Introduction to Java

Table of Contents

These notes show how some Computer Science ideas are used in Java. It does not explain the theory behind these ideas - read the other notes for that.

Procedural programming

Setup

You may be a student at some institution, in which case college machines will already be set up to write Java software.

If not, and you need to set up your own device, the first step is to install a Java Development Toolkit - a JDK. You need the correct version for your operating system (Windows Linux, Mac or whatever) and processor, 32 or 64 bit. One option is:

https://www.openlogic.com/openjdk-downloads

You can check the installation by going to the *command line* in a suiable folder and typing:

```
javac -version
```

and you should get a response similar to (but maybe a different version):

```
javac 1.8.0_275
```

This runs the Java compiler and tells you the version number.

If you are unsure about using the command line on your operating system, now is a good time to learn the basic commands.

Then open a text editor and copy and paste this program:

```
class Test {

  public static void main(String[] args)
  {
    System.out.println("This is a test");
  }

}
```

This should be in a text editor, such as notepad on Windows or gedit on Linux. It should not be a word processor, because that would add format data, and we want pure text only. Save this with filename Test.java (not test.java or anything else).

Go back to the command line, and compile the program like this:

```
javac Test.java
```

You should get no output. If you get an error message, you did not follow these instructions. Try again.

Then run the compiled code by

```
java Test
```

You should get the output:

```
This is a test
```

The use of a compiler is explained later.

Test program skeleton

```
class Test {

  public static void main(String[] args)
  {
  .. new code goes here ..

  }

}
```

This is our skeleton program. We can do a lot just changing what we have in main(). We explain class and so on in later sections.

Variables and primitive types

```
class Test {

  public static void main(String[] args)
  {
  int num1;
  num1=-38;
  System.out.println(num1);
  long num2=987654321;
  System.out.println(num2);

  float num3=12.45f;
  System.out.println(num3);
  double num4=0.3333333333333333333333333333;
  System.out.println(num4);

  }

}
```

A variable is a named value which can change as the program runs.

A data type is a type of data value.

In Java we need to declare variables before they are used. This means telling the compiler their name and type.

```
  int num1;
```

declares num1 to be a variable, with type int.

We assign values to variables like this:

```
  num1=-38;
```

We can output values to the console like this:

```
  System.out.println(num1);
```

We explain System.out later.

Test

1. Read this program. What do you expect the output to be?
2. Copy and paste it into the text editor. Save it. Compile it. Run it. Is the output what you expected?
3. Alter the program in the editor to make num1 99. Save it. Compile it. Run it. If you get an error message, read it and fix it.
4. Make num4 7.2. Save, compile and run.

Java has two kinds of types - *primitives* and *reference types*.

Primitives are simple values. References refer to *objects*, which are bundles of code and data. We learn about primitive types first.

The primitives in this program are types of number. An *int* is an integer, and so is a *long*. An int is stored in 4 bytes of memory, while a long is held in 8 bytes. This means the range - maximum and minimum - of a long is greater than an int. We usually use int, and only exceptionally use long.

The *float* and *double* type are non-integer numbers. A float is held in 4 bytes, while a double is in 8 bytes. A double has greater range, and also better precision. This is because arithmetic with float and double is not totally accurate.

Check the f in

```
  float num3=12.45f;
```

12.45f tells the compiler the 12.45 is a float value in 4 bytes. If we had said

```
  float num3=12.45;
```

then 12.45 is a double value, in 8 bytes, and would not fit in the 4 bytes of a float. The compiler would have reported the error.

Character and boolean types

```
class Test {

  public static void main(String[] args)
  {
  char myChar;
  myChar = 'a';
  System.out.println(myChar);
  boolean myBool;
  myBool = true;
  System.out.println(myBool);

  }

}
```

Test

1. Read this program. What output do you expect?
2. Copy and paste it into the editor. Save it, compile it and run it.
3. Change myChar to 'θ' (copy and paste it). Change myBool to false. Compile and run.

Two more primitive types are *character* and *boolean*.

The character type is named *char*. A character is anything on the keyboard, and many more. Java uses a character set called *Unicode*, which has millions of different characters (but not all fonts can display all characters).

The boolean type has just two possible values - true and false. Read about logic if this is unfamiliar.

An *identifier* is a name chosen by the programmer. The identifiers in this program are named myChar and myBool. Identifier names should be chosen with care, to reflect what they represent. In these programs our identifiers just represent primitive types.

Identifiers should start lower case unless they are the names of classes or interfaces. They cannot have spaces in, so we usually use *camel case*, starting new words with upper case, like thisIsAnExample.

Expressions and assignments

```
class Test {

  public static void main(String[] args)
  {
  int x,y,z;
  x=2;
  y=3;
  z=x+y;
  System.out.println(z);

  }

}
```

Something like x+y is an *expression*. It uses *operands* like x and y, and *operators* like +. Other operators are -, * for multiplication and / for divide. An *assignment statement* gives a variable the value of some expression.

The operators have *precedence levels*, that controls the order in which they are used. They match normal mathematics, so 2+3*4 is 14, not 20. We can use round brackets, so (2+3)*4 is 20.

If both operands have type int, then *int division* is used. For example 7/2 is 3. This is because 7 divided by 2 is 3 remainder 1, and int division discards the remainder. 14/3 is 4.

But 7.0/2.0 is 3.5. Here the operands are type double, and double division works as expected.

Test

- What do you expect the ouput of the program above to be?
- Run it to find out.
- Change it to work out 2+3*4
- Change it to work out (2+3)*4

Comments

```
/*
File name Test.java
Author: W Milner
Purpose: Illustrate comments

*/
```

```
class Test {

  public static void main(String[] args)
  { // Execution starts here
  // This is a line comment
  // This very simple program does not really need
  // comments
  int x,y,z;
  x=2;
  y=3;
  z=x+y;
  System.out.println(z);
  }
}
```

A comment is text ignored by the compiler. It is intended to say what the program is supposed to do, who wrote it, when, what version it is, explaining how it works and so on.

In Java a line comment starts // and goes to the end of that line.

A block comment starts /* and ends */ and cover several lines.

Do not use comments to say obvious things, so

```
  z=x+y; // add x and y to get z
```

is wrong - the comment is a distraction and adds nothing.

Test

1. Change the comments and run the program

Compound assignments and increments

```
class Test {

  public static void main(String[] args)
  {
  int x;
  x=5;
  x+=2;
  x-=3;
  x*=2;
  x/=4;
  System.out.println(x);
  x++;
  System.out.println(x);
  }
}
```

We often want to do things like increase a variable by, say, 2. We can do that by saying

```
x = x+2;
```

which means take the current value of x, add 2, and store this as the
new value of x.

But another way to say the same thing is

```
x+=2;
```

This is better, because

- It is less typing
- It executes faster
- It makes more sense (to someone who knows Java)

In a similar way

```
x*=2;
```

is the same as

```
x = x * 2;
```

We very often want to increase the value of a variable by 1
(increment it) and

```
x++;
```

does that. Similarly

```
x--;
```

reduces x by 1.

Test

1. What do you expect the output of the program above to be (you may need pencil and paper).
2. Run it and see.

Casting and ranges

```
class Test {

  public static void main(String[] args)
  {
  double d;
  d=5.0;
  int x;
  // x=d; will not compile - "possible lossy conversion
  // from double to int"
  x = (int) d;
  System.out.println(x);
  byte b;
  b = (byte) 127;
  b++;
  System.out.println(b); // -128


  }
}
```

We can assign a 'narrow' type to a wider type with no problem - for example

```
x = y;
```

is fine if y is an int and x is a long, because the 4 bytes of an int will fit into the 8 bytes of a long.

But assigning a wide to a narrow type, like

x = d;

is a problem, if d is a double (8 bytes) and x is an int (4 bytes). The compiler can generate code to do the conversion - but there will be some data loss.

Changing type is called a *cast*. We can force a type cast like this:

```
  x = (int) d;
```

This signals to the compiler that we know there will be data loss, but it does not matter (in this case, double 5.0 has the same value as int 5).

Any type has a maximum value which can be stored in it. A byte is a Java primitive integer stored in 1 byte. These are represented in 2s complement, so possible values range for 0111 1111 = +127 to 1000 0000 = -128.

What happens if we exceed this *range*? Set a byte to be +127, then increment it? We get 1000 0000 = -128. This is called overflow. Notice there is nothing to signal in the program shown that *overflow* has happened and the result is wrong. This means data types need to be chosen with care.

Test

1. Run the program above
2. A short is another Java integer type, stored in 2 bytes, in 2s complement. What is the maximum value of a short?

Boolean expressions

```
class Test {

  public static void main(String[] args)
  {
  boolean plusFlag;
  int x;
  x=5;
  plusFlag = x > 0;
  System.out.println(plusFlag);
  int y;
  y=-8;
  plusFlag = x > 0 && y > 0;
  System.out.println(plusFlag);
  }
}
```

An expression like 2+3 has a value which has type int. Some expressions have a value which is true or false - in other words, boolean type.

For example, x > 0 will be true if x is greater than 0, and otherwise false.

We can assign that result to a variable of type boolean.

Why name it plusFlag? Boolean variables are often called flags. A flag is raised, meaning, yes or true, or lowered, meaning no or false. This flag is about whether something has a plus value or not, and hence plusFlag.

We can also use boolean operators like &&, which means logical 'and' - in other words both. The others are || for 'or' and 1 for 'not'.

Test

1. What output do you expect from this program?
2. Run it.
3. Change it so it outputs true if either x is negative or y is positive. Test it.

What is Java?

It is a general purpose programming language. That means a very wide range of program types can be written in Java. It is not special purpose.

It is a high level language. That means it is not assembler or native code or machine code. It is about some problem to be solved, not about the processor it is running on.

It works on many platforms - that is, operating system and processor combinations. The same Java code will run the same on Windows, Linux and Mac, on Intel AMD Motorola and ARM processors.

It is compiled to bytecode. A compiler (javac) is software which translates Java code to an equivalent program in a language called bytecode. This is what

```
javac Test.java
```

does.

The compiler outputs the bytecode in a class file - in this case, named Test.class:

```
 ls *.class
Test.class
```

The application launcher starts the Java Virtual Machine (JVM), loads the class file, and starts the bytecode running on the JVM. This is what

```
java Test
```

is doing.

Why? Different platforms are different. But every JVM looks the same. The JVM makes the platforms seem the same, so that the same bytecode will run on all of them.

The gain of this is that Java is cross-platform. The main reason for using Java is that it is cross-platform.

This comes at a cost. For example the JVM does not have a serial port, so pure Java cannot access it (it can using something called JNI - Java Native Interface, but then this is no longer pure Java).

Applications needing to access hardware directly are better written in languages compiled to native code, such as C.

Conditional statements

```
class Test {

    public static void main(String[] args) {
        int x;
        double randomNumber = Math.random();
        if (randomNumber < 0.25) {
            x = 2;
        } else {
            x = 3;
        }
        System.out.println(x);
    }
}
```

A conditional statement is an *if statement*.

We use it so that different statements will be executed in different situations. The program has a boolean expression to evaluate. If this is true, some statements run. If not, others are done.

In effect the computer makes a decision.

Compile-time is when the program is being written and compiled.

Run-time is when the compiled form is being executed.

The decision is taken at run-time.

In this program we use Math.random(). This calls a library routine which returns a random double in the range 0 to 1. So it has a 25% chance of being less than 0.25. If it is, x is 2. If not, x is 3. At compile-time we do not know what will happen. When we run it, 1 time in 4 x is 2, and 3 times out of 4 it is 3. More about Math later.

Often the boolean expression involves the results of other computations.

Test

1. Run this program several times and see what the output is
2. Change it so it outputs 2 half the time, and 3 the rest.

while loops

```
public class Test {

    public static void main(String[] args) {
        // while loops
        int n = 20; // initialisation
        while (n < 25) {    // ( boolean expression )
            n += 2;
            System.out.println(n); // 22 24 26
        }                   // end loop body

        // do while - test at end:
        n=1;
        do {
            n*=2;
            System.out.println(n); // 2 4 8 16 32
        } while (n<32);

    }
}
```

Java has several ways of writing loops to achieve *iteration*.

One is while() {..}.This tests the condition at the start of the loop body, and so may repeat zero times.

Another is do { .. } while (), which tests at the end of the loop body, and so executes the body at least once.

Test

1. Run this program

2. Write a program which outputs all multiples of 3 between 3 and 99 inclusive.

for loops

```
public class Test {

    public static void main(String[] args) {
        // for loops
        // very  simple:
        for (int i=0; i<5; i++)
        {
            System.out.println(i);
        }
        // i=5; Syntax error - i is out of scope here
        // add up multiples of 3 from 3 to 9 inclusive
        int total;
        total=0;
        for (int n=3; n<12; n+=3)
        {
            total+=n;
        }
        System.out.println(total); // 18
    }
}
```

The syntax of a for loop in Java is:

for (loop header) { loop body }

The loop header has 3 parts, separated by semi-colons:

(initialisation ; repeat while this is true; change at every loop end)

The initialisation can include a variable declaration.

In this example -

```
int i=0;
```

is the initialisation

```
i<5;
```

repeat while i is less than 5

```
i++
```

add 1 to i each time around

Test

1. Run this program
2. Write a program to add up the even integers from 2 to 1000 inclusive. How can we test it?

Nested loops

```
public class Test {

    public static void main(String[] args) {
        // nested loops
        for (int i = 1; i < 4; i++) {
            for (int j = 1; j < 4; j++) {
                System.out.println("i=" + i + " j=" + j);
            }
        }
    }
}
```

The statements inside a loop can be any type - including loops.Then we have a loop inside a loop - a nested loop.

Note for the indentation helps us make sense of the code.

Check carefully the println being used to output several things. The + concatenates strings, meaning they are joined together. The i and j (which are ints) are cast to strings before being concatenated.

Test

1. Try to predict the output of this program
2. Run it and check
3. Suppose a loop which repeats 5 times contains a loop which repeats 4 times. How many times is the inner loop body repeated?

Methods

```
public class Test {

    static double average(double x, double y) {
        double result;
        result = (x + y) / 2;
        return result;
    }

    public static void main(String[] args) {
        double a, b, c;
        a = 4.0;
        b = 5.0;
        c = average(a, b);
        System.out.println(c);
    }
}
```

Java uses the term 'method' for what is called in other languages a sub-routine, a function or a procedure.

main is a special method - it is the one where execution starts.

A method header must say the type of the returned value of a method. Hence 'double average'.

A method which does not return any value has type 'void'. Hence 'void main'.

The parameter list of a method is in round brackets and must say the type of the parameters. For example 'average(double x, double y)'

static is explained later.

Test

1. What do you expect the output of this program to be?
2. Run it
3. Why should average return a double, and not an int?
4. Suppose in this program it changed the value of y inside method 'average'. Would that change the value of b in main? Change the code and run it to find out.
5. Write and test a method which finds the average of 3 numbers.

Arrays

```
public class Test {

    public static void main(String[] args) {
        // arrays
        int[] arr; // declare the type of variable arr
        arr = new int[5]; // create array and assign it to arr
        arr[0]=99;
        arr[1]=-7;
        System.out.println(arr[0]);
        System.out.println(arr[2]);
        int[] another = {8,2,3,1}; // use an 'array literal'
        System.out.println(another[2]);
    }
}
```

As in most languages, array indexes start at 0, not 1.

The JRE checks array indexes at runtime, and will throw an exception if code accesses an element outside the array bounds.

Test

1. Run this program
2. Write a program which creates an array of 1000 ints and fills it with 99 in every element.
3. Write a program which creates an array of 5 characters, sets them to be 'a','x', 'p', 'z' and '=', and outputs them all.

Searching

```
public class Test {

    public static void main(String[] args) {
        // linear search
        int[] data = {5, 3, 7, 9, 2, 3, 88, 34, 56, 21, 96};
         // data has 11 elements
        int target = 9;  // value to search for
        int where = 0; // start at the beginning
        while (where < 11 && data[where] != target) {
         // while not hit the end, or find target
            where++; // move on
        }
        if (where == 11) // if hit end
        {
            System.out.println("Not found");
        } else // we found it
        {
            System.out.println("Found at " + where);
        }
    }
}
```

This is a linear search. && means 'and', and != is not equal to.

Test

1. Run this program
2. How can we test it for the case where the target is not present in the data?

```
public class Test {

    static int search(int[] data, int target) {
        int where = 0; // start at the beginning
        while (where < data.length && data[where] != target) {
            where++; // move on
        }
        if (where == data.length) // if hit end
        {
            return -1;
        } else // we found it
        {
            return where;
        }
    }

    public static void main(String[] args) {
        int[] data = {5, 3, 7, 9, 2, 3, 88, 34, 56, 21, 96};
        int result = search(data, 9);
        if (result == -1) {
            System.out.println("Not present");
        } else {
            System.out.println("Found at " + result);
        }

    }
}
```

This version separates out the search code into a distinct method.

In Java an array has a field named length, so that data.length is the length of the array data. If this is 10, the array has elements with index 0 to 9.

Note we use == to test if the values are equal. We use = for assignment.

Test

1. Run this program
2. Test it works if the target is not present
3. Test it works with an array of different length

Sorting

```
public class Test {

    // bubble sort
    static int[] sort(int[] data) {
        for (int times = 0; times < data.length; times++) {
            for (int i = 0; i < data.length - 1; i++) {
                if (data[i] > data[i + 1]) { // then swap them
                    int temp = data[i];
                    data[i] = data[i + 1];
                    data[i + 1] = temp;
                }
            }
        }
        return data;
    }
    public static void main(String[] args) {
        int[] data = {5, 3, 7, 9, 2, 3, 88, 34, 56, 21, 96};
        data = sort(data);
        for (int i = 0; i < data.length; i++) {
            System.out.println(data[i]);
        }

    }
}
```

Java has library routines to sort arrays, but here we show a bubble sort algorithm.

Test

1. We cannot swap variables x and y by saying

```
x=y;
x=x;
```

Why not?

2. Run this program

3. Change it so it works on an array of doubles not ints.

4. Change it so it works on an array of 100 doubles. Use Math.random() to initialise it with random values.

5. Change it so it sorts an array of 1000 doubles

6. Why is bubble sort not usually used?

2D arrays

```
public class Test {

    public static void main(String[] args) {
        int[] row1 = {1,2,3};
        int[] row2 = {21,22,23};
        int[] row3 = {31,32,33};
        int[][] matrix={row1, row2, row3};
        System.out.println(matrix[1][2]);
        // often use nested loops
        for (int i=0; i<3; i++)
        {
            for (int j=0; j<3; j++)
            {
                System.out.println(matrix[i][j]);
            }
        }
    }
}
```

If the elements of an array are themselves arrays, we have a two dimensional array, as here.

Test

1. Run this program
2. Add code to exchange the first and second rows. Test it.

Recursion

```
public class Test {

    static int factorial(int n)
    {
        if (n==1) return 1;
        // check - no else needed
        return n*factorial(n-1);
    }

    public static void main(String[] args) {
       System.out.println(factorial(4));
    }
}
```

The factorial function is written as !, so factorial 4 is 4!. It is the product of all integers down to 1, so 4! = 4 X 3 X 2 X 1 = 24

Test

1. Run this program
2. What would happen if you asked for a factorial of a negative value, say factorial(-4) ? Run it to confirm.
3. Change the code to stop this.

OOP - Some standard classes

StringBuilder

```
public class Test {

    public static void main(String[] d) {
        // call constructor with 'new'
        StringBuilder str = new StringBuilder("abc");
        // invoke some methods..
        str.append("CAT");
        System.out.println(str);  // abcCAT
        str.delete(1, 3);
        System.out.println(str); // aCAT
        str.insert(3, "XYZ");
        System.out.println(str); // aCAXYZT
        str.reverse();
        System.out.println(str); // TZYXACa
    }

}
```

The Java standard library has many hundreds of classes which we can use.

The StringBuilder class models a string of characters.

Java class names start Upper Case, to make it easier to see what are classes are what are not.

We make an *object*, as an *instance* of a class, by invoking a *constructor*, using the keyword *new*, as in

```
    StringBuilder str = new StringBuilder("abc");
```

Objects have methods, bundled into the object. We invoke a method (make it run) by saying the object named, a dot, then the method name - like

```
        str.append("CAT");
```

This means the append method of the str object runs, with the parameter "CAT". This method adds the string "CAT" to the end of str.

How do we know the thousands of methods of the hundreds of classes? By reading the Java API:

https://docs.oracle.com/javase/8/docs/api/

Test

1. Run this program

2. StringBuilder has a method setCharAt which places a character at a given place in a string. Find the method in the API (https://docs.oracle.com/javase/8/docs/api/java/lang/StringBuilder.html). Then include it in the above program to try it.

Wrapper classes

```
public class Test {

    public static void main(String[] args) {
      // call the constructor of Integer class
      Integer myInt1 = new Integer(7);
      Integer myInt2 = new Integer(3); // another one
      int result = myInt1.compareTo(myInt2);
      System.out.println(result); // 1
      System.out.println(myInt1.equals(myInt2)); // false
      // convert to a primitive
      int primitive = myInt1.intValue();
      System.out.println(primitive); // 7
    }
}
```

A *wrapper class* is a class that contains something else, sometimes a primitive. So the class named Integer wraps a primitive type, int. We can construct an Integer from an int:

```
      Integer myInt1 = new Integer(7);
```

The Integer class is *immutable*. That means once we construct an Integer object, it cannot be changed. This makes sense - you cannot change 7.

So you cannot do arithmetic with an Integer. It must first be changed to a primitive int.

Test

1. Run this program

2. Another wrapper class is Double, which wraps a primitive double. Look at Double in the API. Change this program so it uses Doubles not Integers.

ArrayLists

```
import java.util.ArrayList;

public class Test {

    public static void main(String[] d) {
        // construct a list
        ArrayList<Integer> myList = new ArrayList<>();
        // insert some elements
        myList.add(new Integer(9));
        myList.add(new Integer(10));
        myList.add(new Integer(12));
        myList.add(new Integer(14));
        // fetch some
        Integer myInt = myList.get(0);
        System.out.println(myInt); // 9
         myInt = myList.get(2);
        System.out.println(myInt); // 12
        // get them all
        for (int i=0; i<myList.size(); i++)
        {
            System.out.println(myList.get(i)); // 9 10 12 14
        }
    }
}
```

An ArrayList is a class which is similar to an array, but we can add and remove elements after it has been created.

ArrayList is a class in the Java Collections framework, which models the standard data structures.

The ArrayList class is in the package java.util, and we need to tell the compiler where to find it with an import statement:

```
import java.util.ArrayList;
```

An ArrayList should be restricted to containing elements of a given type, so the constructor says:

```
        ArrayList<Integer> myList = new ArrayList<>();
```

Here <Integer> is a type parameter. It means we can only put instances of the Integer class into myList. In turn that means if we copy an object out of myList, we can be sure that it will have type Integer.

The ArrayList class has methods to insert new elements at the end, or at some specified position We can also search for items, remove them, and so on. Read the API for all details.

Test

1. Run this program

2. Add and remove more Integers. Try adding a StringBuilder and see what happens.

3. Write a static method getTotal, returning an int, which finds the total of an ArrayList of Integers. Test it.

For-Each loop

```
import java.util.ArrayList;

public class Test {

    public static void main(String[] d) {
        // construct a list
        ArrayList<Integer> myList = new ArrayList<>();
        myList.add(new Integer(9));
        myList.add(new Integer(10));
        myList.add(new Integer(12));
        myList.add(new Integer(14));
       for (Integer myInt : myList)
       {
           System.out.println(myInt); // 9 10 12 14
       }
    }
}
```

Some classes are iterable. If they are, we can use the for-each loop on them (sometimes called the enhanced for loop)

```
       for (Integer myInt : myList)
```

In the loop body, myInt is each element from myList, in sequence.

Test

1. Run this program

2. Change it to an ArrayList containing Characters. Add some Character instances, them use a for-each loop to output them.

HashMap

```
import java.util.HashMap;

public class Test {

    public static void main(String[] d) {
        HashMap<Character, Integer> myMap = new HashMap<>();
        myMap.put('c', 276);
        myMap.put('j', 234);
        myMap.put('x', 999);
        System.out.println(myMap.get('j')); // 234
        System.out.println(myMap.containsKey('a')); // false
    }
}
```

HashMap is another class in the Collections framework. It is an implementation of the *map* ADT, made using a *hash table*.

As a map, it is a set of key and value pairs. Given a key, we can fetch a corresponding value.

We need to supply 2 type parameters. So for example

```
        HashMap<Character, Integer> myMap = new HashMap<>();
```

means this map has keys of type Character, and values of type Integer.

This code uses a feature called *autoboxing*. For example in

```
         myMap.put('c', 276);
```

we add a pair to the map, with key 'c' and value 276. But 'c' is a primitive char, and 276 is a primitive int. Auto-boxing converts these to Character and Integer reference types for us. This is convenient so long as we understand it.

```
import java.util.HashMap;

public class Test {
    static void count(StringBuilder str)
    {
        HashMap<Character, Integer> myMap = new HashMap<>();
        for (int index=0; index<str.length(); index++)
        {
            Character c=str.charAt(index);
            if (myMap.containsKey(c))
            {
                int val = myMap.get(c);
                myMap.put(c, val+1);
            }
            else
            {
                myMap.put(c,1);
            }
        }
        for (Character c : myMap.keySet())
        {
            System.out.println(c+" "+myMap.get(c));
        }
    }

    public static void main(String[] d) {
        StringBuilder str = new StringBuilder("ddeeefghhhhkk");
        count(str);

    }
}
```

Test

1. What does this program do?

2. How does it do it?

String

```
public class Test {


    public static void main(String[] d) {
        String str = "abc ABC αβγδ";
        for (int index=0; index<str.length(); index++)
        {
            System.out.println(str.charAt(index)); // a b c and so on
            System.out.println(str.codePointAt(index)); // 97 98 99 32..
        }
        String[] words = str.split(" ");
        for (String s : words)
        {
            System.out.println(s); // abc, ABC, αβγδ
        }
        str=str.replace('a', 'x');
        System.out.println(str); // xbc ABC αβγδ
    }
}
```

String is a class that models a string of characters. It is probably the most commonly used Java class, but it is unusual.

```
        String str = "abc ABC αβγδ";
```

Here, "abc ABC αβγδ" is a *string literal* - that is, a constant value, of type String, in source code. When that is compiled into a bytecode class file, the value is actually stored in the class file, and str gets a value which is a reference to that value, when it is loaded into memory.

String has some ordinary methods. charAt returns the character at some index position. codePointAt returns an int, which is the Unicode code point of that char - which is how Java strings are stored.

But String is immutable. Methods which look like they change a String, like

```
        str=str.replace('a', 'x');
```

actually return a reference to a new String.

Test

1. Run this program

2. Write a program which counts and outputs how many vowels (a e i o u) there are in a string.

The System class

```
public class Test {

  public static void main(String[] args) {
    String str = System.getenv("USERNAME"); // walter
    str = System.getenv("SHELL"); // bin/bash
    long now=System.currentTimeMillis(); // 1611730494749
    System.exit(1);
  }
}
```

The methods in a class are usually *per object*. But *static* methods are *per class*. Instead of saying:

<object>.<method>

we say

<class>.<method>

System is an example of a class which only has static methods available. Its constructor cannot be called, so we cannot instantiate a new System object. The idea is that System models the device the code is running on. There is only one system, so making more System instances makes no sense.

The program above calls some System methods. getEnv returns an environment variable. What these are depends on which operating system is in use.

The exit method shuts down the JVM.

System has 3 members named in, out and err. These objects have type PrintStream, and are connected to the system's default input, output and error channels. These are normally keyboard, screen and screen.

A PrintStream class has a method println. This is why we say

System.out.println...

currentTimeMillis returns the current time, as Unix time - that is, the number of milliseconds (thousandths of a seconds) since midnight 1 January 1970. This can be used for timing purposes. We say something like

```
long start = System.currentTimeMillis();
.. code..
long now =   System.currentTimeMillis();
```

then now-start is how many milliseconds the code took

Test

1. Read the API of System

2. Write a program to find out how long it takes to loop 1000 times. How long for 1 million?

The Math class

```
public class Test {

  public static void main(String[] args) {
    System.out.println(Math.PI); // 3.141592653589793
    System.out.println(Math.sin(Math.PI/2)); // 1.0
    System.out.println(Math.sqrt(9.0)); // 3.0
  }
}
```

Math is another class which only has static members. It is intended as a place to put the standard mathematics library. We cannot instantiate Maths objects, and it does not make sense to hav emore than one maths.

Math.PI is a constant, and this is indicated by PI being in capitals. Math has static methods for trigonometric, exponential an dlog functiosn plus others. These mostly use double arithmetic, so have limited accuracy.

Test

1. Read the API for full details.

2. tan(π/4) = 1. What does Math say it is?

OOP - writing classes

Class definition files

```
class Person
{
  String firstName;
  String lastName;
  int age;

}
```

```
public class Test {

  public static void main(String[] args) {
    Person p1=new Person();
    p1.firstName="John";
    p1.lastName="Smith";
    p1.age=22;
    System.out.println(p1.firstName);
  }
}
```

Most Java programming means creating our own classes.

A class must be defined - we need to say what is in a class and how it works. The class definition is in a file with extension *.java*, and a with a filename matching the class name. So a class named Person must be in a file named Person.java. The class Test is in a second file, named Test.java.

The compiler compiles these into bytecode, in files with extension *.class*. So Person.java compiles to Person.class.

A typical project will have many such files. The project is usually distributed in a single *jar file*. This is a simple zip archive file containing the byte code needed, together with any resources needed, such as image files.

A project will usually have a class with a method named main(). Execution starts here.

Test

1. Copy these two classes into files with the correct names.

2. Compile them (javac Test.java. Person.java will also be compiled automatically).

3. Run it (java Test).

4. Change Test so it outputs the last name. Save, compile and run.

Class members

```
class Person
{
  String firstName;
  String lastName;
  int age;

  void describe()
  {
    System.out.println(firstName+" "+lastName+ " age "+age);
  }

}
```

A class has two kinds of members - data fields and methods. This class has 3 data fields, named firstName, lastName and age, and one method, named describe.

Class member names should start lower case, while class names start upper case.

We can access the data fields of an object, and invoke its methods, like this:

```
public class Test {

  public static void main(String[] args) {
    Person p1=new Person();
    p1.firstName="John";
    p1.lastName="Smith";
    p1.age=22;
    p1.describe();

    Person p2=new Person();
    p2.firstName="Jane";
    p2.lastName="Jones";
    p2.age=19;
    p2.describe();


  }
}
```

This members are *per object*. In other words each object has its own values for each field. So firstName is not a variable. It is an *attribute* of a Person object.

Check thsi statement in describe()

```
System.out.println(firstName+" "+lastName+ " age "+age);
```

It refers to firstName. But firstName of what? The object which is executing this method. So in main we said

```
p1.describe();
```

then its the firstName of person p1. Then in

```
    p2.describe();
```

its the firstName of person p2.

Test

1. Run this.

2. Add a method showBirthYear to the class Person. This should output the year the person was born. Use their age field, and calculate the birth year as 2021-age. Test it.

Constructors

```
class Person
{
  String firstName;
  String lastName;
  int age;

  Person(String f, String last, int a)
  {
    firstName=f;
    lastName=last;
    age=a;

  }

  void describe()
  {
    System.out.println(firstName+" "+lastName+ " age "+age);
  }

}
```

We make a class instance, a new object, using the keyword *new*.

We usually have some code to execute when a new object is created. This is doen in a special type of method known as a *constructor*.

In code, a constructor must be named the same as the class, with no return type. So the constructor of class Person must be named Person, as here.

A common use of a constructor is to assign values to some or all data fields. Then we would use the constructor like this:

```
public class Test {

  public static void main(String[] args) {
    Person p1=new Person("John", "Smith", 22);
    p1.describe();

    Person p2=new Person("Jane", "Jones", 19);
    p2.describe();

  }
}
```

Test

1. Run this code.

2. Add a field to person named 'eyeColor'. Chose a suitable type. Add it to the constructor and the describe method. Test it.

this

```
class Person
{
  String firstName;
  String lastName;
  int age;

  Person(String firstName, String lastName, int age)
  {
    this.firstName=firstName;
    this.lastName=lastName;
    this.age=age;

  }

  void describe()
  {
    System.out.println(this.firstName+" "+this.lastName+ " age "+this.age);
  }

}
```

When writing a constructor, what should we name the parameters? The obvious answer is what they stand for, as in

```
  Person(String firstName, String lastName, int age)
```

But then, how do we show the difference between firstName the parameter, and firstName the object field? We use this, which is means the object executing the code. So in

```
    this.firstName=firstName;
```

this.firstName means the object field firstname, while just frstName means the parameter.

We can also use 'this' in methods, to refer to the object executing the method. So for example in method describe(),

```
    System.out.println(this.firstName+..
```

this.firstName means the firstName field of the object executing the describe method. But usually we do not need to say 'this', because

```
    System.out.println(firstName+..
```

means the same thing.

Method return types

Suppose we add to the Person class a way to get the person's initials. We show two approaches:

```
class Person
{
  String firstName;
  String lastName;
  int age;

  Person(String firstName, String lastName, int age)
  {
    this.firstName=firstName;
    this.lastName=lastName;
    this.age=age;
  }

  String initials()
  {
    return firstName.substring(0,1)+lastName.substring(0,1);
  }

  void printInitials()
  {
    System.out.println(firstName.substring(0,1)+lastName.substring(0,1));
  }

  void describe()
  {
    System.out.println(this.firstName+" "+this.lastName+ " age "+this.age);
  }

}
```

The first version, named initials(), uses the substring method of String to get the first part of the first and last name, and returns them. This returns a String, so this method header says

```
  String initials()
```

The second version, named printInitials, is similar, but it calls System.out.println. So it does not need to return anything. In other words its return type is void, and its headed

```
  void printInitials()
```

We use the two methods in different ways:

```
    Person p1=new Person("John", "Smith", 22);
    System.out.println(p1.initials());
    p1.printInitials();
```

The initials() method is more flexible, because we can use the returned value for any purpose. We can output it - or do anything else with the returned string.

It is usually more effective to have a method return a value, then to output it. Then code which calls it can do anything it likes with the returned value.

Access control

```
class Circle
{
  private double radius;
  Circle(double r)
  {
    this.radius=r;
  }


}
```

Here we define a new class for a Circle object, with field radius. But this has an *access modifier* private.

That means we cannot refer to radius from outside the class.

If we say

```
 public class Test
{

  public static void main(String[] args) {
    Circle myCircle=new Circle(5);
    myCircle.radius=6;

  }
}
```

We get the error message at compile-time:

```
Test.java:6: error: radius has private access in Circle
    myCircle.radius=6;
```

The purpose of this is to enforce *encapsulation*. We want to be sure the data in the object can only be accessed in a controlled way.

What if we need to change the radius? Or simply find out what the radius is? We use *public getters* and *setters:*

```
class Circle
{
  private double radius;
  Circle(double r)
  {
    this.radius=r;
  }

  public void setRadius(double newR)
  {
    if (newR>0)
      radius=newR;
  }

  public double getRadius()
  {
    return radius;
  }


}
```

and use it like

```
public class Test
{

  public static void main(String[] args) {
    Circle myCircle=new Circle(5);
    System.out.println(myCircle.getRadius()); // 5.0
    myCircle.setRadius(12);
    System.out.println(myCircle.getRadius()); // 12.0

  }
}
```

The point is that setRadius *validates* any changes. If we try to say

```
    myCircle.setRadius(-3);
```

the change is ignored, so we can be sure at any time that the circle data is valid.

Test

1. Run this code

2. Add public getArea and getCircumference methods Test them.

3. Write a class named Rectangle, with private data fields width and height. Write a constructor, and public getters and setters for width, height, area and perimeter. Test it.

Comparing objects

Suppose we compare two Circle instances:

```
public class Test
{

  public static void main(String[] args) {
    Circle myCircle=new Circle(5);
    Circle another=new Circle(5);
    System.out.println(myCircle==another); // false

  }
}
```

If the both have the same radius - why do they not equal each other?

In Java if we say something like

```
    Circle myCircle=..
```

then myCircle is not exactly an object. It is a *reference* to an object. That is, a pointer the system uses to find the Circle object that myCircle refers to.

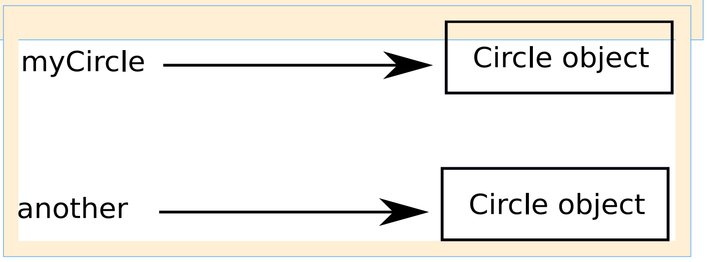
If we say

```
   ..=new Circle(5);
```

that causes a new Circle object to be constructed. If we say

```
   ..=new Circle(5);
```

again, that means another new object is constructed. The situation is like this:

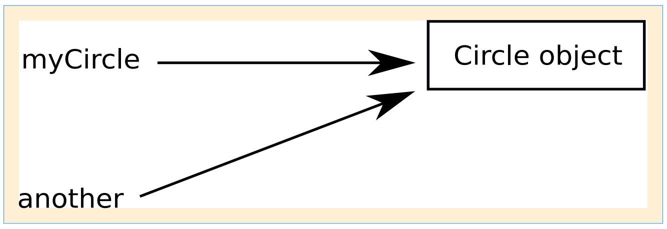


myCircle and another refer to different objects, so they are not equal.

If instead we say

```
    Circle myCircle=new Circle(5);
    Circle another=myCircle;
    System.out.println(myCircle==another); // true
```

We have only said new once, so only one object has been made. The situation is like this:



So myCircle and another point at the same object, so they are equal.

But - maybe we think circles with the same radius should count as being equals. So we add an equals method to Circle:

```
  public boolean equals(Circle other)
  {
    return this.radius==other.radius;
  }
```

Then we can say:

```
    Circle myCircle=new Circle(5);
    Circle another=new Circle(5);
    System.out.println(myCircle.equals(another)); // true
```

Our Circle object might have other attributes, such as colour and position. Should these matter whne deciding if 2 circles are equal? This is part of the design.

Test

1. Try this code out

2. Add an equals method to the Rectangle class.

Overloading

```
class Circle
{
  private double radius;
  Circle(double r)
  {
    this.radius=r;
  }

  Circle()
  {
    this.radius=1.0; // default
  }

..

}
```

Then we can say

```
    Circle myCircle=new Circle(5);
    Circle another=new Circle(); // gets default radius 1
```

We have 2 constructors.

Overloading is when one thing has 2 or more meanings or uses.

Here we have an overloaded constructor. These must have different numbers or types of parameters - or the compiler does not know which to call.

The API shows most classes have oveloaded constructors.

Methods can also be overloaded.

Test

1. Try this out.

2. Add an overloaded constructor to the Rectangle class, giving default values of 1 to the width and height. Test it.

Inheritance

A Square is a type of Rectangle, so a Square class can inherit from a Rectangle base class:

```
class Rectangle
{
  private double width;
  private double height;

  Rectangle(double width, double height)
  {
    this.width = width;
    this.height=height;
  }

  double getArea()
  {
    return width*height;
  }
}
```

and

```
class Square extends Rectangle
{
  Square(double size)
  {
    super(size, size);
  }
}
```

which we use by:

```
    Square mySquare=new Square(2);
    System.out.println(mySquare.getArea()); // 4
```

Because Square extends Rectangle, it inherits the width and height fields. Its constructor says

```
    super(size, size);
```

which means to call the constructor of the base class, with width and height equal. Square inherits getArea() from Rectangle, so we can say

```
mySquare.getArea()
```