The author gratefully acknowledges the assistance of Ian Toyn, Department of Computer Science, University of York, in assisting with the preparation of these notes.
Contents

1 Java & its Architecture ............................................. 7
  1.1 Java Virtual Machine ............................................. 7
  1.2 Garbage Collection ............................................... 8
  1.3 The Java Platform ............................................... 8
  1.4 Java Tools ...................................................... 9

2 Types ..................................................................... 11
  2.1 Primitive Types .................................................. 11
  2.2 Reference Types: Objects, Interfaces and Arrays ......... 12
  2.3 Type Literals ...................................................... 13
  2.4 Casting (Converting Types) .................................... 14
    2.4.1 Casting Primitive Types ................................... 15
    2.4.2 Casting Objects ............................................... 15
  2.5 Arrays ............................................................. 16

3 Operators & Control Structures ................................ 19
  3.1 Logical & Bitwise Operators ................................... 21
  3.2 Operator Overloading ............................................ 22
  3.3 Testing the Type of an Object ................................ 22
  3.4 Control Structures ............................................... 23
3.4.1 else/if & switch ................................. 23
3.4.2 While Loops .................................. 24
3.4.3 Do While Loops ................................. 25
3.4.4 For Loops ....................................... 25
3.4.5 Break & continue ............................... 25

4 Object Orientation .................................. 27
4.1 Dynamic vs Static .................................. 28
4.2 Class Hierarchy .................................... 29
   4.2.1 Everything Inherits Object ............... 30
   4.2.2 Access Control ............................... 30
   4.2.3 Overriding Methods .......................... 31
   4.2.4 Single Inheritance ......................... 32
   4.2.5 Abstract Classes ............................. 32
   4.2.6 Interfaces .................................... 33
4.3 Inner Classes ..................................... 34
4.4 Anonymous Classes ............................... 34

5 Methods, Exceptions & Static Initializers .... 37
5.1 Methods ............................................ 37
   5.1.1 Constructor Methods ....................... 37
   5.1.2 main Methods ................................ 38
5.2 Exceptions ........................................ 38
5.3 Static Initializers ................................ 39

6 Input Output ........................................ 41
6.1 Streams ........................................... 41
CONTENTS

7 Java 5
7.1 Generics ................................................. 43
7.2 Enhanced For Loop ...................................... 44
7.3 Type Safe Enums ........................................ 44
7.4 Autoboxing .............................................. 46
7.5 Varargs .................................................. 47

8 Packages .................................................. 49
8.1 Writing & Compiling Packages ......................... 49
8.2 Using Packages .......................................... 49
8.3 The java.lang Package ................................. 50
  8.3.1 Strings & StringBuffers ............................ 50
  8.3.2 Primitive Wrappers ................................. 51
8.4 The java.util Package .................................. 51

9 Useful Tips ............................................... 53
9.1 Comparing Objects ................................. 53
  9.1.1 Testing Equality ................................. 53
  9.1.2 Testing Magnitude ............................... 54
  9.1.3 Comparing Strings & Regular Expressions .... 56
Chapter 1

Java & its Architecture

Java is a byte code compiled interpreted language. That is to say that converting a text file containing a java program into a process executing on a computer requires the following two steps.

Compilation in which the text is transformed into an assembler code written in eight bit bytes, bytecode.

Execution when the bytecode is run on simulated computer called the Java Virtual Machine (JVM).

Compilation is achieved using a program called javac, the java compiler. During which stage many errors relating to typing and syntax may be detected.

1.1 Java Virtual Machine

The JVM simulates a computer and must provide the following:

- A binary instruction set called bytcodes

- The ability to read a binary file format called the java class file format. This is used to store programs and classes that have been compiled into bytecode.

- A verification algorithm that can be used to identify possibly dangerous programs.

The memory of the JVM is divided into 4 separate areas.
Class Area  this is where any bytecode, that has been loaded, is stored.

Java Stack  this is the stack used by java methods (functions), each call to a method adding a
stack frame to the top of the stack as in any other language.

Heap  the area of free memory where objects are stored.

Native method stack  this is a stack for any methods (functions) that have been written and
compiled in other languages (usually C/C++), and linked to a Java class using the Native
Method Interface.

1.2 Garbage Collection

The JVM provides a garbage collection system, which automatically frees the memory associ-
ated with dead objects. This avoids many of the memory leaks that can occur in other languages.
The garbage collector works by listing all the objects in the heap then identifying which are alive,
then reclaiming the rest. An object is deemed to be alive if any of the following conditions apply.

1. There is at least one reference to the object by a variable on the java stack.

2. There is at least one reference to the object by a variable within an alive object.

3. There is a reference to the object from within the JVM itself.

The garbage collector runs automatically, but can be invoked by a running program using the
gc() method, provided in the java.lang.Runtime object. Garbage collection can be turned off
by giving the -Xnoclassgc switch on the java command line, garbage collections may also be
logged using the -Xloggc:<filename> switch.

1.3 The Java Platform

The JVM is an idealised computer and on its own virtually useless, in order to allow it to do useful
work it is embedded in a larger system called the java platform. The java platform includes sets
of classes providing many useful functions. These classes are divided into packages according
to their use.

java.lang provides basic language features such as ClassLoader for loading classes into
the JVM or String for text processing. This package is always visible and does not need
to be explicitly imported.
java.util provides some useful data structures such as Hashtable, LinkedList or Date.

ejava.io provides basic input output functions.

java.awt provides access to a graphical user interface (GUI).

javax.swing a more advanced GUI package allowing the look and feel of a windowing system such as Windows or KDE to be applied.

1.4 Java Tools

Java comes with a set of tools, than can be used for producing programs and documentation.

javac The Java compiler, takes a Java text file (postfix .java) as its input, and produces a byte code output file (postfix .class).

java The java interpreter, takes a Java class file as its input, and tries to run the byte code within the file. To achieve this it finds and runs the public static void main(String args[]) method. Any command line options are passed to the main method via the String array, which forms its sole parameter. If no command line options are given the parameter array will be set to null. If no main method is found an exception is thrown by the JVM.

jar The Java Archiver, can take an entire program, or package, and produce a Java archive file, which is a zip file containing the class files. Archives will also contain a manifest, detailing information about the classes. In the case of a program, you can provide your own manifest, telling java which class contains the main method. Additional files can be included in an archive, these may be data files, source code, or javadoc web pages.

javadoc The Java documentation tool, automatically generates web pages for a package, showing the class hierarchy. You can add comments before class and method declarations giving extra data.

Programming and compiling can be made much simpler by the use of an integrated development environments (IDE). Several of these exist for java, two of the most popular are:

NetBeans Is Sun’s own IDE, it is available in a single package with the Java language itself, and can be downloaded from www.java.com.

Eclipse Is an open source IDE maintained by a consortium including IBM. It also functions as an IDE for C/C++ and PHP, it is available from www.eclipse.org.
Chapter 2

Types

Java types can be divided into two categories: primitive types, which are designed into the JVM; and composite types consisting of arrays and classes. The sizes of all types except boolean are defined as integer multiples of the java byte, which is defined as eight bits. This is fixed by the JVM and is the same on all hardware.

2.1 Primitive Types

The eight primitive types are the types built into the JVM. A variable of a primitive type simply refers to a region of memory with the appropriate size and format (eg. signed integer). No methods or fields exist within the primitive types.

When primitive types are used in the parameters of a method they are always passed by value. This means that the variable within the method is a copy of the parameter and changes to this will not affect the original parameter.

When used as a class variable primitive types are all assigned a default value, this does not occur when they are used for local variables within a method. Attempting to compile code in which a primitive type is used without first being initialised to a value will generate a compiler error.

Each primitive type has a wrapper class in the java.lang package. Primarily these provide useful methods for handling the types such as valueOf which transforms a string into the appropriate type. For example the following code will transform a String into an int.

```java
String s = "7";
int i = Integer.valueOf(s);
```
### 2. Types

<table>
<thead>
<tr>
<th>Type</th>
<th>Size</th>
<th>Format</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>1 bit</td>
<td></td>
</tr>
<tr>
<td>byte</td>
<td>1 byte</td>
<td>signed integer</td>
</tr>
<tr>
<td>char</td>
<td>2 bytes</td>
<td>unicode character (unsigned int)</td>
</tr>
<tr>
<td>short</td>
<td>2 bytes</td>
<td>signed integer</td>
</tr>
<tr>
<td>int</td>
<td>4 bytes</td>
<td>signed integer</td>
</tr>
<tr>
<td>long</td>
<td>8 bytes</td>
<td>signed integer</td>
</tr>
<tr>
<td>float</td>
<td>4 bytes</td>
<td>floating point number (IEEE 754-1985)</td>
</tr>
<tr>
<td>double</td>
<td>8 bytes</td>
<td>floating point number (IEEE 754-1985)</td>
</tr>
</tbody>
</table>

Table 2.1: A list of all Java primitive types, together with their size and format.

<table>
<thead>
<tr>
<th>Type</th>
<th>Range</th>
<th>Default Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>boolean</td>
<td>true, false</td>
<td>false</td>
</tr>
<tr>
<td>byte</td>
<td>-128 to 127</td>
<td>0</td>
</tr>
<tr>
<td>char</td>
<td>\u0000 to \uFFFF</td>
<td>\u0000</td>
</tr>
<tr>
<td>short</td>
<td>-32768 to 32767</td>
<td>0</td>
</tr>
<tr>
<td>int</td>
<td>-2147483648 to 2147483647</td>
<td>0</td>
</tr>
<tr>
<td>long</td>
<td>-9223372036854775808 to 9223372036854775807</td>
<td>0</td>
</tr>
<tr>
<td>float</td>
<td>±1.4E-45 to ±3.4028235E+38</td>
<td>0.0</td>
</tr>
<tr>
<td>double</td>
<td>±4.9E-324 to ±1.7976931348623157E+308</td>
<td>0.0</td>
</tr>
</tbody>
</table>

Table 2.2: The allowable and default values for all Java primitive types. Note only class variables have default values. Local variables within methods have no defaults and will generate compile time errors if you try to use them uninitialised.

As their name suggests, the wrappers classes can be used when primitive types have to be handled as objects. For example when they are to be added to a data structure that only accepts objects java.util.Vector for example.

### 2.2 Reference Types: Objects, Interfaces and Arrays

Reference types are data structures formed out of primitive types and blocks of executable code. When a reference type is instantiated an object of the type is constructed in the memory of the JVM.

There are 3 distinct reference types in Java, they are:

- **Objects** containers holding fields (variables, constants) and methods (functions)
Interfaces containers for method signatures (function names) or static fields (constants)

Arrays special types of objects containing numbered lists of primitive values or objects or arrays (multidimensional array)

The instantiation of a class is called an object, this exists as a data structure in the memory of the JVM. Any variable which has a class type and has been instantiated will be a handle on the area of memory that forms an object of the given class. When passed as a parameter to a method the object is not copied but instead a new reference is made to it from within the method. This means that any change made to the object within the method will remain when execution of the method has finished.

Classes are extensible so a new subclass can be declared as an extension of an existing class. The subclass will inherit fields and methods from its superclass if they were not declared private.

Interfaces cannot be instantiated but may be implemented by a class, in which case the class must provide a definition of any methods named in the interface and can access any constants declared in the interface. Interfaces may be extended in just the same way as classes.

There exists a reference null to which any variable with reference type can be set. Reference type class level variables are initialised to null by default, unless otherwise instantiated. Local variables within methods are not instantiated in any way and attempting to use them, without at least assigning null to them, will result in a compiler error.

2.3 Type Literals

Type literals are where a value is typed into the source code, see table 2.3.

```
int i = 7;
String s = "Hello World!";
```

Type literals are processed by the compiler. Unless otherwise instructed the typing is:

- integer values are transformed to type int;
- decimals to double;
- anything between double quotas is a String;
- any character in single quotas is a char;
2. Types

### Table 2.3: A list of possible literals together with the type they will be transformed into by the compiler.

<table>
<thead>
<tr>
<th>Literal</th>
<th>Example</th>
<th>Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>whole numbers</td>
<td>7</td>
<td>int</td>
</tr>
<tr>
<td>decimal numbers</td>
<td>7.5</td>
<td>double</td>
</tr>
<tr>
<td>characters</td>
<td>'b'</td>
<td>char</td>
</tr>
<tr>
<td>strings</td>
<td>&quot;hello&quot;</td>
<td>String</td>
</tr>
<tr>
<td>booleans</td>
<td>true</td>
<td>boolean</td>
</tr>
</tbody>
</table>

Table 2.4: A list of possible labels that can be applied to literals. The two prefixes force the compiler to read integers in different number bases. The three postfixes allow different types to be forced on integers and floating point numbers.

<table>
<thead>
<tr>
<th>Label</th>
<th>Pre/postfix</th>
<th>Example</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (zero)</td>
<td>prefix</td>
<td>0733[1]</td>
<td>base 8 (octal)</td>
</tr>
<tr>
<td>0x</td>
<td>prefix</td>
<td>0x7fa</td>
<td>base 16 (hexadecimal)</td>
</tr>
<tr>
<td>\u</td>
<td>prefix</td>
<td>\u07af</td>
<td>unicode base 16</td>
</tr>
<tr>
<td>f or F</td>
<td>postfix</td>
<td>7.5f</td>
<td>float</td>
</tr>
<tr>
<td>d or D</td>
<td>postfix</td>
<td>7.5d</td>
<td>double</td>
</tr>
<tr>
<td>l or L</td>
<td>postfix</td>
<td>7l</td>
<td>long</td>
</tr>
</tbody>
</table>

[1] digits 8 and 9 must not occur

- the special keywords `true` and `false` exist for type `boolean`.

It is possible to use prefix and postfix characters to change the compilers interpretation of a literal. So the prefix `0x` means interpret the following as an integer in hexadecimal format, see table 2.4.

### 2.4 Casting (Converting Types)

It is possible to convert types in Java, this is known as *casting*. A cast operation may occur automatically, for example when an `int` type is added to a `double` type, the `int` is cast to a `double`. Alternatively the cast may be explicitly implemented by the code *(explicit cast)* by writing the name of the new type before the variable to be cast in brackets, for example to cast an `int` to a `double` you would write `myDouble = (double) myInt`. 
2. Types

```java
int i = 7;
double d = 9.5;

// quick way of adding
double sum = i + d;

// above is equivalent to
double temp = (double) i;  // cast the int to double
double sum = temp + d;    // add the doubles
```

2.4.1 Casting Primitive Types

Java can carry out type conversions between all primitive types except boolean (remember char is a positive number representing a unicode character). There are 2 types of primitive cast widening and narrowing.

Widening conversions occurs when a type is converted to a new type with a wider range of values. Java will perform these conversions automatically.

Narrowing conversions occur when a type is converted to a new type with a narrower range of values. Because it is possible to lose information Java will only allow such type conversions with an explicit cast.

2.4.2 Casting Objects

For a objects a widening cast means casting an object up to an object of its own superclass. This can be performed automatically. A narrowing cast means casting an object from an object of type superclass to an object of typo subclass. This must be performed by an explicit cast.

The following examples use objects of the types Object and one of its subclasses String.
2. Types

Java Notes

```
\\ will work
String s = new String("hello world");
Object o = s;  \\ widening
String tmp = (String) o;  \\ narrowing

\\ will not compile
String s = new String("hello world");
Object o = s;
String tmp = o;  \\ compiler error narrowing with no explicit cast

\\ will compile but fail at runtime
String s = new String("hello world");
Object o = s;
Integer tmp = (Integer) o;  \\ throws java.lang.ClassCastException
```

2.5 Arrays

An array is a reference type consisting of a numbered list of items of the same type. The items may be primitive types, objects or other arrays, so long as they have the same type. Internally an array contains a contiguous block of memory and holds a pointer to the end of the block where the first element is put. All other elements are found by multiplying the number of the element by the memory size of the type and adding that to the pointer to find their memory location. This is why all the elements in an array must be of the same type. The advantage is that array access is fast and independent of size.

Arrays are declared using square brackets [], these may be placed after the type to make all variables array type or after the variable name to make only that variable an array.

```
int[] i, j, k;  \\ i j and k are arrays
int i, j[], k;  \\ only j is an array
```

Elements within arrays are accessed by using the [] operator to identify the element.

```
i[5] = 78;
```

Arrays may be explicitly initialised either when they are declared or at a later stage. If not initialised, class arrays are set to null, while locals are just left uninitialised. An array can always be initialised by using the new keyword and the type, at declaration an array may also be initialised by providing a list of elements (array literal).
2. Types

```java
int[] i = new int[5]; \ all set to zero
int[] s = new String[5]; \ all set to null
int[] i = {0, 1, 2, 3, 4}; \ array full with array literal
int[] i; \ declare array variable
i = new int[5]; \ initialise array all zero
int[] i;
i = new int[] {0, 1, 2, 3, 4} \ initialise array from literal;
```

Multidimensional arrays are always implemented as arrays of arrays, which means arrays may be ragged.

```java
\ i is a size 5 array of size 3 int arrays
int[][] i = new int[5][3];

\ j is a ragged array
\ j[0] is size 3
\ j[1] is size 4
int j[][] = {{0, 0, 0}, {1, 2, 3, 4}};

\ an alternative way of producing a ragged array
int k[][] = new int[2][];
k[0] = new int[] {1, 1, 1};
k[1] = new int[] {5, 6, 7, 8};
```

All arrays contain the variable length which simplifies writing loops.

```java
for (int i = 0; i < myArray.length; i++)
    foo(myArray[i]);
```
Chapter 3

Operators & Control Structures

Having described types of variables that can occur in Java, we need to examine how they can
be put together, forming expressions that actually do something. This is achieved by the use of
operators, which specify various actions that can be performed on variables.

The simplest expression is assignment (=) that forces a variable to take a particular value.

```
    a = 7;
    String s = "Hello world";
```

The mathematical operators are also very straightforward as they have the same meaning as in
arithmetic.

```
    a = 7 + b;  \ \ \ a has the value of b plus 7
    a = b / c;  \ \ \ a has the value of b divided by c
```

These operators must be applied to any numerical types, which are the numerical primitives such
as int or float, and their wrapper classes Integer Float. They will cast the narrower type
to the broader, so an int plus a float will give a float result. Table 3.1 contains a list of
numerical operators.

There are also comparison operators such as equals, which compare two variables, returning
true if the relation between the variables holds and false otherwise. The comparisons equals
and not equals can be applied to all variables, but references are treated differently. The magni-
tude comparisons, such as greater than, only apply to numeric types, that is primitives excluding
the type boolean. A full list of comparison operators is given in table 3.2.
3. Operators & Control Structures

<table>
<thead>
<tr>
<th>Operator</th>
<th>Example</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ –</td>
<td>a + 7</td>
<td>add and subtract</td>
</tr>
<tr>
<td>/ *</td>
<td>a * b</td>
<td>divide and multiply</td>
</tr>
<tr>
<td>%</td>
<td>a % b</td>
<td>remainder</td>
</tr>
<tr>
<td>++, —</td>
<td>a++</td>
<td>increment or decrement by 1</td>
</tr>
</tbody>
</table>

Table 3.1: A list of Java operators for numeric types. These can be applied to any numeric primitive byte, char, int, long, float, double and to their wrapper classes.

```java
int a = 7;
int b = 8;
a == b;  \// evaluates to false

String s0 = "hello world";
String s1 = "hello world";
s0 == s1;  \// evaluates to false (different objects)
s0.equals(s1);  \// returns true
```

When applied to reference types, the `equals` and `not equals` operators compare the memory addresses of the objects to which the references are pointing. The result is, that if two strings containing the text “hello” are created and assigned to different variables, the variables will be considered not equal. However if one string is created, and two variables set to point to it, the variables would be considered equal. Reference types can be compared for equality using the `equals(Object o)` method, and any types that require such comparison should provide their own version of the method. A full description of comparisons for reference types can be found in section 9.1.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Applies to types</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>==</td>
<td>all</td>
<td>equal to</td>
</tr>
<tr>
<td>!=</td>
<td>all</td>
<td>not equal to</td>
</tr>
<tr>
<td>&lt;</td>
<td>numeric primitive</td>
<td>less than</td>
</tr>
<tr>
<td>&lt;=</td>
<td>numeric primitive</td>
<td>less than or equal to</td>
</tr>
<tr>
<td>&gt;</td>
<td>numeric primitive</td>
<td>more than or equal to</td>
</tr>
<tr>
<td>&gt;=</td>
<td>numeric primitive</td>
<td>more than</td>
</tr>
</tbody>
</table>

Table 3.2: A list of Java operators used for Comparison. The first two items `equals` and `not equals` can be used with all types. The remainder can only be used for numeric primitives, that is all primitives except `boolean`. 
3. Operators & Control Structures

3.1 Logical & Bitwise Operators

Logical and bitwise operators are similar, in that they both provide operations such as “and” and “or”. The difference is that logical operator take boolean expressions and return booleans. The bitwise operators take the type int and returns an int, in which the value of each bit is the result of applying the operator to the bits in the operands.

```java
int a = 1, b = 2;
int c = a | b; // the binary OR of 01 and 10 has value 11

boolean boolA = true, boolB = false;
boolean boolC = boolA | boolB; // the logical OR of (true, false)
has value true
```

The type boolean also has 2 conditional operators reserved exclusively for it, they are alternative versions of the “and” and “or” operators. They differ from the ordinary operators in that the second operand is only evaluated if the first operand has not determined the value of the operation. This is useful if the operation is using the results of complex method calls. Boolean types also have a special operator “not”, which returns true if the value it is applied to is false and vice versa.

```java
// the second hasEntry() will only be evaluated if the
// first returns true.
database1.hasEntry("hello") && database2.hasEntry("hello");

// the second hasEntry() will only be evaluated if the
// first returns false.
database1.hasEntry("hello") || database2.hasEntry("hello");
```

Java allows many operations to be combined with the assignment operator. When this occurs the operation is applied to the operands and the result assigned to the first operand.

```java
int a = 5, b = 7;

a += b \ // a would equal 12
a -= b \ // a would equal -2
a *= b \ // a would equal 35
```
### 3. Operators & Control Structures

<table>
<thead>
<tr>
<th>Operator</th>
<th>Applies to</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&amp;</td>
<td>integer, boolean</td>
<td>and</td>
</tr>
<tr>
<td>^</td>
<td>integer, boolean</td>
<td>xor</td>
</tr>
<tr>
<td></td>
<td></td>
<td>integer, boolean</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td>boolean</td>
<td>conditional and</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>!</td>
<td>boolean</td>
<td>not</td>
</tr>
</tbody>
</table>

Table 3.3: A list of Java logical and bitwise operations. The bitwise operator apply to any integer types, which are primitive types byte, char, int, long and objects of their wrapper classes. Used as boolean operator they take any boolean types, the primitive type boolean and it wrapper class Boolean. The conditional operations are boolean, and only evaluate the second argument if the first is not sufficient to define the output. The not operator applies to boolean types and returns the inverse of the argument.

Strings have their own operator for concatenation (joining), it is + operator. This is the only operator, other than assignment defined for strings.

```java
String s0 = "hello";
String s1 = "world";

s will have value "hello world"
String s = s0 + " " + s1;
```

### 3.2 Operator Overloading

The ability of a single operator, to be defined correctly for different arguments, is called operator overloading. For example + is defined as the addition operator for the int primitive and the Float class, but as the concatenation operator for the String class. In Java users cannot define new overloaded operator for their own types. However the assignment operator = is always provided automatically for every new class.

### 3.3 Testing the Type of an Object

The instanceof operator takes an object and a class name as its parameters returning true if the object is an instance of the class and false otherwise.
3. Operators & Control Structures

3.4 Control Structures

There are

if/else  simple branch

switch  branch on a large set to choice, effectively the same as a large set of else if statements.

for loop  iterate a fixed number of times

while loop  iterate while a condition holds, possibly 0 iterations

do while loop  iterate while a condition holds, at least 1 iteration

break  break out of a loop and starts next statement.

continue  exits the current loop and starts the next iteration, jump to the test (increment and test in case of for loop.

3.4.1 else/if & switch

The if else construct is very simple, the brackets following the if statement must contain an expression that evaluates to a boolean. If this expression evaluates to true the statement following the if is carried out. In the case of an if/else construct, the statement following the else is carried out when the test expression evaluates to false. Note that in Java a group of lines in braces (\{\}) counts as a statement and so can be placed after an if or else.

```java
if(flag < 0)
    something ;    \ carried out when flag < 0
else{
    line1;    \ These lines carried out
    line2;    \ when flag >=0
}
```

It is possible to nest an if statement, within another if statement

```java
if(flag < 0){
    if("help".equalsIgnoreCase(s))
        code for (flag < 0) and (s equals help) here;
    else
        code for (flag < 0) and (s not equal to help) here;
}
```
It is possible to follow an else with another if to allow testing for many conditions.

```java
if(flag < 0)
    code for flag less than 0;
else if(flag == 0)
    code for flag equals zero; // These lines carried out
else if(flag < 10)
    the code for flag in range 1 to 9;
else
    the code for all other cases;
```

A switch statement can be faster than a long list of else if statements. In this structure possible int values are listed, each in a case statement. Each case statement contains a block of code, which is executed, if the value of the variable matches the value defined in the case statement. The tested variable must be of type int or a type that can be automatically cast to an int, or in Java 1.5 an enum type, see 7.3.

When a case has been executed the program does not leave the switch, but continues down the remaining case statements in the switch. This behaviour can be avoided by using a break to jump to the end of the switch, see 3.4.5. Since all possible values of the type int are un-lightly to be listed, a default statement is provided for any not listed.

```java
switch(n){
    case 1:
        if n equals 1 start here;
        //falls through to case 2
    case 2:
        if n equals 2 start here;
        break; // jumps to end
    default :
        if n is other than 1 or 2 start here;
}\\ the break statements jump to here
```

### 3.4.2 While Loops

In a while loop, a boolean test expression is evaluated and, if true, the following statement is executed, in a similar manner to an if statement. However when the statement has been evaluated the program returns to the test expression again, a process that repeats until the test expression evaluates to false. This is a pretested loop and may never be executed if the test expression is false when it is first entered.
3. Operators & Control Structures

3.4.3 Do While Loops

In a do while loop, the loop is first entered and executed by the program, and then a test expression is evaluated. If the test expression evaluates to true the loop is repeated, a process which continues until the expression evaluates to false. This is an example of a post-tested loop, which has to be executed at least one time. Note there has to be a semicolon after the final test.

```java
do{
    do something that changes n and or j;
}while(n<5 && j>10);
```

3.4.4 For Loops

The for loop is the most common, it is designed to be used where a loop must be repeated until a counter passes some value. The loop has initialise, test and update expressions at the start, followed by a body. The initialise statement is executed on entry, after which the test is evaluated and if true, the body of the loop is executed. The update expression is executed after the body, then the program returns to the test expression.

```java
for(initialise; test; update){
    do something;
}
for(int i=0; i < myArray.length; i=i+1){
    do something;
}
```

3.4.5 Break & continue

The break and continue statements allow the program to immediately exit a loop, bypassing the usual tests. A break jumps out of the bottom of the loop, while a continue stops the current iteration of the loop and starts the next one. When loops are nested they can be labelled, allowing the break and continue statements to specify which level of nesting the are to exit, see figure 3.1.
3. Operators & Control Structures

Figure 3.1: An example of the use of break and continue. In the code the outer of a pair of nested loops has been labelled coldtrance, while the inner loop contains a break or continue statement guarded by an if and some other code. A simple break in the inner loop causes execution to jump to the end of the inner loop, while a continue jumps to the next iteration of the loop missing the other stuff. However if the statements are labelled to jump to coldtrance then the break is to the end of the outer loop and the continue is to the beginning.
Chapter 4

Object Orientation

Java is an object orientated language (OOL), and every java program must contain at least one object. An object is a region of the memory in the JVM containing data and methods (functions) that operate on the data.

A class is the description or definition of a particular type of object. The class defines the data that objects of this type can hold and the methods that can operate on that data. The class is the design that the Java will use to build an object in the memory of the JVM.

The essential idea behind OO programming is, that objects in the program represent the conceptual objects used to reason about the problem being addressed by the program. Hence in a banking program, the objects that would occur would be customer and account. Objects of each type would have to hold the appropriate data for their type, and methods for accessing that data. So the customer object would have to contain name and address fields, together with methods to access and change them.

Once we have identified the objects, that are to be used in a program, the classes defining the objects can be designed. During the design pay attention to the following points, which are expanded in subsequent subsections.

Dynamic fields and methods are those that belong to an individual object. Their use makes no sense when there is no object. Any field or method is dynamic by default unless otherwise specified.

Static fields and methods belong to the class as a whole, and may be accessed without constructing an object. The keyword static is used to declare these.

Access control to the data and methods in object, is enforced by the keywords public, protected, private.
Hierarchy allows new classes to be defined as extensions of existing classes, inheriting their data and methods.

4.1 Dynamic vs Static

When a class is defined, a field or method may be declared as static. Which means that the field or method belongs to the class itself, and not to the individual objects of that class. A good example of the use of static methods and fields is in the java.lang.Math class, where methods such as sqrt and fields such as PI are declared to be static. If you want to calculate \( x = \sqrt{a\pi} \), you write \( x = \text{Math.sqrt}(a \times \text{Math.PI}) \). Both the method and the field are accessed directly, by using the name of the class, no object of type Math is required.

Any field or method that is not declared static is dynamic by default. The result is that they are available only in objects of the class type, not in the type itself. If a Person class were defined to hold a person’s name, the name and its access method would have to be dynamic. However if the total number of persons created was required, a static field could be added, which each new object would increment. The following code illustrates how it could work.

```java
public class Person{
    private String name;
    private static int count; // class variable initialised to 0

    Person(String name){
        this.name = name;
        count++;
    }

    public String getName(){
        return name;
    }

    public static int getCount(){
        return count;
    }
}
```

The field count belongs to the class Person, and not to any particular object of that class. The count++ instruction in the constructor, ensures that when a new object of type Person is created, the count is incremented.
4.2 Class Hierarchy

A central theme of OO programming is the use of hierarchy to capture the relationships between objects. This is best illustrated by an example, consider a bank having a well written Person class. For its own records the bank needs both a Customer and an Employee type. Both of these require the use of a Person type, but both of them require unique data of their own: tax and national insurance in the case of employees; and the accounts held in the case of customers. There are two obvious ways of doing this without using a class hierarchy.

1. Include all the extra data in the Person class. This has the drawback that there is a lot of redundancy in the class. Also it is possible a programmer may accidently treat an Employee as a Customer, with unpredictable results.

2. Write separate classes each including a Person field, with wrapper methods for all the Person methods. The problem in this case is that all the methods that have been implemented in Person have to be re-implemented twice.

A better solution is to subclass (or extend) the class Person, producing the new types Employee and Customer. Each of these types inherits the properties of a Person, but each add different fields and methods of their own. The Employee class can add fields for tax code and national insurance number, together with the access methods, while Customer adds the fields that its data requires.

```
public class Employee extends Person{
    private String niNumber;
    private String taxCode;

    public Employee(String name, String niNumber, String taxCode){
        super(name);
        this.niNumber = niNumber;
        this.taxCode = taxCode;
    }

    // getter and setter methods
}
```

The key word extends, tells Java that Employee is a subclass of Person. This means that, all the public and protected methods of Person, can be called on an object of type Employee. The method super, called in the constructor, calls the constructor of type Person, passing it the persons name. The following sections describe various features related to class hierarchy.
4. Object Orientation

4.2.1 Everything Inherits Object

In Java it is not possible to design a class hierarchy independent of everything else, as you can, for example, in C++. In Java every object must be a subclass of java.lang.Object. This has the advantage that containers, defined to store items of type Object, can store any reference type. It also enforces a certain uniformity in design of objects. Unfortunately it introduces the confusing concept of objects of the class Object.

The class Object provides the following useful methods that may often have to be overridden in subclasses. Three of the most commonly used are:

- `equals(Object o)` returns `true` if `o` is equal to the object on which the method is called, and `false` otherwise.
- `toString()` returns a description of the object. This is almost always worth overriding, if only for debugging purposes.
- `clone()` returns a copy of the object. This seems simple, but when objects have internal states, such as “activated”, or when they hold resources, such as input streams, the policy is not always clear.

4.2.2 Access Control

Three keywords are applied to fields (methods and variables in classes) in order to enforce access control.

- **public** Means that the field or method can be accessed from outside the class or object.

- **protected** Means that the field or method may be accessed by subclasses but not by any outside objects. So if in Person, the name field were to be made protected, methods in Employee would no longer have to use `getName()`. However any program handling an Employee or Person object would still have to call `getName()`.

- **private** Means that the field or method can only be accessed by other methods in the same object. In Person the name field is declared private, hence a public method `getName()` is required, to allow other objects access to the value of the field. Even a subclass of Person such as Employee cannot access the name field, so methods within Employee that need to use the name must call `getName()`.
4. Object Orientation

**No keyword** Generates a condition called as **package-private**, in which a field or method can be accessed by methods in the same class, or from other classes in the same package. However it can not be accessed from the outside the package, or from subclasses (unless subclasses are in the package).

### 4.2.3 Overriding Methods

It is possible for a subclass to redefine methods in its parent class, this is called **overriding** the method. When a program calls an overridden method Java will automatically call the correct method. An example of overriding a method is shown in the following code, in which a class Poly overrides the `toString()` method, of the of the class Object.

```java
public class Poly{
    private String name;

    Poly(String name){
        this.name = name;
    }

    public String toString(){
        return name;
    }
}
```

Now any call of the `toString()` method of Poly will return the `name` field. This will apply even if the variable holding the reference to the Poly object is of type Object. Java will automatically identify the correct version of `toString()` to call. The decision of which method to use is taken based on a method signature, which is made up of the following items.

1. The method’s name.
2. The number, types and order of the parameters.
3. The type of the return value.
4. The type of the checked exceptions thrown by the method.
5. Any other modifiers such as `public` or `private`
4. Object Orientation

4.2.4 Single Inheritance

In Java a class can only inherit from one parent, it is possible for the parent to inherit from a grandparent and so on indefinitely. Multiple inheritance, which can be used in C++, is expressly forbidden in Java. This simplifies the design of the JVM, as the complexities of resolving the cases where method with the same signature have been provided by both parents are avoided. It also avoids the problem of what to do if both parents inherit from a single grandparent. The circumstances where multiple inheritance is required can be handled using interfaces, see section 4.2.6.

4.2.5 Abstract Classes

It is possible to define abstract classes which are like an ordinary class, but must contain at least one abstract method, possibly inherited. An abstract method give the name, return type, parameters and access control of a method, but do not define it. An abstract class must be defined by the keyword abstract, and must contain at least one abstract method, which also requires the keyword abstract. An abstract class cannot be instantiated, but may be used as a base class for other classes, which must implement all unimplemented abstract methods above themselves in the hierarchy. Alternatively abstract classes can be extended by other abstract classes.

```java
public abstract class Animal{
    int age;
    public Animal(final int age){
        this.age = age;
    }

    public abstract void makeNoise();
}

public class Cow extends Animal{
    public Cow(final int age){
        super(age);
    }

    public void makeNoise(){
        System.out.println("Moo");
    }
}
```
4. Object Orientation

4.2.6 Interfaces

An interface consists of the definition of some implicitly abstract methods that any class implementing the interface must provide. Any class can implement an interface by using the `implements` keyword and providing the required method or methods. An interface is also a type so a method can be defined as returning an interface type or taking it as a parameter. There are two important differences between abstract classes and interfaces.

- A class can only extend one parent class but it can implement any number of interfaces. This is how Java avoids the multiple inheritance used in C++.

- An interface must contain no implementation or variable fields (unless they are `static final`), while an abstract class can provide data fields and implemented methods. Indeed an abstract class should provide such things if it did not you would have used an interface.

```java
public interface NoiseMaker{
    public void makeNoise();
}

public class Cow implements NoiseMaker{
    int age;
    public Cow(final int age){
        this.age = age;
    }

    public void makeNoise(){
        System.out.println("Moo");
    }
}
```

In the above code note, that the `Cow` must now look after its own age variable, as an interface cannot provide a data field. The advantage is, that the interface can be used for defining classes where age would not be required. In general interfaces represent a higher degree of abstraction than is provided by abstract classes. The interface `NoiseMaker` is a type in its own right, so an array of type `NoiseMaker` could be defined, or a method taking a `NoiseMaker` parameter.
4.3 Inner Classes

In Java it is possible to define one class inside another class. The classes are said to be nested, and the one on the inside is called an inner class. This can be very useful when writing multi-threaded code, or user interfaces. In these applications a class must be written to handle some event, such as a button click. There is no problem with this except, the class which handles the click must be defined in a separate file to the class that provides the button. Also the handler class may need access to internal data, from the class that defines the button.

The solution is to define the handler as an inner class of the class providing the button. An inner class can see all the fields and methods of the class in which it has been defined, allowing the inner class to access the internal state of the button class. It also allows the handler code to be defined in the same file as the code defining the button, see figure 4.1.

4.4 Anonymous Classes

It is possible to take the idea of inner classes one step further, and actually define the class in the location where it is created. In this case the class will have no name, hence the title anonymous. The class must be a subclass of some previously defined class or interface.

For an example, consider how the event generated by a button being clicked is handled. The Button class has a method called addActionListener(ActionListener a). ActionListener is an interface requiring any subclass to implement the method actionPerformed(ActionEvent e).

It would be possible to define an inner class implementing ActionListener, then make an instance and pass it to a button. However if only a simple implementation of actionPerformed is required this may be excessive. In which case the class can be defined in the action of creating a new ActionListener, see the following code.

```java
Button button = new Button("please click");

button.addActionListener(new ActionListener(){
    public void actionPerformed(ActionEvent e){
        System.out.println("Thank you for clicking");
    }
});
```
The first line of the above code creates a new Button object. This has an ActionListener added, so that it can do something when the button is clicked. The code uses the new keyword to create an ActionListener, but this is only an interface so a full class definition must to be provided. The definition follows the constructor method and is enclosed in curly brackets {}, it defines the required actionPerformed method.

Generally anonymous classes should be kept very small only doing one thing. Anything more and the advantages of defining your class where it is needed are outweighed by the impenetrability of the code.
public class Outer{
    private String name;
    
    public Outer(String name){
        this.name = name;
    }
    
    public String toString(){
        Inner in = new Inner(" dolly");
        return in.toString();
    }
    
    public class Inner{
        private String iName;
        
        public Inner(String name){
            this.iName = name;
        }
        
        public String toString(){
            return name + iName;
        }
    } //end Inner
    
    public static void main(String args[]){
        Outer p = new Outer("hello ");
        System.out.println(p.toString());
    }
} //end Outer

Figure 4.1: A rather forced example of the use of an inner class. The class Outer contains a class Inner that has a toString() method that uses the private name field of Outer. If the code is run, it will print out the words “hello dolly”.
Chapter 5

Methods, Exceptions & Static Initializers

Java classes can store code for execution in two ways:

Methods are the embodiment of the mathematical idea of a function. They take parameters carry out an operation and return a result.

Static Initializers are block of code with are guaranteed to run when a class if first loaded into a JVM

5.1 Methods

A method is block of code in a class with a name that can be used to call it. It may have a list of parameters that will appear as local variables in the block of code. A method may return a value, or can be declared void. If an error occurs while a method is running, a throwable object may be returned by the method. Further discussion of this is in section 5.2.

Several methods may be declared having the same name, but with different sets of parameters, this is called polymorphism. Also subclasses can declare methods with the same name and parameter list as their super classes, a feature called overriding.

5.1.1 Constructor Methods

Special methods that have no return values and the same name as the class can be written. These are called constructors, and are used to build an instance of the class.
5. Methods, Exceptions & Static Initializers

5.1.2 main Methods

For a stand alone program to run there must be a method called main.

5.2 Exceptions

Exceptions are objects, returned by a method in place of the usual return value. They are usually used to indicate an error, or some other exceptional condition, hence the name. All exceptions in Java are derived from the class java.lang.throwable. There are 2 basic types of throwable objects in Java.

Error exceptions of this type indicate very serious problems such as the JVM running out or memory.

Exception this is used for less serious, possibly recoverable, errors, such as an array index being out of bounds.

There is a further distinction between checked and unchecked throwables.

unchecked methods which throw these do not have to declare so with a throws statement, they can simply be allowed to “bubble up” to the main method. All subclasses of Error and RuntimeException are unchecked.

checked methods which throw these must (including main) declare they do so, using a throws statement. These exceptions are generally caught and handled within a program.

The reason for this classification, is that the problems signalled by subclasses of Error or RuntimeException represent problems that can occur at any time in most, or all methods. For example an out or memory error, or an illegal argument exception. Requiring all methods to declare “throws Error, RuntimeException” would be onerous, so these exceptions are allowed to be unchecked.

Examples of checked exceptions are end of file exception, and illegal access exception. Not only are these less common, but they tend to be the sort of thing that a programmer would want to catch and do something about. Hence the compiler checks that any method that throws such an exception, must either declare it does using a throws statement, or handle the exception itself.

Exceptions are handled using a try / catch / finally block. The code that may throw an exception or exceptions goes in the try part and the exceptions are handled by the catch part. The finally part contains code that is always run whether or not an exception is thrown.
try{
    Do some file reading;
}catch(FileNotFoundException e){
    Sort out how to get right file
}catch(EOFException e){
    Whatever you do when the file is finished
}finally{
    This code will run if
    an Exception is thrown and caught
    an Exception or Error is thrown and not caught
    nothing at all is thrown
    it always runs
}
This code will not run if an Exception or Error is
thrown and not caught cleaning up files should not go here

5.3 Static Initializers

Static Initializers are blocks of code that are run when a class is loaded into the JVM.

static{
    assign some static variable;
}
Chapter 6

Input Output

Java has 3 methods of input output (IO).

**Graphical user interface** (GUI) access to the computer’s windowing system, if there is one.

**java.io** the ability to read text, or bytes of binary data, to and from Streams, which encapsulate terminals, sockets or files.

**java.nio** (New IO) a more advanced package, optimised for moving large amounts of data between multithreaded programs, over networks.

6.1 Streams

The basis of IO in java is the stream, which is a one way flow of data into, or out of, a program. A InputStream is terminated in the program by an object of type Reader, while an OutputStream has a Writer at its end.

Two output and one input streams are provided in the java.lang.System object that is always available in Java. The streams are:

- **out** standard output to the terminal
- **err** standard error to the terminal
- **in** standard input from the terminal
Files may be accessed using the FileWriter and FileReader classes. Note that opening the FileWriter may throw an IOException.

```java
try{
    FileWriter output = new FileWriter(new File("myFile"));
    output.write('"hello"');
    output.close();
} catch(IOException e){
    // handle the exception
}
```

If you require the print and println methods the FileWriter can be placed inside a PrintWriter.

```java
FileWriter fileWrite = new FileWriter(new File("myFile"));
PrintWriter output = new PrintWriter(fileWrite);
output.println("hello world");
```

In order to read lines of text from a file we use a BufferedReader, this provides the method readLine() which, unsurprisingly, reads a line of text from a file. BufferedReader also provides a ready() method, which returns true, if there is more data to be read in the file and false otherwise. This allow data to be read from a file using a while loop.

```java
try{
    BufferedReader input = new BufferedReader(
        new FileReader("myFile"));

    while(input.ready()){  
        String in = input.readLine();
        System.out.println(in);
    }

    input.close();
} catch(IOException e){
    // handle the exception
}
```
Chapter 7

Java 5

Java 5 is the latest version of Java, you can find it on the Sun site as Java 1.5. It contains a number of improvements over the existing Java 1.4 version.

7.1 Generics

Generics allows data structures from the java.util package to be defined as containers for objects of a particular type. This avoids the need for casting the contents back to their original types, and allows for compile time type checking.

The following code exhibits how generics can be used. The method takes a Collection, that has to be of type String, as a parameter, so Vector<String> or ArrayList<String> will be acceptable. Any attempt to pass a Collection that is untyped, or of the wrong type would result in a compile time error.

```java
static public void printHello(Collection<String> c){
    Iterator<String> i = c.iterator();
    while(i.hasNext()){  
        String s = i.next();
        if(s.indexOf("hello") != -1)
            System.out.println(s);
    }
}
```

Note, that the Iterator is also of type String, so that the next() method returns a String, rather than a Object. This avoids explicitly casting the returned objects to their original types, and provides greater type safety because the types can be checked by the compiler.
7.2 Enhanced For Loop

This feature, which works with Generics, gives Java a “for each element in collection” loop, similar to those available in Python or Perl. The following example shows how the loop works.

The loop is declared using the keyword for, in the brackets following this a Collection is named, and a variable of the same type as the collection declared, the two separated by a colon. The loop will then steps through each object in the container, assigning it to the variable. The variable must be of the same type as the collection, or a super type, if not a compiler error will occur.

```java
static public void printLength(Collection<String> c){
    for(String s : c)
        System.out.println("String length "+ s.length());
}
```

It is possible to use the enhanced for loop with untyped containers, by declaring the variable to be of type Object. You will then be limited to the methods in Object, or use an explicit cast.

```java
static public void printToString(Collection c){
    for(Object o : c)
        System.out.println("String is: " + o.toString());
}
```

Since, the ordinary arrays used in Java have to be typed, they can be used in enhanced for loops as well as collections.

```java
static public void printInts(int array[]){
    for(int i: array)
        System.out.println(i);
}
```

7.3 Type Safe Enums

Enumerations allow the construction of a finite set of labelled elements, as in other languages such as C. However, in Java, each element of an enumeration may be treated as a class in its own right. The basic enumeration type is introduced using the enum keyword, and variables of an enumeration type can be used in a switch or if statements.
public enum Season{WINTER, SPRING, SUMMER, FALL}

Season timeOfYear = Season.WINTER;

System.out.println(timeOfYear.toString());  \ prints WINTER

switch(timeOfYear){
    case WINTER:
        do something
        break;

    case SUMMER:
        do something else
        break;
}

An enumeration is treated as a class definition, in which each constant is a separate object. Each constant object implements all the methods associated with an object, such as toString(), which returns the elements name. It is possible contain data within an enumeration, and provide access and set methods that use that data. Also each enumeration has a values() method, which can be used with an enhanced for loop.

public enum Season{
    WINTER(-1.5), SPRING(5.0), SUMMER(23.0), FALL(7.0);

    private final double averageTemperature;

    private Season(final double averageTemperature){
        this.averageTemperature = averageTemperature;
    }

    public double averageTemperature(){
        return averageTemp;
    }
}

for(Season s: Season.values())
    System.out.println(s + " average temperature is " +
    s.averageTemperature());
It is even possible to specify a method as abstract, and then have each element provide its own implementation of the method. An example of this is given in the following, where the enumeration Operation defines an abstract method \texttt{eval}, and each element provides its own implementation.

```java
public enum Operation {
    ADD {
        public double eval(final double a, final double b) {
            return a + b;
        }
    },
    SUBTRACT {
        public double eval(final double a, final double b) {
            return a - b;
        }
    }
}

public abstract double eval(final double a, final double b);
```

### 7.4 Autoboxing

Autoboxing allows Java to automatically wrap a primitive, in a wrapper type, when it is placed into a collection, and unwrap it when it is extracted. Combined with generic collections it can provide better type safety, the following code illustrates how it works.

```java
int i = 5;
Stack<Integer> v = new Stack<Integer> ();
v.push(i);  // auto box int in Integer

int j = v.pop ();  // auto unbox Integer to int
```

Note, this is no quicker than creating new objects to store, then explicitly casting back to the wrapper, and extracting the primitive value, on retrieval. The advantage is, that the Java compiler knows the Stack is of type Integer, and is be able to check that the variables, pushed in or popped out, are of type int or Integer. Stick to ordinary arrays for numerical code that is performance sensitive.
7.5 Varargs

Varargs allows the argument of a method to be an array of type `Object`. A variable number of arguments can then be provided to the method, each argument being stored as one object in the array, with primitives autoboxed. The operation is achieved by using three dots (…) to indicate the array, see the following code.

```java
static public void varargs(Object... array){
    for(Object o: array)
        System.out.println(o.toString());
}
varargs(7, "you", new Double(8.9));
```
A package is a collection of classes, interfaces and subpackages. A package has an name, and its contents are only available by using its name. This allows material to be grouped in a rational way.

### 8.1 Writing & Compiling Packages

To write packages place all the classes that form the package in the same directory, which must have the same name as the package. Then add the `package` command to the top of each class. For example if I wanted to create the package `myStuff.dataBase` I would have to do the following.

1. Make the directory `myStuff` with the subdirectory `dataBase`.
2. Move all classes that are to be in the package to `myStuff/dataBase`
3. Add the line `package myStuff.dataBase;` to the start of each java class file.

A package may be compiled into a set of Java class files or, more commonly, into a Java Archive (JAR) file.

### 8.2 Using Packages

Two conditions must apply for a class, from a given package, to be used in a program,
1. The JVM must be able to find the able to find the compiled package. This can be achieved by using the -cp or -classpath options, with javac and java. Alternatively, a CLASSPATH or LD_LIBRARY_PATH environment variable can be set, containing the location of the archive.

2. The name of the class must be made available in the program’s namespace. This can be achieved in three ways:
   - Every time the class is used the whole package path must be written out. For example `java.util.Vector`.
   - The class can be imported into the programs namespace, by using an import statement at the start of the program. `import java.util.Vector;`
   - The entire package can be imported into the programs namespace using a wildcard, for example `import java.util.*;`

### 8.3 The java.lang Package

As the name implies this package contains classes that are essential to the Java language. The package provides:

- string types for handling text
- wrappers for the primitive types
- the Runnable interface and types for threading
- the Throwable class and the error and exception types.
- access to the low level features of the JVM via the Runtime and System classes.

#### 8.3.1 Strings & StringBuffers

The String class provides a means of representing a text string. The class is immutable, meaning that once an object of type String has been created, the contents cannot be changed. Methods such as toLowerCase() do not change the contents, but return a new String that contains the changed text.

StringBuffer provides a mutable container for a text string. The class provides a range of methods such as append, insert and setCharAt, which can be used to alter the text. The method toString returns the contents as a String.

Strings may be created using constructors or the static method `valueOf`. 
8. Packages

```java
// will work
char c[] = 'h', 'e', 'l', 'l', 'o';
String s1 = new String(c);
String s2 = String.valueOf(c);
```

### 8.3.2 Primitive Wrappers

Each primitive type has an immutable wrapper class in `java.lang`. The wrappers can be used if an object version of a primitive is needed, and also provide some useful methods for handling the type.

```java
String s = new String("23");
Integer myInt = Integer.valueOf(s);
int i = Integer.parseInt(s);
```

### 8.4 The `java.util` Package

The `java.util` package provides a large set of useful classes and interfaces, some of the more commonly used ones are listed below.

- **Collection** an interface implemented by all the container types in `java.util`
- **LinkedList** a class providing a linked list, storing type `object`
- **Vector** a thread-safe variable size array, for storing type `object`
- **Stack** a last in first out array, for type `object`
- **Hashtable** a table, in which key `objects`, are mapped to value `objects`
- **Enumeration** this interface provides a means of stepping through the elements in a collection. It is being replaced by Iterator, try to avoid using it.
- **Iterator** (Java 1.2) The same as Enumeration, but providing a thread-safe `remove` method.
- **Arrays** provides a lot of static methods for operations on arrays, such as filling, sorting and searching.
- **Date & Calendar** provide means of doing date arithmetic, and translating from machine time (milliseconds since midnight 1 Jan 1900), to dates and times in local time zone.
Random  a random number generator

StringTokenizer  a class that can take a string and a character called a *token*, and break the string up at occurrences of the character, then allow access to the parts one by one. This is used in text processing, particularly when writing compilers.

The containers all store their contents as the type object, which means that they can store objects of any class. Obviously an explicit cast is required to convert the objects back when they are extracted.

```java
String s1 = new String("hello");
Stack stack = new Stack();
stack.push(s1);

Object o = stack.pop();
String s2 = (String) o;
```

Enumerations and Iterators provide a means of stepping through the elements of a collection. Iterators were introduced in Java 1.2, and differ only in that they provide a thread-safe way of removing an element, while continuing to step through the collection. As a result, Iterators are the preferred interface for use with collections, but a vast amount of code still uses Enumerations.

```java
String s = "hello";
Integer i = new Integer(2);

Vector vector = new Vector();
vector.add(s);
vector.add(i);

Iterator it = vector.iterator();
while(it.hasNext()){
    Object o = it.next();
    doSomethingToObject(o);
}
```
Chapter 9

Useful Tips

Having described the main features of the java language, we can now address problems of using the language to write programs. To help with this a set of tips is provided, relating to common tasks undertaken by programmers.

9.1 Comparing Objects

When applied to a reference type, the \( == \) operator will compare the memory addresses of the objects being referred to. This is also the operation supplied by the default \( \text{equals}(\text{Object } o) \) method in \texttt{Object}. More sophisticated testing is clearly required.

For example, if a class maintains a list of Persons, it may have a method that takes a person, and checks to see if they are already in the list. In a case like this, two different Person objects containing identical data would be considered a match, and the method would return \( \text{true} \), However \( == \) would return \( \text{false} \) as they are different objects with different memory addresses. The problem may be extended if the list of Person objects has to be sorted, in which case a means of defining a \textit{greater than} operation on Persons must be found.

9.1.1 Testing Equality

Objects may be compared for equality by overriding the \( \text{equals}(\text{Object } o) \) method. An example of this is the \texttt{String} class, which defines \( \text{equals}(\text{Object } o) \) a test for an exact match of the characters in the two strings. Helpfully it also provides an \( \text{equalsIgnoringCase}(\text{Object } o) \) method, that will ignore capital letters.
Writing an `equals(Object o)` method, for a new class, requires you to work out a definition of what equality means for objects, of your new class. Assuming that the fields required for a comparison are not private, an appropriate set of if statements can be written. If the data is not accessible, you may have to provide special matches for each item, such as `.codeEquals(String s)`.

There is one important feature of the `equals(Object o)` method, that must always be addressed. It takes a parameter defined as being of type `Object`, and, as a result, you may have any possible reference type coming it to your method. If you try to cast, you risk generating a `java.lang.ClassCastException`. The solution is to use the `getClass()` method in `Object`, combined with the `instanceof` operator, see the following for some class `MyClass`.

```java
class MyClass{
    public boolean equals(Object o){
        if(o instanceof MyClass){
            MyClass m = (MyClass) o;
            // test for equality
        } else
            return false;
    }
}
```

The method first tests if `o` is an instance of type `MyClass`, returning `false` if it is not. The cast and test for equality only occurs if the test for correct type has been passed, avoiding a cast error.

### 9.1.2 Testing Magnitude

**Using Comparable**

The `Comparable` interface, in `java.lang`, requires classes implementing it to provide a `compareTo(Object o)` method. There is a generic version of this interface, which requires the `compareTo` method to have a parameter of the type named in the class declaration. This is the preferred way of using the interface as it avoids cast errors. The `Arrays` class, in `java.lang`, provides a sort function for arrays of `Comparable` objects.
public class TestClass extends Object
    implements Comparable<TestClass>{
    public final int i;
    
    public TestClass(final int i){
        this.i = i;
    }
    
    public int compareTo(TestClass t){
        if(this.i == t.i)
            return 0;
        else if(this.i < t.i)
            return -1;
        else
            return 1;
    }
}

Using Comparator

In java.util there is a Comparator<T> interface, this requires implementing classes to provide a compare(T a, T b) method, and an equals(Object o) method that compares Comparators. Comparators can be used with Arrays, but also with any Collections.sort method, on any implementation of Collections.

Note that when implementing the Comparable interface, you adding an comparison method to your object. In the case of Comparator<T>, you design a new class that carries out the comparison between two objects. You also have to provide the comparing class with an equals method that can check if other comparators are equal to itself. This may seem absurd, but it is possible for a comparator to have internal state, which could affect its behaviour. If no equals(Object o) method is provided, the default one from Object will be used.
import java.util.Comparator;

public class TestComparator implements Comparator<TestClass> {
    public int compare(TestClass t1, TestClass t2) {
        return t1.compareTo(t2);
    }

    public boolean equals(Object o) {
        if (o instanceof TestComparator) {
            return true;
        } else {
            return false;
        }
    }
}

The above code implements a trivial Comparator for the TestClass. The compare method wraps the TestClass.compareTo method, while the equals method just checks that class name. If you only need a comparator in one place, you might implement it as an anonymous class.

More complicated still are the circumstances where we wish to find all the words in a string that match a particular pattern. For example, to find all proper nouns used as possessives, we would look for all words starting with a capital letter and ending apostrophy s. This requires some code to be used to describe the match, these codes are called regular expressions.

9.1.3 Comparing Strings & Regular Expressions

Strings are commonly used to store complex data, and the various ‘meanings’ associated with the data complicate the problem of comparison. For example a list of words, could be sorted into alphabetic order. However codes for lowercase and uppercase letters are different so some policy for handling uppercase must be enforced.

Methods in String

String overrides the equals(Object o) method, and provides an additions equalsIgnoreCase(String s) method. String also implements Comparable, and so provides a compareTo(Object o) method, which can be used for sorting.

String also provides many methods for finding the first and last index of characters in the string. These can be used, with one of the substring methods to isolate parts of a string. A useful pair
of methods are the two split functions that return the arrays of strings, produced by splitting themselves using the parameter.

**Regular Expressions**

Several regular expression packages have been written for Java. The standard one that comes with the language is in the package `java.util.regex` and provides three classes: `Pattern`, `Matcher`, and `PatternSyntaxException`.

The following method illustrates the use of a regular expression. The regular expression `\w` means match a word character A-Z and a-z. Since `\` is the escape character in Java strings, we must double up the backslash forming `\\w`. Adding a + symbol to the end means match one or more characters, and adding brackets means consider the match as a group.

```java
public void printStuff(final String target){
    //a group of word characters
    final String regex = "(\\w+)";

    //obtain the Pattern object for the regex
    final Pattern pattern = Pattern.compile(regex); // obtain a pattern object

    //build a matcher for the pattern and the target
    final Matcher matcher = pattern.matcher(target); // obtain the matcher

    //match and print the regex   while(matcher.find())
        System.out.println(matcher.group());
}
```

Applied to the string "Hello world", the output would be

```
Hello
world
```