line in your if statement, put the {…} in – you will be thankful if (when!) you later change your code. So our example above should correctly be written:

```
if (fruit.equals("Lemon")) {
     System.out.println("My favourite!");
     System.out.println("Better than oranges.");
} else {
     System.out.println("Not as good as Lemons…");
}
```

Newer languages – e.g. Apple's Swift – are beginning to insist that the {…} are present in if statements to eliminate this problem.

KEY POINT: Indenting is for You, not the Compiler

As we stated in Chapter 1, Indenting is to make code readable – but the compiler does not pay any attention to it. Just laying out your code in the way you want it to work without grouping things together with the {…} will not work.

Controversial Formatting – Avoid

Another thing you will sometimes see in code is that short if statements can be written on one line. E.g.

```
if (year % 4 == 0)) {System.out.println("Olympics!");}
```

or even:

```
if (year % 4 == 0)) System.out.println("Olympics!");
```

(but remember what we said about always including the {..}!)

Some like this but I'm not keen and think it's often not clear, so would prefer you avoided it.

Switch Statements

If statements are very useful and the main way of doing conditionals. But if there are *lots* of *simple* cases, they can get very tedious. For example, suppose you have a string representing the value of a *playing card* and you want to do something different for each possibility:

```
if (card.equals("Ace")) {
     //do something
} else if (card.equals("King")) {
     //do something else
} …
…
} else if (card.equals("one")) {
     //do one last thing…
}
```

(Note the comments are place holders for real code which I haven't actually written for this example). This would be a long a tedious piece of code – and in *simple* but long cases like this, we can use a switch statement

```
switch(card) {
     case "Ace" :
          //do something
          break;
     case "King":
          //do something else
          break;
     …
     case "one":
          //do one last thing
          break;
}
```

Here's a real complete example:

```
//What colour are “major” London Underground lines on the
normal map:
String lineColour;
switch (lineName) {
            case "Bakerloo":
                lineColour = "brown";
                break;
            case "Central":
                lineColour = "orange";
                break;
            case "Circle":
                lineColour = "yellow";
                break;
            case "District":
                lineColour = "green";
                break;
            case "Jubilee":
                lineColour = "grey";
                break;
            case "Metropolitan":
                lineColour = "purple";
                break;
            case "Northern":
                lineColour = "black";
                break;
            case "Picadilly":
                lineColour = "dark blue";
                break;
            case "Victoria":
                lineColour = "light blue";
                break;
        }
```

A few important points about switch statements.

Notice the “break” statements: After the code for each case, there is a break statement. This causes execution of the program to skip to the end of the case statement (the final }). Without the break, execution would continue with the next case which is not usually what we want. For example, if we had, in error:

```
case “District”:
     lineColour = “green”;
case “Jubilee”:
     lineColour = “grey”;
     break;
```

then `lineColour` would always end up set to grey because the initial assignment to green if it was the District line would be overwritten (more on this later).

Only test for equality: You can only test if something equals something else in a switch statement – not more complex tests (less than, greater than or equal, combinations of conditions).

Indenting slightly different: Unlike most cases, we don't indent the code within the switch statement – each word '`case`' appears directly under '`switch`' (at least we usually don't – you can if you want to).
Mainly simple types: For the most part, switch statements can only be applied to simple data types. That meant until very recently that in Java you could only do things like `int` and `char` (characters) – but because switching on strings was such a popular request it was added. But you won't be able to use them with the more complex things we'll get to at the end of the module.
Default case: In the example above, if the variable `lineName` does not equal one of the cases, `lineColour` will not be set to anything. And, for that reason, the code above would not compile – Java insists there is no possible way a variable could end up unassigned and generates an error if it thinks it's possible that could happen (warning this can be annoying). One way to fix it would be to assign lineColour a value initially:

```
String lineColour = "Error!";
switch(lineName) {
..
```

but it's very common to want to have an "everything else" value – which we can do with `default`:

```
case "Victoria":
     lineColour = "light blue";
     break;
default:
     lineColour = "Error!";
     break;
}
```

Final break: Notice that in both the example of default above, and the earlier complete example that ended with Victoria, we have a `break` statement at the end even though this is the last possible case, and execution will automatically go to the end of the case statement anyway. In this case you can leave the `break` out - ***but don't!*** That's because you are very likely to want to add additional cases later and forget to put in the extra breaks. For example, suppose you wrote the first version of the Underground switch statement leaving off the final `break`:

```
case "Victoria":
     lineColour = "light blue";
}
```

Now we compile it and there are errors because `lineColour` might not get assigned a value. So we fix it:

```
case "Victoria":
     lineColour = "light blue";
default:
```

```
      lineColour = “Error!”;
}
```

And… forget to put the break in – *so now it’s wrong*.

KEY POINT: Breaks in Switch

It’s always good practice to put in the final break statement even though it’s redundant - to avoid future errors.

Why the Break? – Skip at first Reading if New to Programming

You might ask why we have to put the break in, and why it does not happen automatically? This is a good question – and some languages (e.g. Apple’s new Swift again) do this. The answer is that you can come up with examples where it’s easier and shorter to write code using switch that does not need the break statement – but it’s pretty rare (I’ve never written one except to show it’s possible) and often quite hard to read (not to mention it’s likely you will make mistakes if you try to change it). So, on balance, it’s probably a failing in language design to require it – and newer languages (as well as Swift) may well drop it.

Multiple Cases

Slightly related to the bit about why you need the break statement, and slightly more useful and safer – you can have more than one case statement per ‘piece’ of code:

```
int daysInMonth;
switch(month) {
case “February”:
      daysInMonth = 28;
      break;
case “April”:
case “June”:
case “September”:
case “November”:
      daysInMonth = 30;
      break;
default: //If sure month can’t be an illegal value
      daysInMonth = 31;
      break;
}
```

Ternary Operator – Skip on First Reading if New to Programming

There is another way of doing conditionals in Java that is sometimes handy – it’s also controversial though and some people think the potential hazards are

outweighed by the advantages. I don't have strong views about this, though it's not something I use a lot.
The *ternary* operator is just that – an *operator* which returns a *value* just like addition, subtraction etc. The syntax is:

```
(condition)?true value: false value
```

If the condition is true, the value of the operator is the true value (between "`?`" and "`:`") – if false, it's the false value (after the "`:`"). Here's an example:

```
int absoluteValue = (value >= 0)?value : -value;
```

If `value` is greater than or equal to zero, then `absoluteValue` is set to `value`; if negative then to –`value` (i.e. to the absolute value). If we did this with an if statement it would be:

```
int absoluteValue;
if (value >= 0) {
     absoluteValue = value;
} else {
     absoluteValue = -value;
}
```

So the ternary operator is shorter and arguably clearer. In practice, this kind of thing does not come up that much (so it's fairly rare that it's something you can choose to do) and because it's an operator you can combine it with other arithmetic and logic operators, and the code quickly becomes very 'dense' and hard to read. Also in this specific case the correct way to do it is to call the math function:

```
int absoluteValue = Math.abs(value);
```

But it is an option and can sometimes be useful – just don't go mad.