

Introduction to Java

Overview

- ▶ Java is a platform as well as a language
- ▶ The Java language was developed by Sun Microsystems and first released in 1995
- ▶ The Java platform allows software to be developed and used across different architectures and operating systems



The editions



- ▶ **Java Micro Edition (ME)** – designed for running Java applications on mobile devices with limited resources
- ▶ **Java Standard Edition (SE)** – the general purpose version for desktop PCs and servers
- ▶ **Java Enterprise Edition (EE)** – SE plus some additional APIs for large enterprise server applications

The platform

▶ **Java Runtime Environment (JRE)**

- This a virtual machine which runs programs which have been compiled
- Contains a large library of classes for lots of different purposes

▶ **Java Development Kit (JDK)**

- Contains tools such the compiler
- Has a copy of the JRE

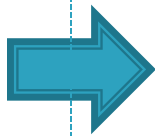
A C program...

Source file(s)

Compiler

Machine code

```
int  
main()  
{  
}
```



```
0101110  
0010110  
1101010  
1010101
```



For example...

Test.c

gcc.exe

Text.exe
containing
x86 instructions

Runs on an x86
processor

A Java program...

Source file(s)

Compiler

Java byte code

Virtual machine(s)

```
class  
Test {  
  ...  
}
```



```
store x  
pop y  
load z
```



For example...

Test.java

javac.exe

Test.class

Virtual machines

Class file(s)

Virtual machine(s)

Machine code

```
store x  
pop y  
load z
```

Test.class

Windows
x86 JVM

Symbian
RISC JVM

```
0101110  
0010110  
1101010  
1010101
```

```
0101110  
0010110  
1101010  
1010101
```



Example

```
class Kigali {  
    public static void main(String[] args){  
        System.out.println("Hello Kigali!");  
    }  
}
```

```
$ javac Kigali.java
```



Kigali.class

```
$ java Kigali
```



```
$ Hello Kigali
```

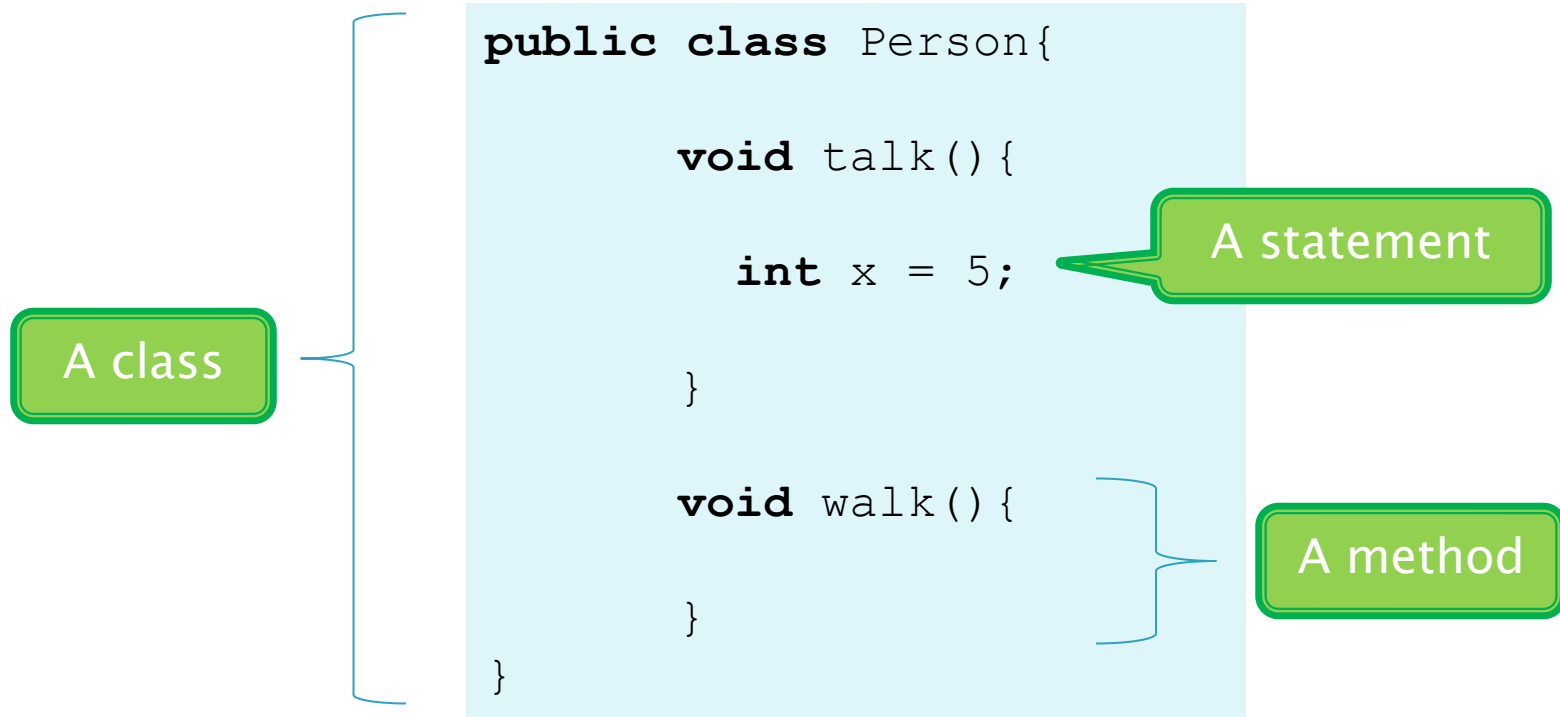
Write the
source file

Compile the
class file

Run the class
file in the
virtual
machine

View the
output

A Java source file



`public`, `class`, `void` and `int` are all Java keywords

More detail...

Access modifier

No return
value

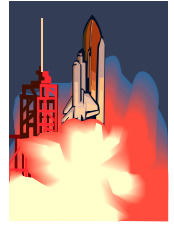
Method
name

A method
argument

```
public class Kigali{  
    public static void main(String[] args){  
        System.out.println("Whats up Kigali!");  
    }  
}
```

Output text to
the console

Launch time



- ▶ Calling the `java` program launches the JVM
- ▶ You specify the name of the class and any arguments to send it, e.g.

```
$ java Kigali "Hello" 4
```

- ▶ The JVM searches the class for a method called `main`, and then calls that method
- ▶ It sends the arguments (e.g. `"Hello"` and `"4"`) as items in the array called `args`

Write your first class!

- ▶ Create a **.java** file

- Define a class (should have the same name as the Java file)
- Add a the main method

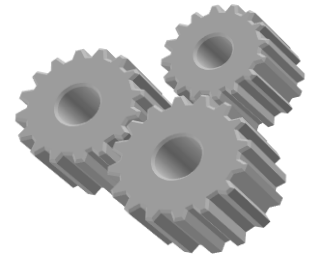
```
public static void main(String[] args) { ... }
```

- Call `System.out.println(...)` to print something

- ▶ Compile with **javac**

- ▶ Run the file with **java**

The compiler



- ▶ Converts the .java files to Java **bytecode**
- ▶ Reports any errors that prevented it from completing, or warnings that the developer should consider
 - Tells the developer what is wrong
 - Gives the source file and line number

Statements



- ▶ A statement does something, e.g.
 - Declaring a variable, `int x;`
 - Assigning a value to a variable, `x = 10;`
 - Incrementing a variable, `x++;`
 - Calling a method, `System.out.println("X");`
- ▶ Statements are separated by semi-colons, e.g.
 - `x = 10; y = x;`

Expressions



- ▶ An expression evaluates to a value, e.g.

◦ $10 + 2 * 3 / 4$	➡	9
◦ "Hello" + " world"	➡	"Hello world"
◦ 3	➡	3
◦ "kind of.."	➡	"kind of.."

- ▶ A statement can be an expression if it evaluates to a value, e.g.

◦ $x = 10$	➡	10	e.g. $y = (x = 10);$
◦ <code>Math.sin(5)</code>	➡	-0.9589...	

Variables



- ▶ A value is stored in a variable so that it can be used elsewhere in a program, e.g.

```
int x = 10;
```

Variable type

Variable name

Initial value
(optional)

- ▶ Variables can be **primitive types** or **object references**

Primitive types



► These are the types which are part of the Java language

- `boolean` true or false (1 bit)
- `byte` a 8bit signed number
- `char` a 16bit Unicode character
- `short` a 16bit signed number
- `int` a 32bit signed number
- `long` a 64bit signed number
- `float` a 32bit floating-point number
- `double` a 64bit floating-point number

Primitive types



- ▶ You can generally assign the value of a smaller primitive type to a larger one, e.g.

```
short big = 5646;  
int bigger = big;  
long biggest = bigger;
```

- ▶ But not the other way around

```
long big = 3453434623426;  
int notSoBig = big;  
short evenSmaller = notSoBig;
```

Compiler error!

Casting primitives



- ▶ To assign a value to a smaller type, you have to use the cast operator, e.g.

```
long big = 3453;
```

```
int notSoBig1 = big;
```

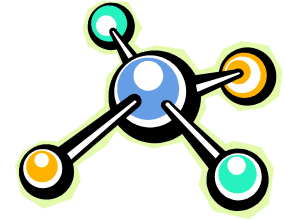
Compiler error!

```
int notSoBig2 = (int)big;
```

```
short smaller = (short)big;
```

Works

Objects



- ▶ These are complex types which are defined in the JDK or in your code, e.g.
 - `String` – a sequence of characters
 - `Date` – a date and time value
- ▶ An object is created in memory using the **new** keyword, e.g.

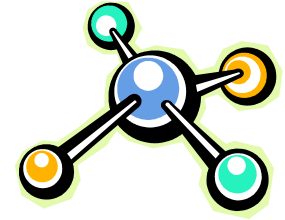
```
String s1 = new String("Hello");
```

The object's
class

Reference
variable

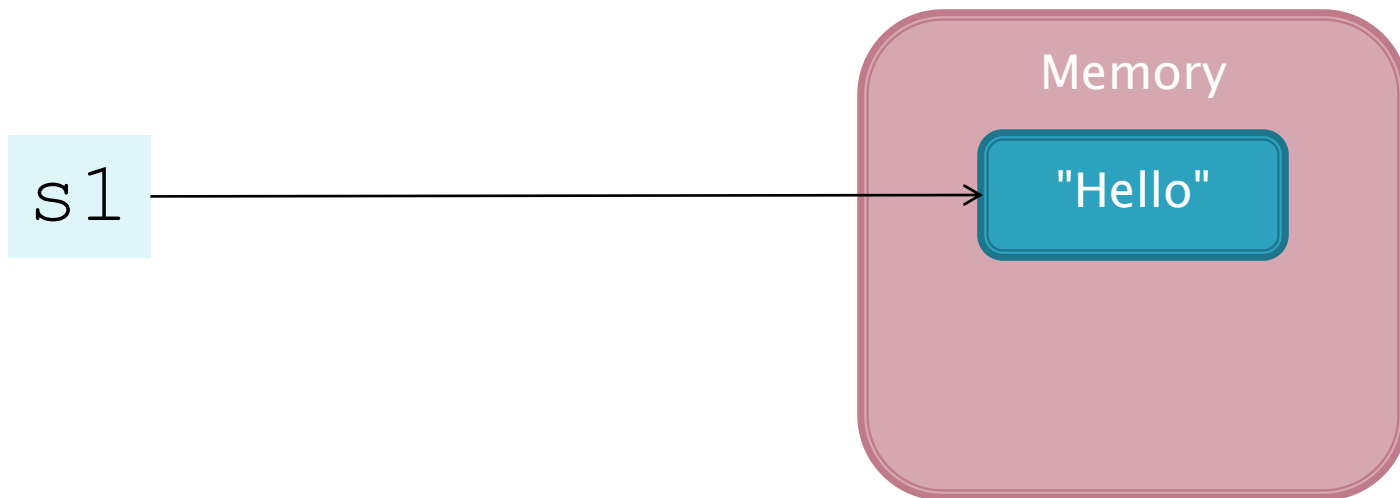
The object
itself

Objects

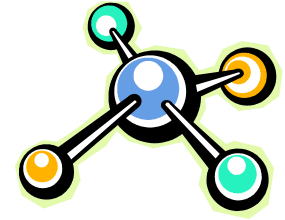


- ▶ Variable references aren't objects themselves
- ▶ They reference an object in memory

```
String s1 = new String("Hello");
```



Objects



- ▶ Reference variables can be null which means they don't reference an object anymore, e.g.

```
s1 = null;
```

s1

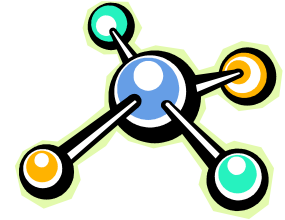
Variable
points to
nothing...

Memory

"Hello"

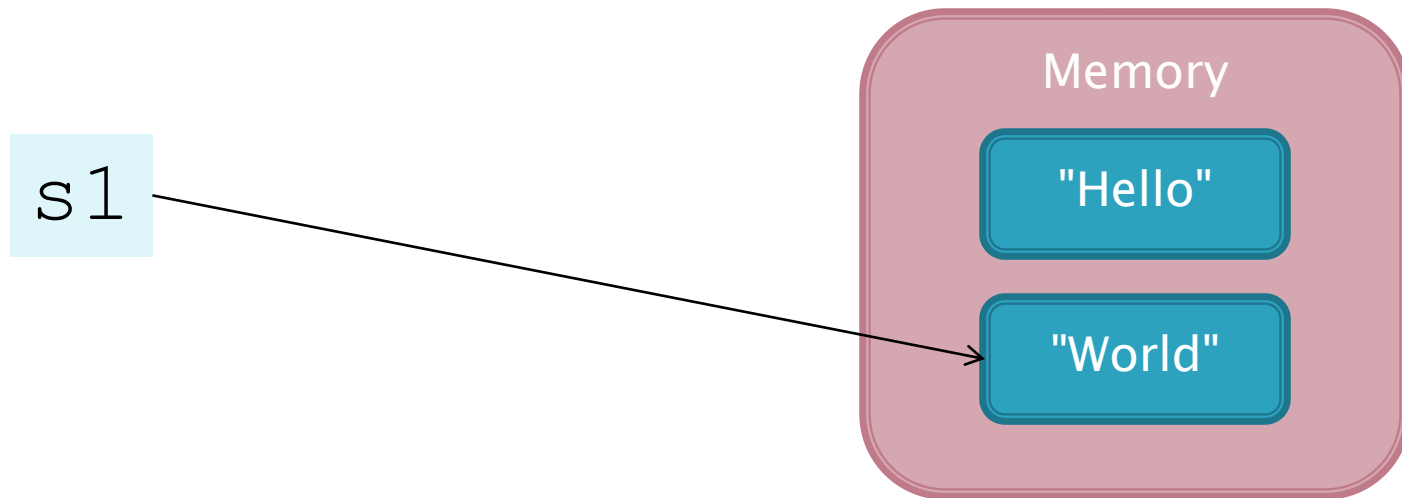
Object still
exists in
memory

Objects

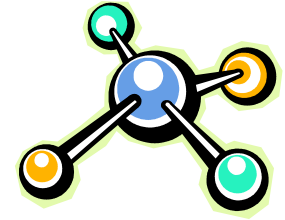


- ▶ A variable can be changed to reference a different object, e.g.

```
s1 = new String("World");
```

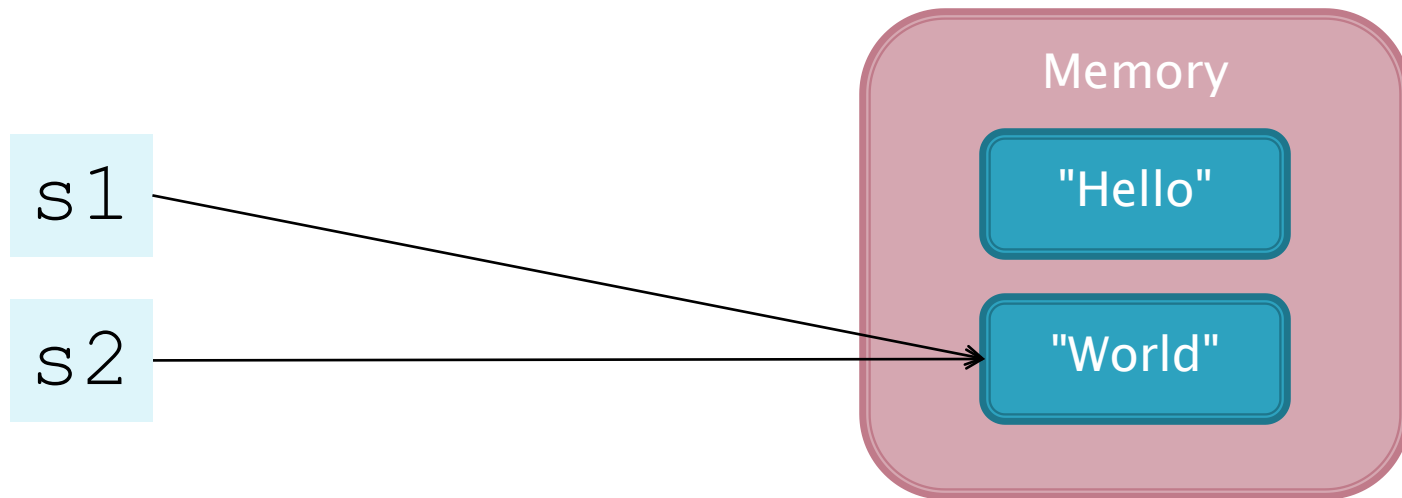


Objects



- ▶ And more than one variable can reference the same object, e.g.

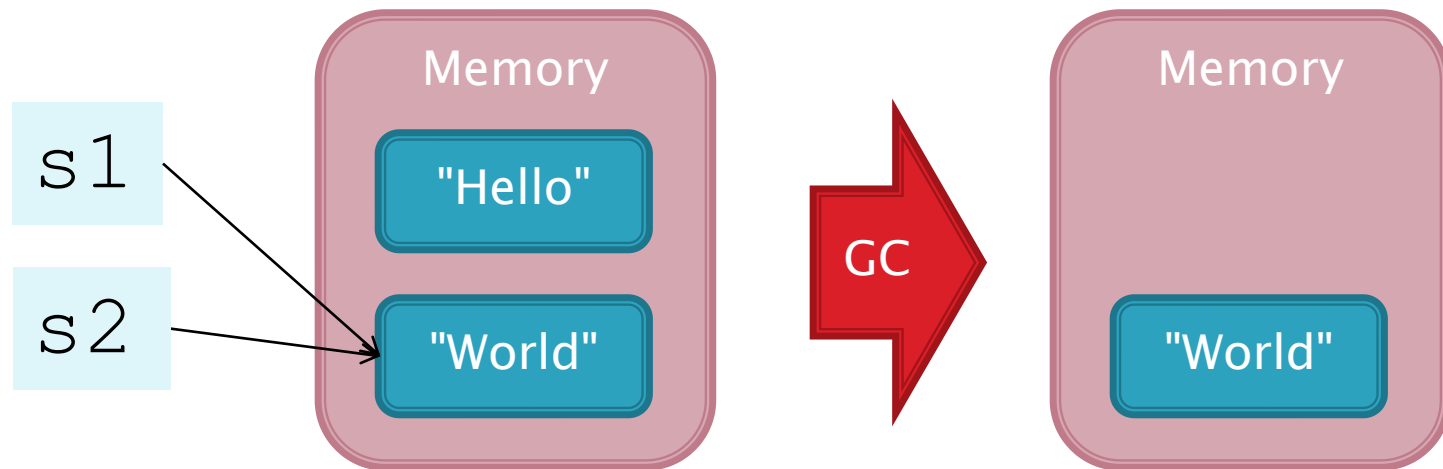
```
String s2 = s1;
```



Garbage collection



- ▶ This is process which runs in the background looking for objects with no references
- ▶ It deletes such objects from memory to free space for new objects



Equality



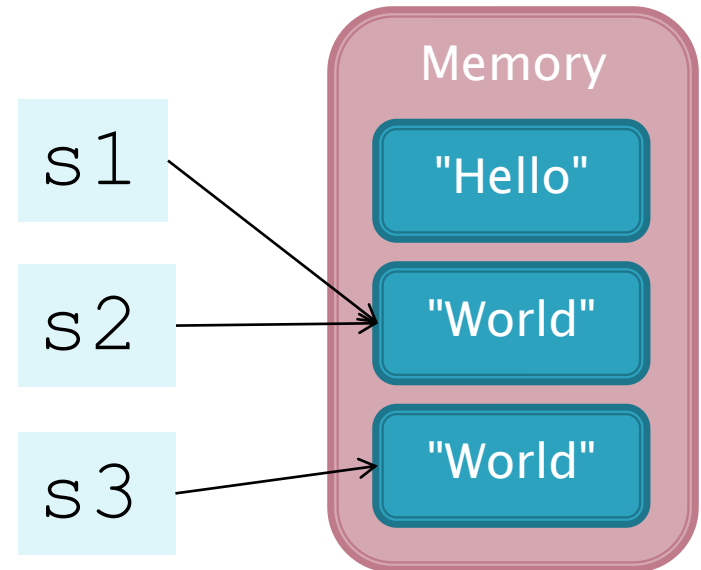
- ▶ Two types of equality...
- ▶ **Reference equality** `x == y`
 - Checks if the variables reference the same object in memory
- ▶ **Object equality** `x.equals(y)`
 - Checks if the objects which the variables reference are equal, i.e. have the same meaning or content
 - E.g. for Strings – do they have the same characters?

Equality example



```
String s3 = new String("World");
```

- ▶ `s1` and `s2` reference the same object so `s1 == s2` is **TRUE**
- ▶ `s1` and `s3` reference different objects so `s1 == s3` is **FALSE** even though the strings are the same



Equality example



- ▶ `s1` and `s3` reference `String` objects with the **same content** so `s1.equals(s3)` is **TRUE**

- ▶ Summary...

`s1 == s2`

`s1 == s3`

`s1.equals(s2)`

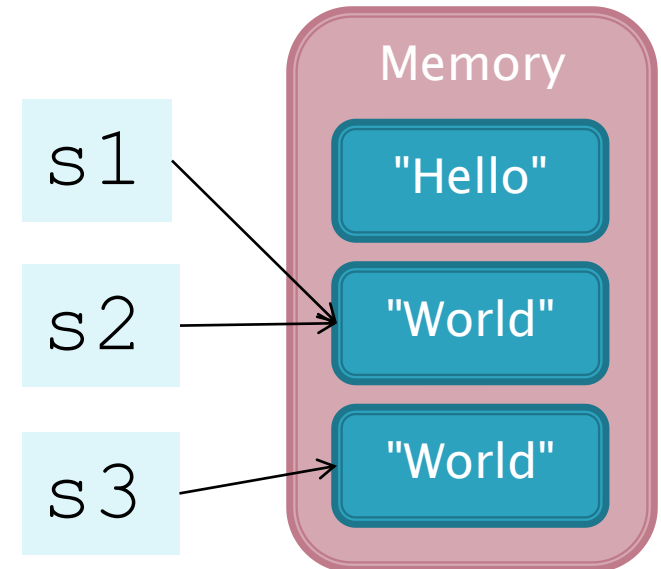
`s1.equals(s3)`

TRUE

FALSE

TRUE

TRUE



Strings are an exception...

- ▶ Strings can be created in two ways...

```
String s1 = new String("Hello");
```

As a normal
object in memory

```
String s2 = "Hello";
```

```
String s3 = "Hello";
```

```
String s4 = "World";
```

As literals which go
into the "String pool"

- ▶ `s1` and `s2` will reference different objects so
`s1 != s2`
- ▶ But `s2` and `s3` will point to the same object so
`s2 == s3`
- ▶ `s2` and `s4` will not so `s2 != s4`

Variable terminology

```
public class SuperApp {  
  
    int count = 0;  
  
    void setCount(int c) {  
        count = c;  
    }  
  
    void print() {  
        String s = "Val = " + count;  
  
        System.out.println(s);  
    }  
}
```

Instance variable
– it's available
anywhere in the class
instance

Method parameter
– it's available only in
the method

Local variable
– it's available only in
the method

Variable scope

```
public class SuperApp {  
  
    int a = 10;  
  
    SuperApp {  
        foo(a);  
    }  
  
    void foo(int b) {  
        int c = b;  
  
        for (int d = 0; d < c; d++) {  
            int e = d;  
  
            System.out.println(e);  
        }  
    }  
}
```

a



b



The scope of a variable is where it can be accessed in your code

Naming conventions



- ▶ The compiler won't complain if you don't stick to these, but we will!

Entity	Convention	Examples
Class name	Camelcase, starts uppercase	HelloWorldProgram PatientViewPage
Method name	Camelcase, starts lowercase	getPatientCount main
Variable	Camelcase, starts lowercase	numPatients listOfUsers
Constant	Uppercase with underscores	MAX_PATIENT_AGE DEFAULT_USER

Flow control: if-else

```
if (condition) {  
  
}  
else if (another condition) {  
  
}  
else {  
  
}
```

```
if (a == b) {  
    doMethod();  
}  
else if (isJava()) {  
    a = 10;  
}  
else {  
    out.println("X");  
}
```

```
if (a < x) {  
    a = 5;  
}
```



```
if (a < x)  
    a = 5;
```

If only one statement follows the if or else if, then the braces aren't needed

Flow control: while / do-while

```
while (condition) {  
    // loop these statements  
}
```

```
while (a < 10) {  
    out.println(a);  
    a++;  
}
```

```
do {  
    // loop these statements  
}  
while (condition);
```

do while means that the statements will always be executed at least once

Flow control: for

```
for (statement; condition; statement) {  
    // loop these statements  
}
```

```
for (int i = 0; i < 10; i++) {  
    System.out.println(i);  
}
```



```
int i = 0 ;  
while (i < 10) {  
    System.out.println(i);  
  
    i++;  
}
```

for is a alternative way
to write a while loop

Flow control: continue/break

```
for (int i = 0; i < 10; i++) {
```

```
    if (i == x)
```

```
        break;
```

```
    else if (i == z)
```

```
        continue;
```

```
    System.out.println(i);
```

```
}
```

break causes the for loop to finish immediately

continue goes to the next iteration if there will be one

Flow control: switch

```
switch (variable) {  
  case <value1>:  
    statements  
    break;  
  case <value2>:  
    statements  
    break;  
  default:  
    statements  
}
```



```
switch (choice) {  
  case 'Y':  
    doThing();  
    break;  
  case 'N':  
    exitProgram();  
    break;  
  default:  
    showHelp();  
}
```

switch can work with byte,
char, short and int values

Flow control: switch

```
switch (choice) {  
  case 'Y':  
    doThing() ;  
    break ;  
  case 'N':  
    exitProgram() ;  
    break ;  
  default:  
    showHelp() ;  
}
```



```
if (choice == 'Y')  
  doThing() ;  
else if (choice == 'N')  
  exitProgram() ;  
else  
  showHelp() ;
```

switch is often a better
way of writing an if-else
statement

Flow control: switch

```
switch (choice) {  
  case 'Y':  
    doThing() ;  
  case 'N';  
  case 'X';  
    exitProgram() ;  
    break ;  
  default:  
    showHelp() ;  
}
```

What happens now when
choice equals 'Y'?

What about 'X'?

Further reading

- ▶ <http://java.sun.com/docs/books/tutorial/java/nutsandbolts/index.html>