COM6516
Object Oriented Programming and Software Design

The contents of this module has been developed by Adam Funk, Kirill Bogdanov, Mark Stevenson, Richard Clayton and Heidi Christensen

1. Introduction to Java

Aims

Briefly cover basic Java constructs that are similar to other languages

Objectives

At the end of this lecture, and by studying the additional info signposted, you will know how to use the basic constructs of the Java language

1. Introduction to Java

Outline

- Structure of a simple Java program
- Variables: assignments, data types, typecasting
- Control structure
- Arrays: initialisation, copying
- Characters & Strings

A simple Java application

```
public class Welcome{
  public static void main(String[ ] args){
    System.out.println("Welcome to . . .");
    System.out.println("the Java language");
  }
}
```

public — access modifier, indicates what parts of a program can use this code

class — everything in Java programs live in classes

Welcome — class name

- must begin with a letter, case is important
- should be same as file name (**Welcome.java**)

Classes contain methods, in this case there is one method — **main**

Methods are enclosed by braces { }, and each statement is terminated with a semicolon

Bracket styles and indentation should be consistent

Variables and assignment

```
public class DistanceConverter{
  public static void main(String[ ] args){
      double distanceInMiles;
       distanceMiles = 10.0;
       double distanceInKm = distanceInMiles * 8/5;
       System.out.println("Distance is " + distanceInMiles + " miles
(" + distanceInKm + " km)");
  }
}
```

Variables in Java have a `name` and a `type` (Java is a strongly type language)

Variables are declared by a `type <variable name>`; statement

The operator = assigns a value to a variable

Can declare and initialise a variable on the same line anywhere in a program

Primitive data types

In Java, *primitive types* store a data value, whereas *object types* store a reference (memory address) to a data value

Type	Size	Description	Range	Default
boolean	1 bit		true or false	FALSE
byte	8 bits	Signed integer	[-128, 127]	0
short	16 bits	Signed integer	[-32,768, 32,767]	0
char	32 bits	Unicode character	[‘\u0000’, ‘\uffff’] or [0, 65535]	\u0000’
int	32 bits	Signed integer	[-2,147,483,648 to 2,147,483,647]	0
long	64 bits	Signed integer	$[-2^{63}, 2^{63}-1]$	0
float	32 bits	Floating point	32-bit IEEE 754 floating-point	0.0
double	64 bits	Floating point	64-bit IEEE 754 floating-point	0.0

Type promotion and casting

Converting variables from one type to another

Arithmetic expressions can mix types, but beware (see ***TypeCast.java***)

```
…
int int1 = 4;
int int2 = 5;
int average = (int1+int2)/2;
                                    // average = 4 (all operands are integer)
double average = (int1+int2)/2;
                                    // average = 4.0 (all operands are integer)
double average = (int1+int2)/2.0;
                                    // average = 4.5
                                    // one floating point operand
                                    // -> all operands converted to floating point
```

Type promotion and casting

“Smaller” types are promoted to “larger” types

Sometimes a manual cast is required

```
…
int int1 = 4;
int int2 = 5;
double average = (double) (int1 + int2) /2;
double average = (int1 + int2)/(double) 2;
```

Operators for primitive types

The following operators are commonly used with primitive Java types

Operator	Operand type	Operation performed
=	primitive	variable assignment
==	primitive	equals
!=	primitive	not equal
+,-	arithmetic	addition, subtraction
++,—	arithmetic	increment, decrement
*,/,%	arithmetic	multiplication, division, remainder
!	boolean	logical complement
&&,||	boolean	logical AND, OR
>,>=	arithmetic	greater than; great than or eqaual
<,<=	arithmetic	less than; less than or equal

Enumerated types

Java 5.0 and later version allow definition of enumerated types

These are useful where a variable can only hold a restricted set of values

For example, a `Size` variable may be restricted to the values SMALL, MEDIUM, LARGE and EXTRA_LARGE

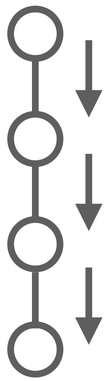
```
enum Size {SMALL, MEDIUM, LARGE, EXTRA_LARGE};
Size s = Size.MEDIUM;
```

Control structure in Java

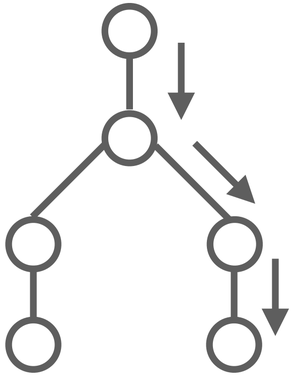
Flow of control — the way that Java moves from one statement to the next

- Sequence: simply do one statement after the next
- Selection: flow of control is determined by a simple decision
- Repetition: execute the same set of statements more than once

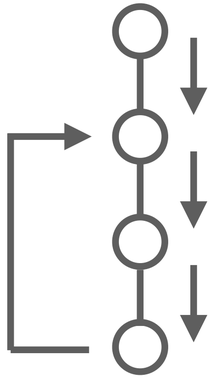
Sequence



Selection



Repetition



Selection

Conditional statements involve Boolean expressions that can be either true or false (a so-called binary decision). The action performed depends on the value of the expression.

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

where a block is a sequence of statements, sometimes called a compound statement.

Conditional statements perform a Boolean test using a relational operator (<, <=, ==, !=, =>, >):

```
if(t < worldRecord) {
    worldRecord = t;
    System.out.println("New world record! " + t + "s");
}
else
    System.out.println("Better luck next time...");
```

Precedence

More complicated expressions can be constructed using the logical operators: and (&&), or (||), not (!) and equals (==, !=).

The following are equivalent:

```
if ((x>2000) || (x>0) && (y>0))

if ((x>2000) || ((x>0) && (y>0)))
```

Precedence

Precedence rules apply to the operators used in a logical expression

++ has a higher precedence than >=, so is evaluated first

Evaluated first

```
int a = 2;
if (++a >= 3)
     System.out.println("a >= 3, a = " + a);
else
     System.out.println("a < 3, a = " + a);
```

```
int a = 2;
if (a++ >= 3)
     System.out.println("a >= 3, a = " + a);
else
     System.out.println("a < 3, a = " + a);
```

Evaluated after

Selecting alternatives

Java provides a switch statement for selecting one of many alternatives when testing the same variable or expression

Example: a vending machine computing the value of coins deposited based on their weight:

```
int credit = 0;
int weight = keyboard.readInt();
switch (weight) {
  // must be char or whole number
    case 35: credit += 50; break;
    case 19: credit += 20; break;
    case 16: credit += 10; break;
    case 9 : credit += 5;  break;
    case 7 : credit += 2;  break;
    case 3 : credit += 1;  break;
    default: screen.println("Unknown coin!");
 }
// more code
```

`Break` transfers control to the end of the `switch` block

Repetition — for loops

The **for** statement – a ‘counting loop’

Know (or can calculate) in advance how many times we wish to execute the loop.

```
for (statement1; expression1; expression2)
  {code block}
```

The **for-each** statement enables you to traverse each element of a collection – e.g. an array. Syntax is

```
for (variable : collection)
  {code block}
```

```
  double[] arrayData = . . .;
  double sum = 0.0;
  for (double elem : arrayData){
      sum += elem;
  }
```

Repetition — while loops

The while statement – a conditional loop

- Repeatedly execute the loop while the condition is **true**
- This is an indeterminate loop since it is not known how many times a loop should be processed in advance
- Use with care!

```
while (condition) {code block}
```

The **do…while** statement – a conditional loop

- Test at the end of the loop
- Always executed at least once

```
do {block} while (condition);
```

Nesting loops

Since a `for` loop is a kind of statement, it can be included as one of the statements in a `for` loop body — this is called a nested loop

```
public class NestedLoopExample {
   public static void main(String args[]) {
      int z;
      for (int x=0; x<10; x+=1) {
         for (int y=0; y<10; y+=1) {
            z = x * y;
            System.out.print(x + "*" + y);
            System.out.print( "=" + z);
         }
      }
   }
}
```

Arrays

An array is a data structure to store a collection of values of the same type
(e.g.) an array of `int`s has type `int[]`

An individual value of array `a` is accessed via an integer index (e.g.) `a[10]`

The first element of an array has *index* 0

The length of array a is given by `a.length`

Arrays

The following creates an area of memory, which is *referenced* by the array variable `a`:

```
int[] a = new int[100];
```

Loops are natural partners for arrays:

```
for (int j=0; j<100; j++)
     a[j] = j*2;

for (int j=0; j<a.length; j++)
     System.out.println(a[j]);
```

Arrays can have more than one dimension:

```
int[][] grid = new int[3][3];
```

Initialising arrays 1

An array can be initialised when declared using a *literal array expression*:

```
int[] b = { 2, 3, 5, 7, 11, 13, 17, 19};
```

Or you can reinitialise an array using an *anonymous array*:

```
int[] b;

b = new int[] {1, 1, 2, 3, 5, 8, 13, 21, 34};
```

Right-hand side is created in memory and then immediately assigned to existing variable ~ *anonymous*

Initialising arrays 2

An array can be initialised in a loop:

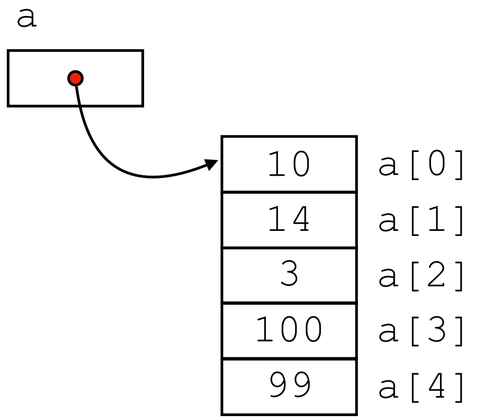
```
int[] b = new int [5]
for (int i=0; i<b.length; i++) {
   b[i] = i*i;
}
```

Copying arrays 1

Remind ourselves what an array variable is: a *reference* to memory space.

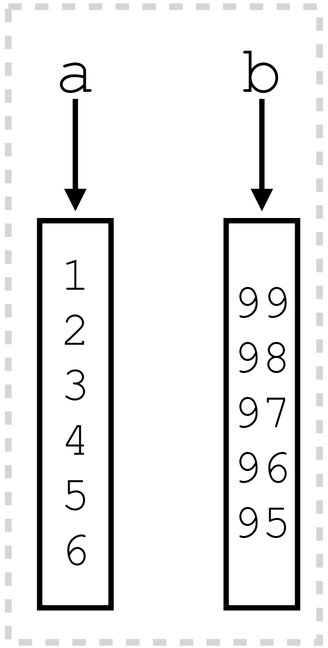
When an object is created by using “new”, a memory space is allocated and a reference i returned.

```
int a[] = {10,14,3,100,99};
```



Copying arrays 2

```
int a[] = {1,2,3,4,5,6;
int b[] = {99,98,97,96,95};
a = b;
```

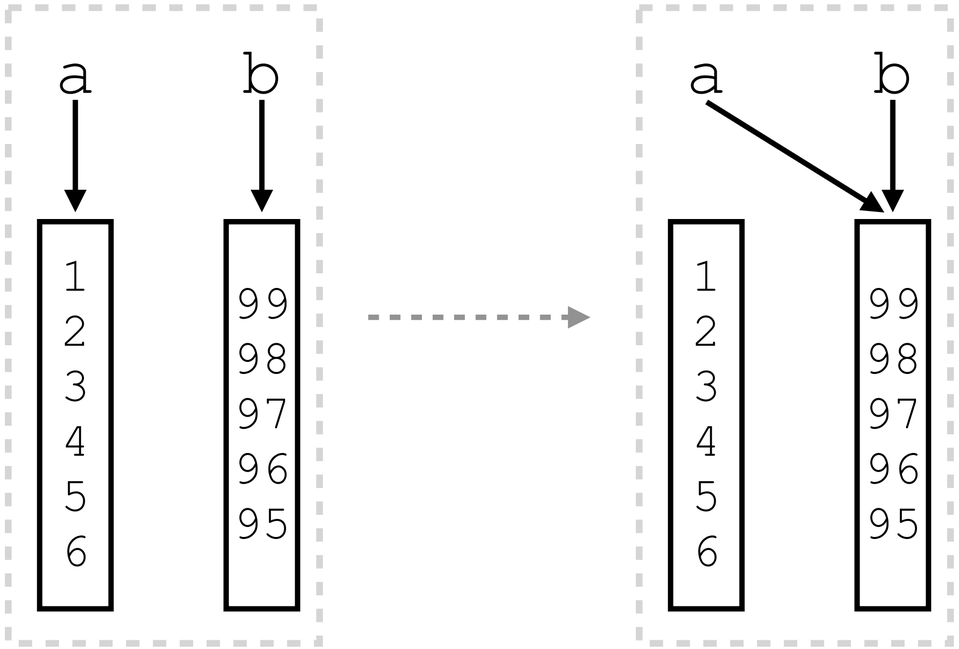


`a` is now pointing to the same part of the memory as `b`; not always what we intended

Copying arrays 3

Simple copying leaves both variables pointing to the same array

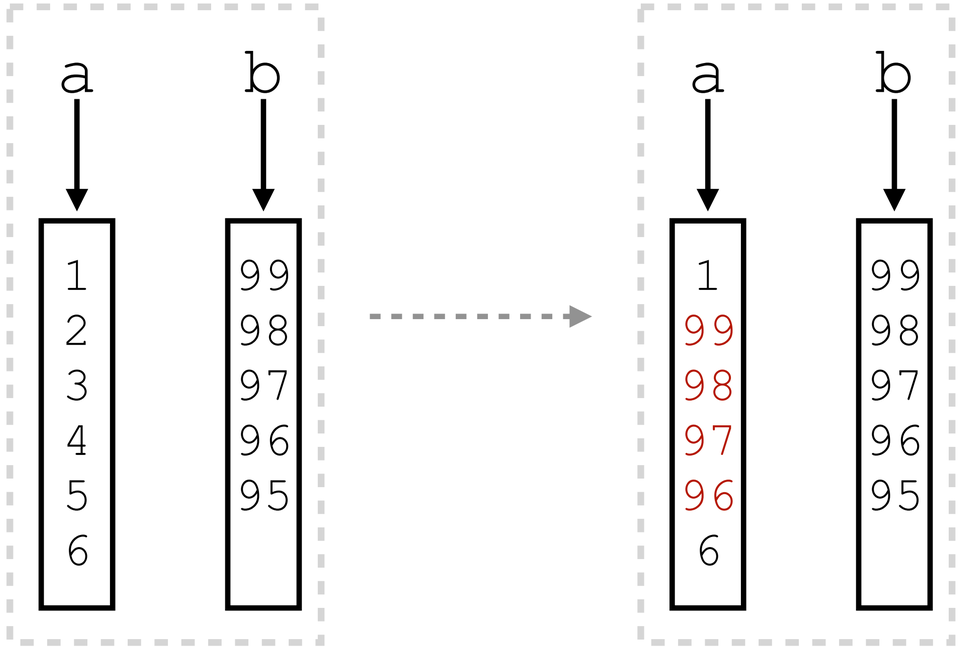
```
int a[] = {1,2,3,4,5,6;
int b[] = {99,98,97,96,95};
a = b;
```



Copying arrays 4

Use `System.arraycopy` to copy value-by-value

```
System.arraycopy(src, srcPos, dest, destPos, length);
```




```
System.arraycopy(b,0,a,1,4);
```

Characters and Strings 1

The `char` type represents a single Unicode character. Unicode is an extension of ASCII, with 65536 (FFFF) characters.

```
char myChar = 'E';

char myChar = '\u03c0'; //hexadecimal 03c0 is 'π'
```

Characters and Strings 2

The `String` type is a class, not a primitive type, so methods can be used to operate on Strings.

However, `String` is not a typical class because strings can be *initialised by assignment*:

```
String greeting = “Hello”;
```

The + operator can be used to concatenate strings; Java handles conversion of non-string types to String where appropriate, e.g. if the variable area is an `int`, then it will be converted to `String` by

```
System.out.print("Your circle has an area of "+area+" metres squared.”);
```