# Java Problems and Solutions

This short text looks at some concepts in Java. The idea is that Java is not a set of rules which have to be memorised. Java was designed, initially by James Gosling, Mike Sheridan, and Patrick Naughton. The design was a set of solutions to problems. In other words the features of Java have been chosen to solve a set of problems - some of which are shown here.

In other words, Java makes sense.

The text does not cover all aspects of Java from primitives up. The reader should also be working through a good textbook. This is a supplementary text intended to clear a few doubts.

Comments to w.w.milner@gmail.com

## Table of Contents

# Objects

## Problem

Software consists of data, and code that processes that data. The data can be global, so it can be accessed from anywhere in the application, or local, when it is only accessible inside some code block. ( By 'code block' I mean any unit of code, and in different programming languages these are called methods or functions or procedures or routines)

The problem concerns global data. When we write a new code block, it might contain a bug so that it accidentally changes the global data, leaving it in an invalid state. It also makes team development difficult, since a group working on one block might produce code which changes the global data, making the work of another group incorrect.

## Solution

Design the language so that there is no global data.

Instead we have bundles of code and data, packaged up into 'objects'. Java is an object-oriented language.

Whether the code and data can be accessed from other objects is controlled by 'access modifiers', public private protected and the default (no modifier) which is 'package private'.

Object data members should normally be private - so they cannot be accessed directly from outside the class. We write public getter and setter methods. A public setter lets a private data member be changed, provided the change is valid. A public getter lets the private data to be read.

By declaring a data member to be public we can fake global data. But that is the actual problem we are trying to solve.

Objects are like structs in C or records in Pascal, but they have code blocks (methods) as well as data fields.

## Class

An object has some structure - some data members and some methods, and the methods have coded implementations.

Very often we have many objects with the same structure - the same set of data and methods. This means it is convenient to define a *class* - a structure which several objects will share. For example we might be writing a calendar application, and for this an object representing a date, with day month and year fields, would be useful. But we would need many date objects. Instead of repeatedly setting up day month year structures, we define a Date *class*. Then different date objects are instances of the Date class.

Java source code consists of class definitions. These say the name of the class, and its *members*. There are two kinds of members - data fields (sometimes called data members or attributes) and methods. A class also has *constructors* - code to run when making a new instance of the class.

A class is not a code block. You cannot 'run' a class. It sets out what is in an object with this type. The order these are set out in makes no difference.

A class is also not a collection of data. Objects are not *in* a class. A class is a *type* of object.

## static

static is a keyword meaning per class not per object.

For example suppose we have a class like this:

```
class MyClass
{
int x;
static int y;
}
```

Here x is a normal, non-static field. That means every object of type MyClass has an x field, and they are probably all different. So we can say

```
MyClass object1 = new MyClass();
object1.x=9;
MyClass object2=new MyClass();
object2.x=27;
```

Then the x field of object1 has a value 9, and the x field of object2 has value 27.

But y is a static field, which means it is a property of the class as a whole - not different values for each instance. We can say

```
MyClass.y=43;
```

We use the class name, MyClass, to refer to the static field.

It is possible (but don't do it) to say

```
object1.y=43;
```

This is confusing, because it looks like y is non-static, and that object1 has its own, different, y field - when in fact it belongs to the class, not the object. Confusing code is bad code.

## The Math class

A good example of the use of static is in the Math class.

This has a static data field named PI, type double, value 3.14159.., so we can use this in calculations, such as

```
circumference = 2*Math.PI*radius;
```

The Math class also has a data field E, and static methods like sin cos tan log and so on.

It makes no sense to have more than one Math object - there is only one pi, and only one sine function. So all the members are static. The constructor is private, so that if you say

```
Math myMathObject = new Math();
```

you get a syntax error.

## final

static sounds like it means constant and unchanging. static does not mean constant. It means per class not per object.

The keyword *final* means constant. To be precise, a final data member can only be assigned to once. So you can declare..

```
final int x;
```

then somewhere say

```
x=23;
```

but only *once* - you cannot change x after its first assignment.

Constants are usually named upper case. So Math.PI is declared to be final - you can't change pi.

final applied to a class means it cannot be sub-classed. final applied to a method means it cannot be over-ridden.

# Interfaces

## Problem

A class defines a *type of thing*. What is we want to define a *type of ability*?

For example, we can put numbers into order. We can put strings into alphabetical order (dictionary or lexicographic order). What if we want to define being able to be put into order?

## Solution

An Java interface sets out one or more 'abilities', by listing one or method signatures (method names with parameter lists). There are no method bodies - no definitions of how the methods work.

A class can be declared to 'implement' an interface. Then it must provide an implementation of all methods in the interface.

## Example

This is just to show the syntax:

```
interface MyInterface {

  void someMethod(int x); // no body - no implementation
}

public class Test implements MyInterface {

  public void someMethod(int y) // match signature
  {
    System.out.println(y); // implementation
  }

  public static void main(String[] args) {

  }
}
```

As with classes, we can

1. Write our own interfaces
2. Use standard Oracle interfaces or
3. Use interfaces from third-party libraries

## Comparable

A useful example interface is Comparable.

The Arrays class has a method sort, which can sort an array. But the array elements must implement Comparable, which is a method to put things in order.

Comparable (an standard interface from Oracle) has just one method:

int compareTo(T other)

This method must return a negative integer, 0 or a positive integer, depending on whether the instance is less than, equal to or more than the other

Many classes implement Comparable. For example:

```
    String[] array={"dd", "ab", "aa", "d"};
    Arrays.sort(array);
    for (String str:array)
      System.out.println(str); // aa ab d dd
```

so the compareTo implementation of String puts them into lexicographic order.

Suppose we define a class to represent a graphics point on the screen:

```
class Point
{
  int x,y;
  Point(int x, int y)
  {
```

```
    this.x=x;
    this.y=y;
  }
        public String toString()
        {
          return "("+x+","+y+") ";
        }
}
```

Then we make an array of Points and sort them:

```
    Point[] points={new Point(1,1), new Point(2,3), new Point(3,2), new Point(1,7) };
    for (Point p:points)
            System.out.print(p);
    Arrays.sort(points);
```

This compiles, but when we run it we get an exception - java.lang.ClassCastException: Point cannot be cast to java.lang.Comparable. IOW Point does not implement Comparable - we have not said what it means for one Point to be 'more than' another.

Suppose we order them right to left - IOW on the x co-ordinate. We add a compareTo method to the Point class

```
class Point implements Comparable<Point> {
..
  public int compareTo(Point other)
  {
    if (x==other.x) return 0;
    return x>other.x?+1:-1; // ternary operator

  }
}
```

Then

```
    Point[] points = {new Point(1, 9), new Point(5, 3), new Point(3, 2), new Point(1, 7)};
    for (Point p : points) {
      System.out.print(p); // (1,9) (5,3) (3,2) (1,7)
    }
    System.out.println();
    Arrays.sort(points);
    for (Point p : points) {
      System.out.print(p); // (1,9) (1,7) (3,2) (5,3)
    }
```

## Instantiating interfaces

You can't.

An interface has methods with no definitions, so you can't actually make one.

But you can say

```
SomeInterface link1 = new SomeClass();
```

so long as SomeClass implements SomeInterface.

```
SomeInterface link1..
```

tells the compiler that link1 is a reference to an object which can do the methods declared in SomeInterface.

```
..= new SomeClass();
```

means this will be true, if SomeClass implements SomeInterface. That's what 'implements' means.

## Data in interfaces

We can say for example

```
interface MyInterface
{
  public static final int x=99;
}

public class Test implements MyInterface{

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

We might say x is a 'variable' in an interface. But it must be final, and so is a *constant*, not a variable.

It must also be public and static, so we can miss these out:

```
interface MyInterface
{
  int x=99;
}
```

Why must it be public static final?

An interface data member is an attribute of the *interface*, not the *class* which implements the interface.

So it must be public because otherwise it would be impossible to access.

It must be static because it is not possible to instantiate an interface. So it must be per interface not per object - since there cannot be an object.

It must be final because otherwise any class could alter its value. So one class using it might find another class had changed its value.

This then means interface data members are global constants:

```
interface MyInterface
{
  public static final int x=99;
}

public class Test { // does not implement MyInterface
static int x=100;
  public static void main(String[] args) {
   System.out.println(Test.x); // 100
   System.out.println(MyInterface.x); // 99
  }
}
```

but this is probably an unintended consequence.

## Default implementations

Starting with Java 8, interfaces can have default implementations (the change was made to make lambda expressions work)

```
interface MyInterface
{
  default void method()
  {
```

```
    System.out.println("method");
  }
}

public class Test implements MyInterface{

  public static void main(String[] args) {
    Test t = new Test();
    t.method();

  }
}
```

# Overloading

## Problem

We want to divide double values using /, like

```
double d=7.0/2.0;
```

but we also want to use / to divide ints, like

```
int i=7/2;
```

How?

(d is 3.5, but i is 3)

## Solution

/ is *overloaded*. That means there is *more than one version* of it, depending on the type of values it is given.

When given doubles, / works like division on a calculator, and produces a double result. With ints, / gives the quotient as an int, and ignores any remainder. So 7/2 is 3 remainder 1, so the result is 3.

+ is another overloaded operator. 7+2 is 9, but "Cat"+"Dog" is "CatDog". + *concatenates* Strings.

How? The compiler checks the types of operands, and generates bytecode to do / or + correctly.

In C++ we can overload operators programmatically. For example we can define a Matrix class, then overload + so we can add matrices by actually writing m1+m2. But it makes the compiler complex, so it is not possible in Java ( where you would say instead m1.add(m2) ).

So not much attention is paid to operator overloading in Java.

But we overload constructors and methods all the time.

## Overloaded constructors

For example, in a graphics application we might define a class representing a point on the screen with x and y co-ordinates:

```
class Point
{
  private final int x,y;
  // x and y are final - one assignment only
```

```
  // a Point is fixed - Point is immutable
  public Point() // the no-arg = no arguments constructor
  {
    x=0; y=0; // default values
  }

  public Point(int x, int y) // constructor for non-default values
  {
    this.x=x;
    this.y=y;
  }

}
```

used like

```
    Point p1=new Point(); // (0,0)
    Point p2=new Point(100,200); // (100,200)
```

The compiler can decide which constructor to use by looking at the type and number of arguments.

The String class has 15 overloaded constructors - such as

```
    char[] chars = {'a','b', 'c'};
    byte[] bytes={65,66,67};
    String str1=new String(chars); // from a char array
    String str2=new String(bytes); // from a byte array
    String str3=new String("copy"); // copy another String
    System.out.println(str1+" "+str2+" "+ str3); // abc ABC copy
```

## Overloaded methods

We simply have two or more methods with the same name, but with different types or numbers of arguments.

For example in our Point class, we might have methods to make a new Point shifted to teh right - by a default 1 unit, or a distance supplied as an argument:

```
class Point
{
  private final int x,y;
  ..
  public Point goRight()
  {
    return new Point(x+1,y);
  }

  public Point goRight(int howFar)
  {
    return new Point(x+howFar,y);
  }
  public String toString()
  {
    return "Point x="+x+" y="+y;
  }

}
```

Then..

```
    Point p=new Point(5,5);
    System.out.println(p.goRight()); // Point x=6 y=5
    System.out.println(p.goRight(3)); // Point x=8 y=5
```

The String class has an overloaded indexOf() method, to find characters or Strings inside another String:

```
String str="abcabcg";
System.out.println(str.indexOf('c')); // 2 - first c is at index 2
System.out.println(str.indexOf('c',4)); // 5 first c after index 4 is at  index 5
System.out.println(str.indexOf("bc")); // 1 - start of String cbc
System.out.println(str.indexOf("bc",2)); // 4 - first occurrence after index 2
```

# Packages

## Problem

Suppose you are writing a calendar application, and you define a Date class. But you worry - are theer any other classes named Date? Will there be a name clash if you define a Date class?

## Solution

Classes are grouped into packages. You can have two classes with the same name, so long as they are in *different packages*.

So you might have a package named mycalendar. In that package you can have a class named Date.

The fully-qualified name of a class includes the package. So the fully-qualified name of your class is myCalendar.Date. In fact there are two other Date classes, in packages java.sql and java.util. But there is no clas between java.sql.Date, java.util.Date, and mycalendar.Date.

## Choosing package names

It is important that we do not have two packages with the same name, or the system will fail. How to make sure that does not happen?

Oracle suggests you use your domain name, in reverse. So if your software business has the name GreatCode,  and your website is greatcode.com, then your package name could be com.greatcode.mycalendar. Domain names must be registered and must be unique, so you can be sure there is no other package with this name.

## Packages and folders

Two steps:

1. The source code must start with a package statement. So

```
package com.greatcode.mycalendar;
```

2. The source code file should be a corresponding folder.  So the file would be named Date.java, and be in a folder mycalendar, inside greatcode, inside com.

## package-private access modifier

The assumption is that classes in a package are written together, and 'trust' each other - they have been tested together, and will not corrupt each other. Then if we say

```
class MyClass
{
int x;
}
```

Field x has no access modifier. This means it defaults to 'package-private'. It can be accessed directly (without a getter or setter) from other classes in the same package, but is private from outside the package.

Of course you can alter this using public or private.

## import

Fully-qualified class names are usually a lot of typing. For example

```
com.greatcode.mycalendar.Date someDate = new com.greatcode.mycalendar.Date();
```

is quite annoying.

To fix this, so long as we are only using this Date class, we can use an import statement like

```
import com.greatcode.mycalendar.Date;
```

This tells the compiler that wherever it sees a reference to Date, it means in the com.greatcode.mycalendar package. Then we can just say

```
Date myDate=new Date();
```

Wildcard imports are possibly:

```
import com.greatcode.mycalendar.*;
```

impors all the classes in that package. But this is usually bad practice. Most IDEs have a 'Fix imports' option, and this does it all for you.

Java import is different from C's #include. The import tells the compiler where to look. #include copies and pastes the file into the source code, before it is compiled.

## java.lang

The most key classes, like Object, Math, String and System, are in the package java.lang. There is no need to import it - the compiler will always look in it to find classes.

# Bytecode

## Problem

Computers execute native code. This is 'machine code' instructions which the processor can understand. Computers can *only* execute native code. Not Java or C or PHP - unless they are somehow first changed into native code.

The problem is that each processor has its own set of instructions. Intel processors use one set. Motorola a diffeernt set. ARM a different set. Usually programs also use the operating system (Windows, IOS, Linux). How can we avoid having to write different versions of every program for each different processor and OS?

## Solution

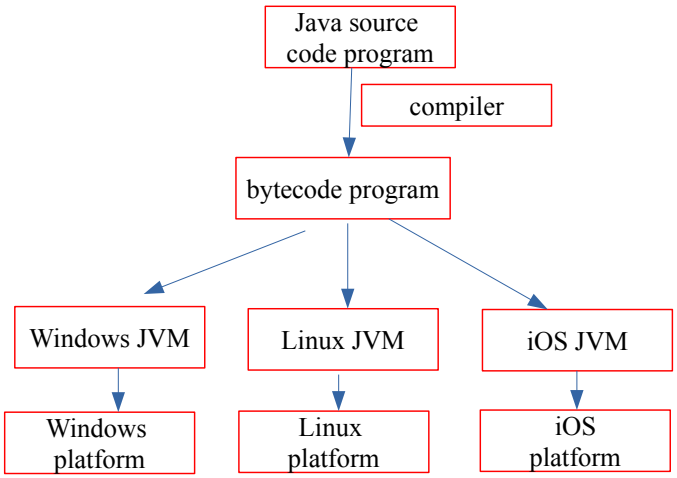
One solution is

1. To write a piece of software for each platform - a virtual machine. The VM makes each platform appear the same. Some platforms might have unusual software and hardware - a finger-print reader for example - and we just ignore those. The VM is written to offer the basic features offered in common by all platforms.

2. Have a language that the VMs can 'understand' and execute.

Then we write programs in that language, and the same code will run the same on any VM.

For Java that common language is called *bytecode* (because each instruction is one byte). But bytecode is pretty low level, so it is easier to write in a high level language - Java as source code - and compile it to bytecode. So the picture is..




The key point is the same bytecode runs on all platforms that have a JVM

## Looking at bytecode

Usually we focus on Java source code, but sometimes it helps to look at the bytecode as well. Souce code of a class named Test is in a file named Test.java, which is compiled into bytecode in a file named Test.class. The utility javap lets us look at the bytecode in a .class file.

We start with a very simple example:

```
public class Test {

  public static void main(String[] args) {
    int x=200;
```

```
    int y=x+5;

  }

}
```

At the command line, in the folder containing Test.class, we can see the bytecode by

javap -c -l Test.class

The -c option shows bytecode, and -l shows line numbers. We get:

```
walter@walter-s5-1030uk:~/Documents/OldStuff-No backup/test3/build/classes$ javap -c -l Test.class
Compiled from "Test.java"
public class Test {
  public Test();
    Code:
       0: aload_0
       1: invokespecial #1    // Method java/lang/Object."<init>":()V
       4: return
    LineNumberTable:
      line 1: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       5     0  this   LTest;

  public static void main(java.lang.String[]);
    Code:
       0: sipush        200
       3: istore_1
       4: iload_1
       5: iconst_5
       6: iadd
       7: istore_2
       8: return
    LineNumberTable:
      line 4: 0
      line 5: 4
      line 7: 8
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       9     0  args   [Ljava/lang/String;
          4       5     1     x   I
          8       1     2     y   I
}
```

How to make sense of this? To start with, what is

```
  public Test();
    Code:
       0: aload_0
       1: invokespecial #1    // Method java/lang/Object."<init>":()V
       4: return
    LineNumberTable:
      line 1: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       5     0  this   LTest;
```

public Test() is a constructor of class Test. But we did not write a constructor. Remember that if we do not write a no-argument constructor, the compiler makes one for us, and this is what this is. It has just one bytecode instruction:

```
       1: invokespecial #1    // Method java/lang/Object."<init>":()V
```

and this just calls the 'method' <init>. We usually say constructors are not methods, because they are not inherited. But internally a constructor is just a code block, like a method, and <init> does what is needed to create a class instance.

The local variable table just has one entry, with name 'this', which is the instance being constructed. The 'signature' is its type - L for a reference type, and the class name is Test.

Then we have the bytecode for main. The local variable table is:

```
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       9     0  args   [Ljava/lang/String;
          4       5     1     x   I
          8       1     2     y   I
```

Slot 0 is the parameter to main, args, of type [Ljava/lang/String; which is an array of Strings. Then we have x, in slot 1, and y in slot 2. Both are type I for int.

The code is:

```
       0: sipush        200
       3: istore_1
       4: iload_1
       5: iconst_5
       6: iadd
       7: istore_2
       8: return
```

Details of all bytecode instructions is in the JVM Specification, at

https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-6.html#jvms-6.5

Bytecode makes extensive use of a stack. This is working as follows:

```
0: sipush   200         ; push 200 onto the stack
3: istore_1             ; pop value off stack and store slot 1 IOW x=200
4: iload_1              ; push slot 1 (=x) onto stack
5: iconst_5             ; push 5 onto stack
6: iadd                 ; pop top two values off stack, add, and push result back
7: istore_2             ; pop value off stack, store in slot 2 IOW y=x+5
8: return               ; finished
```

## Bytecode of an if

For example

```
    int x=6;
    int y=9;
    if (x<y)
      x=0;
```

The bytecode is

```
  public static void main(java.lang.String[]);
    Code:
       0: bipush        6
       2: istore_1
       3: bipush        9
       5: istore_2
       6: iload_1
       7: iload_2
       8: if_icmpge     13
      11: iconst_0
      12: istore_1
      13: return
    LineNumberTable:
```

```
      line 4: 0
      line 5: 3
      line 6: 6
      line 7: 11
      line 9: 13
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      14     0  args   [Ljava/lang/String;
          3      11     1     x   I
          6       8     2     y   I
```

x is slot 1 and y is slot 2. This works as follows:

```
0: bipush  6           ; push 6
2: istore_1            ; pop to slot 1, so x=6
3: bipush  9           ; push 9
5: istore_2            ; pop to slot 2, so y=9
6: iload_1             ; push x
7: iload_2             ; push y
8: if_icmpge 13        ; op two values. goto step 13 if greater or equal
11: iconst_0           ; push 0
12: istore_1           ; x=0
13: return             ; finished
```

Line 8 - if_icmpge means if i integer, cmp compare, ge greater or equal to

## Bytecode of a for loop

For example:

```
    int x;
    for (int c=0; c<100; c++)
      x=c;
```

The bytecode is:

```
 Code:
       0: iconst_0
       1: istore_2
       2: iload_2
       3: bipush        100
       5: if_icmpge     16
       8: iload_2
       9: istore_1
      10: iinc          2, 1
      13: goto          2
      16: return

    LocalVariableTable:
      Start  Length  Slot  Name   Signature
         10       3     1     x   I
          2      14     2     c   I
          0      17     0  args   [Ljava/lang/String;
```

x is slot 1 and c is slot 2:

```
0: iconst_0            ; push 0
1: istore_2            ; pop into 2 so c=0
2: iload_2             ; push c onto stack
3: bipush 100          ; push byte 100 onto stack
5: if_icmpge 16        ; pop two off stack. if greater or equal goto 16
8: iload_2             ; push c onto stack
9: istore_1            ; pop into slot 1 so x=c
10: iinc 2, 1          ; Increment slot 2 by 1, so c=c+1
13: goto 2             ; goto step 2
```

```
16: return
```

Steps 2,3 and 5 check if c<100, and if not, we go to step 16 (and end)

Steps 8 and 9 do x=c

Step 10 does c++

Step 13 is a 'jump' back, for a loop

# Declarations and assignments

## Problem

Suppose we have code (in some language) like

```
x="Black cat";
y=3*x;
```

This is a bug - a problem about type. x has type string of characters, 3*x says to multiply x by 3 - which is OK if x is type number, but not if it is a string.

We could just have this program crash at runtime, and then the programmer has to look through the source code and work out what the problem is - which may be difficult to find.

## Solution

Design the language so that it is strongly typed. That means that the type of any variable must be known at compile-time. So we must say

```
String x="Black cat";
int y=3*x;
```

Then when we compile this, we will get a syntax error. It will tell us we cannot multiply an int by a String.

Then the programmer will look at the line with the error, and quickly fix it. This is much easier than simply having a program which crashes.

## Declarations

But this means we must tell the compiler what type every variable is. So

```
int y;
```

tells the compiler that y is type int.

## In bytecode

What bytecode does a declaration produce? For example..

```
public class Test {
  public static void main(String[] args) {
   int y;
  }
}
```

The bytecode of main is:

```
  public static void main(java.lang.String[]);
    Code:
       0: return
    LineNumberTable:
      line 5: 0
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0       1     0  args   [Ljava/lang/String;
```

So - it produces no bytecode!

A declaration produces no bytecode.

A declaration simply provides the compiler with some information - what type a variable is - so that it can do type checking. It has no effect at runtime, so no bytecode is produced.

An assignment means something happens at runtime, so here bytecode is produced:

```
public class Test {
 public static void main(String[] args) {
  int y;
  y=899;
 }
}
```

becomes

```
public static void main(java.lang.String[]);
   Code:
      0: sipush        899    ; push 899 onto stack
      3: istore_1             ; pop into slot 1 so y=899
      4: return
   LocalVariableTable:
     Start  Length  Slot  Name   Signature
         0       5     0  args   [Ljava/lang/String;
         4       1     1     y   I ; here is y
}
```

We often combine the declaration and assignment:

```
int y = 899;
```

but these are different ideas.

# References

## Problem

Suppose you are writing a graphics program which the user can use to draw circles, squares, lines, triangle and so on.

You are using an OOP language, so the shapes will be objects.

How do we know, when we write the program, how many objects we should declare? How can we know in advance how many shapes the user will want to draw?

In general, if we need to declare all variables in our program, how can we handle more and more data being used at runtime?

## Solution

We need some way to obtain more memory at runtime. We need to be able to say something like

getMemory(1000);

to get memory to store 1000 bytes of data in. In C we do this using malloc or calloc. In Java we use new. So this is *dynamic memory allocation* - instead of declaring variables at compile-time, we ask for more memory as needed at runtime.

But when we get this memory, we somehow need to know where it is - else we cannot use it.

So the call for more memory must somehow return a link to where it is. In Java, this link is a reference.

## References as pointers

We might say:

```
int[] data=new int[1000];
```

to reserve space at compile-time for 1000 ints. But maybe we do not know how much space we need, so we can input that at runtime:

```
Scanner scanner=new Scanner(System.in);
int size=scanner.nextInt();
int[] data=new int[size];
```

Here new int[size] does two things:

1. constructs an array of ints, large enough to store 'size' elements, and

2. returns  a link to that array, which is assigned to 'data' in 'data=new..', so that we can access elements of the array, as data[9] for example.

But what exactly is that 'link'? Let us check the bytecode of:

```
Object obj1 = new Object();
Object obj2 = obj1;
```

We get:

```
      0: new           #2       // class java/lang/Object
      3: dup
      4: invokespecial #1       // Method java/lang/Object."<init>":()V
      7: astore_1
      8: aload_1
      9: astore_2
     10: return
   LocalVariableTable:
     Start  Length  Slot  Name   Signature
         0      11     0  args   [Ljava/lang/String;
         8       3     1  obj1   Ljava/lang/Object;
        10       1     2  obj2   Ljava/lang/Object;
```

new creates a new object. The #2 refers to the class's constant pool, which here is the object class - which is java.lang.Object. new leaves a reference to the object on the stack.

dup duplicates that reference

invokespecial #1 calls the special <init> method of the class - the constructor. It pops a refernce off teh stack to find which object to run teh constructor of. #1 in the constant pool is <init>.

So that making a new object actually happens in two stages - make an object, then call its constructor method.

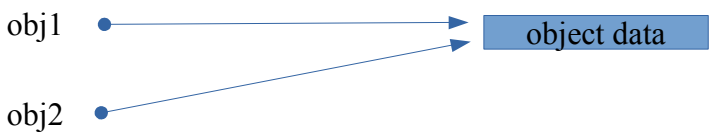
astore_1 pops the object reference off the stack, and  stores it in slot 1 - which is obj1. So this does obj1=new..

aload_1 pushes slot 1 onto the stack, and

astore_2 pops it off into slot2. So this does obj2=obj1.

But this does not tell us what a 'reference' actually is. There is a reason for that.

The idea of a reference is to be a link or a pointer to something. The picture is:




So we have one object, in memory, and two pointers. obj2=obj1means the pointers are equal - in other words, they point to the same place.

Still - what exactly are these references? In C, a pointer is simply a RAM address:

int * ptr = (int *) calloc(1000, sizeof(int));

Here calloc finds space for 1000 ints, and returns the address of the start of the block, which is assigned to ptr.

However this requires the C programmer to manage memory use themselves, and this is a common source of bugs.

The equivalent in Java is a 'reference'. How exactly does that work? Here is one possibility:

```
Object obj1 = new Object();
Object obj2 = new Object();
```

The bytecode has a local symbol table:

```
    LocalVariableTable:
      Start  Length  Slot  Name   Signature
          0      17     0  args   [Ljava/lang/String;
          8       9     1  obj1   Ljava/lang/Object;
         16       1     2  obj2   Ljava/lang/Object;
```

The actual symbol table contents might be

| Slot | Name | Value |
|---|---|---|
| 0 | args | null |
| 1 | obj1 | address in RAM of obj1 |
| 2 | obj2 | address in RAM of obj2 |

The bytecode just refers to slot 1 and slot 2.

The JRE can move objects around in memory during runtime. It might do this for example during a garbage collection, where objects no longer in use are returned to free memory for re-use. So the RAM address of an object can change, and the entry in the symbol table needs to be adjusted.

But then obj1 is still in slot 1, and the bytecode does not need to be changed.

Why is this one possibility? Because the JVM Specification does not require references to be implemented in this way - or in any other way. Different JVMs can do it in different ways - the bytecode will be the same.

For someone programming Java source code, how references are implemented does not matter. The only need is to understand the difference between a reference, and the object in memory it links to.

## References to immutable objects

Some classes are *immutable*, which means their instances cannot be changed after construction. String is an example - a String object cannot be changed.

Immutable is not a keyword. We make a class immutable by having its data members private, and having no setter methods - so its data cannot be changed.

But even if objects cannot be changed, references to them can. For example

```
String ref = "Hello";
```

Now the "Hello" object cannot be changed, since String is immutable.  But we can say

```
ref="Goodbye";
```

Now we have changed ref so it points to a *different object* - the String object "Goodbye".

If we want to make it impossible to change the reference, we can make it final -

```
final String ref="Hello";
```

After that,

```
ref="Goodbye";
```

is a syntax error.

## Parameter passing

Data is passed into methods as parameters. For example:

```
public class Test {
  static int addUp(int[] values) // total an array of ints
  {
    int total=0;
    for (int d:values)
      total+=d;
    return total;
  }

  public static void main(String[] args) {
    int[] numbers={1,2,3,4,5};
    int sum=addUp(numbers);
    System.out.println(sum); // 15
  }
}
```

Here the actual parameter, numbers, is passed into the method addUp as  the formal parameter values. How does this work?

In different languages, parameters are passed by two methods - by reference and by value.

If parameters are passed *by reference*, this means the address of the parameter is passed to the method. That is, a pointer to the parameter is passed into the method. The method can follow the pointer to get the value. It can also change the value there.

If parameters are passed *by value*, a copy is made and passed to the method. The method can change this copy. It cannot change the original.

In Java, parameters are passed by value.

We do an experiment:

```
public class Test {
  static int addUp(int[] values) // total an array of ints
  {
    int total=0;
    for (int d:values)
      total+=d;
    values=new int[3];
    values[0]=99;
    return total;
  }

  public static void main(String[] args) {
    int[] numbers={1,2,3,4,5};
    int sum=addUp(numbers);
    System.out.println(sum); // 15
    System.out.println(numbers[0]); // 1

  }
}
```

Inside the method, we change the values array, and put a new value at index 0. But in the calling code, numbers[0] is unchanged. Why?

In Java, arrays are objects. So 'numbers' is not an array. It is a reference (pointer) to an array object.

Parameters are passed by value. So in

```
    int sum=addUp(numbers);
```

a *copy* of the pointer is made, and is referred to as 'values' in the method. In other words, 'value' points to the same array as 'numbers'. The method adds them up. Then it says:

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

This *changes the copy* - to point to a new array. But that will not alter the original array.

Another experiment:

```
public class Test {
  static int addUp(int[] values) // total an array of ints
  {
    int total=0;
    for (int d:values)
      total+=d;

    values[0]=99;
    return total;
  }

  public static void main(String[] args) {
    int[] numbers={1,2,3,4,5};
    int sum=addUp(numbers);
    System.out.println(sum); // 15
    System.out.println(numbers[0]); // 99
```

```
  }
}
```

Here in the method, we have followed the copied pointer, and *altered the object pointed to*. So number[0] changes.

In this example there is only one array:

```
    int[] numbers={1,2,3,4,5};
```

and we copied the pointer to it. We did not copy the array.

People sometimes get confused. In Java parameters are passed by value. But in Java values (except for primitives) *are* references. *Copies* of those references are passed.

Why?

Objects can be very large. An array with a million elements is an object. Copying it would be slow, and use a lot of memory. But the reference to it is just a few bytes, and copying that is fast.

# Inheritance

## Problem

A toString() method could produce a string which describes the object, perhaps by saying what class it is, and showing the values of some or all fields. This would be useful because we could then print out the toString method of an object, so we could 'see' it, and use it in logs.

Another useful method would be equals(). This could be used to return true if two objects had the same value of key fields.

How can we make it so that all classes have these methods, without having to write them again and again?

## Solution

1. Have a hierarchy of classes, like a family tree. A parent class is defined to have fields and methods. A child class inherits them. It can also provide new versions of the methods ('over-riding' the parent methods), and add additional members.

2. Have a class at the top of the hierarchy, called Object. All other classes are direct or indirect children of Object.

3. Define equals and toString methods for Object. Then all other classes inherit those methods.

## Inheritance

Putting equals and toString in every class is not the only purpose of inheritance. It is often useful to put code in a base class (parent) and then to *re-use* the code by inheriting it in sub-classes (child).

As an example, look at the class Number. This has Object as parent, and ten children:

AtomicInteger, AtomicLong, BigDecimal, BigInteger, Byte, Double, Float, Integer, Long, Short

Byte, Double, Float, Integer Long and Short are the primitive wrapper classes. Each has a field of primitive type (like Double has a double field) plus methods connected with it.

BigDecimal and BigInteger have almost unlimited size and precision.

AtomicInteger and AtomicLong are like Integer and Long, but will be arithmetic in one 'step', to avoid problems with threads.

Number has a method byteValue(), which returns the value as a byte. The API says "This may involve rounding or truncation." In other words, if the actual number is more than 8 bits (such as a 16 bit short) you only get the lower 8 bits - which is truncation. Or if it has a fractional part (say 2.3) you get the whole number part (2), and this is rounding.

Then all the 10 sub-classes inherit the byteValue method:

```
Number[] numbers = new Number[3];
numbers[0]=new Integer(1025);
numbers[1]=new Double(2.8);
numbers[2]=new Byte((byte)15);
for (Number n:numbers)
  System.out.println(n.byteValue()); // 1 2 15
```

We have declared numbers to be an array of 3 Number instances. But in fact the elements have type Integer, Double and Byte. Why is this allowed?

Because Integer Double and Byte extend Number. This means an Integer can do everything a Number can (because it inherits all its members). So the compiler has no problem that we might tell an Integer to do something a Number can do. If a Number can do it, so can an Integer.

We say:

An Integer is-a Number

This is an example of the *Liskov substitution principle*. Wherever we have a base class, we can replace that with a sub-class, because a sub-class can always do what a base class can do. (The method might be over-ridden, so it might do it in a different way. But it still has a version of the method.

Why do we get 1 2 and 15?

The Integer is decimal 1025. In hex that is 401, or in binary, 0010 0000 0001 (with leading zeros). If we just have the lowest 8 bits of that (which is what byteValue() does, truncating the value ) we get binary 0000 0001, or decimal 1

The Double is 2.8, and when we convert that to a whole number we get 2 (so in fact it does not round)

The byte is 15, which already fits into an 8bit byte, so we get 15.

## abstract classes

Any Number is one of those 10 sub-classes. We cannot have a Number instance, unless it is one of those - it makes no sense. There is a fix for that:

```
Number n = new Number();
```

is a syntax error:

error: Number is abstract; cannot be instantiated

Number is declared to be an abstract class. That means it cannot be instantiated. The idea is to stop us writing code that does not make sense. A number must be a Double or an Integer or whatever - it cannot be nothing.

## Over-riding methods

In a sub class, we can provide a new implementation of a base class method. This is called over-riding. For example byteValue() is over-ridden in the sub-classes of Number. A simple example:

```
public class Test {
  public static void main(String[] args) {
    Sub sub = new Sub();
    sub.method(); // sub
  }
}

class Base
{
  void method()
  { System.out.println("Base");}
}

class Sub extends Base
{
  void method()
  { System.out.println("Sub");}
}
```

Here method is over-ridden in Sub, and sub.method calls this new version.

## Declared type and actual type

Suppose we have:

```
public class Test {
  public static void main(String[] args) {
    Base ref = new Sub();
    ref.method(); // sub
  }
}

class Base
{
  void method()
  { System.out.println("Base");}
}

class Sub extends Base
{
  void method()
  { System.out.println("Sub");}
}
```

Here ref is declared to be type Base. But it actually points to a Sub instance.

Is this allowed? Yes. By the Liskov substitution principle. A Sub can do anything a Base can do, so the compiler allows it.

What happens at runtime? ref points to a Sub object, so it executes the Sub version.

What about this:

```
    Base ref = new Sub();
    if (Math.random()>0.5)
      ref=new Base();
    ref.method(); // ?
```

This uses a random number. If we run this several times, we sometimes get the Sub version, sometimes the Base version.

At compile time, we cannot tell what the type of ref will be at runtime.

Only at runtime do we know the type of ref. This is called runtime polymorphism. The word polymorphism means different shapes. The idea is that ref might be a Base or a Sub, and we only know which at runtime.

## Fragile base class problem

Some people think inheritance should be avoided, because of the fragile base class problem.

Suppose we have a base class Base, and a sub class Sub - and maybe lots of other sub classes, which themselves have sub classes. They all inherit Base class members.

No suppose we want to change how Base is written, for example by removing a data field and replacing it with another.

As a result all the sub classes which use that data member now fail.

Beware.

## Inheritance by composition

Composition is an alternative to inheritance. For example

```
class MyClass
{
private String value;
..
}
```

Then MyClass can do what Strings can do, by using the methods of 'value'.  For example, String has a 'toUpperCase' method, so..

```
class MyClass
{
  private String value;
  public MyClass(String value)
  {
    this.value=value;
  }
  public void toUpperCase()
  {
    value=value.toUpperCase();
  }
  public String toString()
  {
    return value;
  }
}
```

Then..

```
    MyClass ref = new MyClass("aBcDeF");
    ref.toUpperCase();
```

```
    System.out.println(ref); // ABCDEF
```

String does ot have a reverse method. But our class can have one:

```
  public void reverse()
  {
    StringBuffer str=new StringBuffer(value);
    str.reverse();
    value=new String(str);
  }
```

Then

```
    MyClass ref = new MyClass("aBcDeF");
    ref.reverse();
    System.out.println(ref); // FeDcBa
```

The String class is final, so we could not do this by extending String. If the implementation of String were to change, this would not stop MyClass working, provided its methods still had teh same effect.

## Multiple Inheritance

This is when one class inherits from two or more base classes - so you could say like

```
class Sub extends Base1, Base1
..
```

But there is no multiple inheritance of classes in Java.

You can do this in C++. But analysis showed this was not actually used often in real C++ code, and it made the compiler more complex, so a design decision was taken to not allow multiple inheritance of classes in Java.

A class can implement two or more interfaces:

```
class MyClass implements Interface1, Interface2
..
```

but this is not inheritance. It gets nothing from Interface1 and Interface2. It has itself to provide implementations of both of those methods (apart from those with default implementations).

If you need a class to have the abilities of  two other classes, use composition:

```
class MyClass
{
A a=new A()
B b = new B();
..
}
```

Then you can say

```
MyClass mc = new MyClass()
mc.a.method1(); // call method from class A
mc.b.method2(); // method from class B
```

# Generics

## Problem

Programs use algorithms. As a simple example, we can do a linear search of an array for a value by iterating through the array from the start until we find a matching element.

This algorithm works for any data type. However, Java is a strictly-typed language. We have to declare the type of any variable in code. But we want to write code to express the algorithm, and not have to re-write the code for every possible type.

## Solution

We write the code with a *type parameter*.

Usually a parameter to a method provides some input *data*. A type parameter does not - instead it provides *type* information.

## A linear search

For example, to find a data value in an array:

```
public static <T> int find(T value, T[] array)
{
  for (int i=0; i<array.length; i++)
    if (value.equals(array[i]))
            return i;
  return -1; // not found
}
```

Here the T is the type parameter.

We can use this as:

```
  String[] sa={"a","b", "c", "d"};
  System.out.println(find("c", sa)); // 2

  Integer[] ia={6,3,2,7,3};
  System.out.println(find(3, ia)); // 1
```

We can use the same generic code to search arrays of Strings or Integers or any reference type.

This works for reference types, not primitives, which is why we have Integer[] not int[]. The code uses autoboxing - in ia={6,3,2,7,3}; the 6, for example, is being converted to the corresponding wrapper type - an Integer.

The Arrays class has a set of overloaded binarySearch methods. There is a version for each primitive type, and versions using a type parameter. A binary search is faster than a linear search, but requires the array to be sorted.

## Iterable

A class which implements the Iterable interface can be used in a 'foreach' loop

```
for (SomeClass object: set)...
```

where set is a set of SomeClass objects.

We will write a linked list class which implements this. The Collections framework offers a LinkedList class, so you would not normally do this - this is an example of how this kind of thing works.

The linked list (containing ints) will be a series of linked nodes:

```
class Node {
  int data;
  Node next;
  Node(int data) {
    this.data = data;
    this.next = null; // pointer to next Node in the list
  }
}
```

The list class is:

```
class MyList implements Iterable<Node> {
  Node start = null; // first node in list - null for an empty list

  public void add(int value) { // add an extra value
    Node newNode = new Node(value); // make a node
    if (start == null) { // if empty list
      start = newNode; // start points to this new node
      return; // that's it
    }
    Node nodePtr = start; // look for current last node, starting at start
    while (nodePtr.next != null) { // until find last node..
      nodePtr = nodePtr.next; // move to the next one
    }
    nodePtr.next = newNode; // link last node to new last node

  }
..

}
```

At this stage we can say for example

```
    MyList list = new MyList();
    list.add(3);
    list.add(4);
    list.add(27);
```

The Iterable interface has just one method, which MyList must implement

```
  public Iterator<Integer> iterator() {
    return new ListIterator(this);
  }
```

Iterator is a parametrized interface. Here iterator() must return an instance of a class which implements Iterator<Integer>, an iterator which returns Integer type objects. So we need a class that does this:

```
class ListIterator implements Iterator<Integer>
{
  MyList theList;
  Node pointer; // pointer will move through the list

  ListIterator(MyList list)
  {
    theList=list;
    pointer=list.start;
```

```
  }
..
}
```

But ListIterator must implement the two Iterator methods, hasNext and next..

```
  public boolean hasNext() {
    return pointer!=null; // if next is null, we've passed the end
  }

  public Integer next() {
    Node val=pointer; // current node
    pointer=pointer.next; // move to next node
    return val.data; // return data at what was the current node
  }
```

Then we can say:

```
    MyList list = new MyList();
    list.add(3);
    list.add(4);
    list.add(27);
    for (Integer i: list)
      System.out.println(i); // 3 4 27
```

# The Collections framework

## Problem

Classical computer science is about data structures and the algorithms to use them. Examples of data structures are lists, stacks, queues, trees and hash tables.

But these are not part of the Java language, and coding them from scratch every time they are needed would be very inefficient.

## Solution

Oracle provides the Collections framework as standard library interfaces and classes which model most of the standard data structures.

You should study data structures and algorithms in a language-agnostic way - that is, in a way that does not depend on a programming language. This is because the data structures and algorithms are the same in any language. For example a stack is a stack, in Java or C or C++ or Pascal or assembler. The task is to understand what a stack is, not the syntax of how it is used in any one language.

You should know about data structures and algorithms and big-O analysis before using the Collections framework.

## Generics

The algorithms to use data structures are the same, no matter what data type is held in the structure. As a result the collections use type parameters to control that type.

As an example, suppose we want an array of Strings - but we want to be able to add new Strings at runtime, so we cannot use a normal array (which is fixed in size).

```
import java.util.ArrayList;

public class Test {

  public static void main(String[] args) {
    ArrayList<String> strings = new ArrayList<>();
    strings.add("One"); // put strings into list
    strings.add("Two");
    strings.add("Three");
    strings.add("Four");
    strings.remove("Three"); // remove one
    for (String str : strings) // iterate through list
    {
      System.out.println(str); // One Two Four
    }
    System.out.println(strings.contains("Two")); // true
  }
}
```

The declaration uses the *diamond operator*

```
    ArrayList<String> strings = new ArrayList<>();
```

We said strings contains String, so the compiler can work out what <> must mean - there is no need to say

```
    ArrayList<String> strings = new ArrayList<String>();
```

We can miss out the type parameter

```
    ArrayList strings = new ArrayList();
```

but this is a *bad thing*. If we declare the list this way, it uses raw type. We can then put any object into it, of any type. So that if we take an object out of it, we cannot know what type it might be. For example:

```
    ArrayList strings = new ArrayList();
    strings.add("One");
    strings.add(new Integer(4));
    for (Object obj : strings)
    {
      System.out.println(obj.getClass()); // class java.lang.String class java.lang.Integer
    }
```

Normally we want a *typesafe* list, so we can only add one type to it, and when you take something out, we know what type it will be.

## Maps

A map is a data structure made of key-value pairs. Each pair has a unique key and a value linked to it. No two pairs can have the same key. We put key-value pairs in the map. Then we can use a key to search the map, and get back its linked value.

For example

```
    HashMap<Integer, String> map = new HashMap<>();
    // put some key value pairs into the map
    map.put(6,"one");
    map.put(34, "two");
    map.put(18, "three");
    // get values out, with key
    System.out.print(map.get(34)); // two
```

```
    System.out.println(map.get(5)); // null
```

Suppose we need to take a string, and count how many times each character is found in the string. A map would be useful for this. The map would be Character-Integer. The key is a Character (in the string), and the value, an Integer, is the count of how many times it is found. The algorithm is
for each character in string..
..if not in map, put in map with a count of 1

..else increment count of that key-value pair:

```
    HashMap<Character, Integer> map = new HashMap<>();
    String test="Test aaa Π 111223";
    // go through string..
    for (int index=0; index<test.length(); index++)
    {
      Character c = test.charAt(index);// get each character
      Integer count=map.get(c);        // fetch from map
      if (count==null) map.put(c, 1);  // not in it yet
      else {                           // or
        map.put(c,count+1);            // increase count
      }
    }

    for (Character c : map.keySet()) // go through all keys
      System.out.println(c+" "+map.get(c)); // and fetch count
```

Output is:

```
  3
Π 1
a 3
1 3
2 2
s 1
3 1
T 1
t 1
e 1
```

There is a lot of autoboxing here, between Character and char, and Integer and int. For example

```
Character c = test.charAt(index);
```

charAt returns a char, but this is autoboxed to a Character.

## Sets

Sets do not contain duplicates.

Suppose we have a problem like the last, but just want a list of all characters occurring one or more times, in a given String:

```
    HashSet<Character> set = new HashSet<>();
    String test="Test aaa Π 111223";
    for (int index=0; index<test.length(); index++)
    {
      Character c = test.charAt(index); // get each character
      set.add(c); // if already in, does nothing
    }
    // convert to array
    Character[] array = set.toArray(new Character[0]);

    for (Character c : array)
      System.out.println(c);
```

Output:

```
П
a
1
2
s
3
T
t
e
```

## Interfaces

The Collections framework really starts with a set of interfaces, then has a set of classes which implement those interfaces. The Interfaces are:

Collection - basic add and remove methods. Four sub-interfaces:

- Set - no duplicates
- List - lists
- Queues - first in first out queues, mostly
- Deques - double-ended queues

Map - key value pairs. One sub-interface

- SortedMap

The List interface is implemented by classes like ArrayList, LinkedList and Stack.

See the API for full details

## Abstraction

Abstraction is an idea useful throughout Computer Science.

It means a focus on what things do, and not how they do it.

In terms of Collections, it means looking at what the classes do, not how they are implemented.

An example is a TreeMap. This is a map of key-value pairs, structured as a tree. It is implemented as a red-black tree (which is a way of being sure the tree is balanced - an unbalanced tree becomes slow). It could have been implemented as a AVL tree, which is also balanced. How it is implemented makes no difference. Oracle could change it to an AVL tree, and the API would not alter, and no code which used a TreeMap would need to be altered.

A TreeMap is guaranteed to do containsKey, get put and remove in O(log n) time. This is all that matters. How it does it does not matter.

## Interfaces and abstraction

We often write code like

```
    List<String> list = new ArrayList<>();
```

and not

```
    ArrayList<String> list = new ArrayList<>();
```

List is an interface, while ArrayList is a class.

Firstly - are we allowed to do this?

We cannot instantiate an interface. We cannot say

```
    List<String> list = new List<>();
```

but

```
    .. = new ArrayList<>();
```

instantiates a class, and we can assign this to something declared to be an interface, so long as the class implements the interface. ArrayList implements List, so its OK.

But why is it better? It is more flexible. We might decide that a linked list is better than an ArrayList. So we change it to

```
    List<String> list = new LinkedList<>();
```

but list is still declared as a List<String>. That means no code which uses list needs to be changed.

In general we are saying

```
<interface> ... = new <class that implements interface>
```

This is another example of abstraction. We need list to do List methods. It makes no difference if it works using an ArrayList or LinkedList.