An Introduction to Java

Notes by Scott Hazelhurst
Lectured by Pravesh Ranchod

2013

1 Background

Java developed in the 1990s by Sun Microsystems:

- Basic philosophy: model world as objects
  - Organise the world into classes of objects;
  - Objects can communicate using *methods*
- Underlying model of computation *imperative*.
- Some of the goals are:
  - use of libraries, promote reuse of code, portable
- Provides facilities for: web programming, network programming, multithreading, GUI.
- Progression from existing languages: C-like in syntax.

Note that we are going to use Java 5 or later. Although most of what we shall study here applies to earlier versions of Java too, the following features definitely will not work in earlier versions of Java:

- Enumerated classes
- Extended for statments

- Generic classes

Some Java-Python differences !

- Java has distinct compile and run-time phases
  Python only does very simple syntactic checking.
- Java is explicitly typed.
  - Types checked at compile and run time
- Rules of scope are different (i.e. what's local, what's global)
- Java has much more syntactic overhead.
- Lots of minor differences.

2 Object-orientation and programming-in-the-large

Software crisis Cost of software typically by far most expensive part of any system

Complexity grows Complexity of program *superlinear* in size of program:

- *Mythical Man-Month*

Modularity and **Code re-use** seen as possible solutions
Program is made up of a number of units called *classes*.
Good programmers need to:

- Be able to use classes already defined
- Extend existing classes
- Build new classes
  - Clean interface to outside world
  - Correct, efficient functionality

Classes describes data *and* what can be done to the data.
For each class we say what objects of that class look like:

- *instance* variables for storing state or data;
- *methods*: functions/procedures which modify, operate upon, or access the data in the instance variables.

Idea: try to encapsulate concept being modelled

- The set of public methods forms the *interface* to the class

Classes are the *type* of an object.

- When an object is created, we say what class it is. The object has the instance variables and methods defined by the class.
- To use or modify an object, invoke one of the methods.

Less common

- Less common, but very important exceptions: the class definition provides the functionality etc. directly.
- Examples: *main* program, certain libraries like *Math*.
- We'll see some examples soon.

Programming-in-the-large

- How to build large systems from existing classes and new ones.
- Focus: defining *abstractions* or *interfaces* and re-using others.
- **The real challenge in programming**

Programming-in-the-small

- How to code up individual methods (functions/procedures). Basic syntax, semantics of language. Primitive data types and operations. Loops. Selection ...
- Basis on which it all rests.

We start with programming in the small.

3 Basic Components of a Java Program

A Java program must contain one public class.

- instance variables (either Java primitive types or using other class definitions).
- methods (procedures/functions)
- A method is a *named*, separately callable sequence of code.
- One method called *main* **must** be defined.

When the program starts to run, the *main* methods starts executing. It can call other methods.

Slightly different rules apply to applets.

Our first example

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

- The keyword *public* indicates that the class and method can be accessed outside of the current environment — more detail later.
- Note that we use a method in the *System* class for outputting.
- If we compile and run the program, then the *main* method executes.

4 The imperative style

The underlying model of computation of Java is the imperative programming style:

- We have a set of variables or objects that store data or state.

- Statements are executed in order, changing the values of variables or objects.

```
int i, j;
float f;
.. ..
i = i+10;
j = Math.abs(j-2*i);
i = Math.ceil(f)+i;
.. ..
```

5 Variables

- In Java, all variables must be declared before being used.
- All variables have a type which describes which values they can take on.
- Variables can either be:
  - primitive Java types
  - complex or reference types, objects of any class (actually references to these)

The type of the variable not only describes the values the variable can take, but operations that can be performed.

Need to be type-consistent.

```
int a, apple;
int j33, banana;
int cherry = 3, data = 0, k;

a = 3;
apple = 2;
banana = 1;
a = a + 1;
apple = apple + a;
```

```
data = a;
System.out.print(a);
System.out.println(data);
System.out.println(cherry);
```

The primitive Java data types

Type	Size in bits	Range
bool	1	**true , false**
char	16	ISO Unicode set
byte	8	-128 to 128
short	16	app. $-32000 \ldots 32000$
int	32	app. $-2 \times 10^9 \ldots 2 \times 10^9$
long	64	app. $-10^{19} \ldots 10^{19}$
float	32	app. $-3 \times 10^{38} \ldots 3 \times 10^{38}$
double	64	app. $-1.7 \times 10^{308} \ldots 1.7 \times 10^{308}$

Note: Java 1.5 also has enumerated types *Enum* – these are very useful, and you should read up about them.

```
public class simptypes {

    public static void main(String [] args) {
        int ans, i;
        char reply;
        byte a;

        ans = 0;
        i   = 10;
        a   = 65;
        reply = 'y';
        ans = ans+1;
        i   = i+10;
        reply = (char) a;
        System.out.println("Ans is " + i + " and " + reply);
    }
}
```

6 Declarations

Before a variable can be used, it must be declared.

General form: `type-name  list-of-variables;`

Examples:

```
int i, j;
char j,answer;
long rainfall;
float radius,r3;
```

By convention, variable declarations given first:

- Makes the variables in a function clear;
- But, can give variable declarations anywhere, and sometimes there are good reasons for doing so;
- Variable can only be declared once in a method;
- But variables with same name can exist in many methods, classes.
- **NB:** just because two variables have the same name doesn't mean they are the same.

  Scope rules say which is which.
- Can initialise in a declaration.

Other modifiers possible: e.g. static, final etc.
We'll look at modifiers later
Variables have four important features:

- *Name* by which they are referred in program.

  This is context dependent – scope rules.
- *What it represents conceptually*
- *Reference* or *Address*

  Where in memory it is physically stored
- *Value:* the value that is stored there

7 Flow of control

When a program starts to run, the *main* method starts executing.

Statements are executed sequentially in order except when altered by:

- call to another method

  and

  when a method finishes executing, control passes back to where it was called

- selection

  if, switch statements allow us to make choices

- iteration

  for, while and do loops allow us to repeat sections of code.

A statements in a Java program can be a:

- declaration
- assignment statement
- selection (e.g. if, switch)
- loop (e.g. for, while)
- method invocation (function call)

  method can be one you have written, or from a library, or from some other class.

- compound statement: `{sequence-of-statements}`

7.1 Assignment

The most common form of the assignment is

varname = value;

On the left is a variable, on the right a value that is type compatible:

- The assignment rule is: evaluate the expression on the righthand side fully.
- Change the value of the variable to that value.

Examples:

```
n = 3*u+2 % 5;
m = 'a';
n = (int) m;
```

Java has a number of other operators borrowed from C:

- Increment and decrement: ++ and --

  `x++;` or `++x;` same as $x = x + 1$.

  `y=g*(++x)` same as `x=x+1; y=g*x;`

- Arithmetic assignment operators: `+=` `-=` `*=` `/=` `%=`

  Example: `x += y` is the same as `x = x+y;`

Overuse of these can make code difficult to read.

- My suggestion: don't use $--$ and $++$ on the RHS of an assignment.

7.1.1 Operators on numbers and boolean

Operator	Assoc	Explanation
()	L-R	parentheses
++; --, +; −	R-L	unary operators
∗; /; %	L-R	mult, div, modulo
+; −	L-R	addition, subtraction
<; <=; >; >=	L-R	relational
==; ! =	L-R	equality
&	L-R	boolean and
^	L-R	boolean XOR
|	L-R	boolean OR
&&	L-R	conditional AND
||	L-R	conditional OR
? :	R-L	conditional
=; + =, − =,...	R-L	assignment

- `&&` does short-circuit evaluation; $\&$ does not.

  Suggestion: use one or the other, not both.

- Conditional operator is ternary

```
boolean b;
...
x = b ? y : z;      <---   if (b) x=y; else x=z;
...
```

This is not a complete list of all the operators. For example, there are integer bit-wise operators for doing bit-wise and, or and complementation. For example $325\&7$ represented in binary is $0101000101\&0000000111$ which is 101. We have done a bit-wise and of the bits in the number.

8 Input

Scanner

To read an integer from the keyboard into a variable x:

```
import java.util.Scanner;
public class Test{
   public static void main(String args[]){
      Scanner in = new Scanner(System.in);
      int x = in.nextInt();
   }
}
```

- Creating a `Scanner` object allows us to get that object to do the work for us
- When the program runs, it suspends at `in.nextInt()`, waiting for keyboard input.
- The user must then type in an integer using the keyboard and press the RETURN key.
- The program then continues executing – the value the user typed in is stored in the variable x (it's an assignment statement, remember).

The Scanner class provides a number of useful functions to read input:

- `nextInt()` reads and returns an int
- `nextLine()` reads and returns a String, by reading an entire line of text
- `nextDouble()` reads and returns a double

Note that there are some complications involved, so in a lot of cases it makes sense to use `nextLine()` wherever possible. This can be accomplished by converting the returned `String` into the data type required. For instance, if you needed to read in an `int` value, it could be done in the following way:

```
Scanner in = new Scanner(System.in);
String r = in.nextLine();
int val = Integer.parseInt(r);
```

9 Selection

selection if

The **if** statement is the most important method of selection:

General form 1

```
if (condition) statement;
```

General form 2

```
if (condition)
  statement;
else
  statement;
```

- Parentheses around the condition is obligatory
- If you need multiple statements, then need {sequence-of-statements}

```
num =
a =
if (num < 0)  num = -num;
b = 15;
if (a > b  &  num > 0)   {
   x = 5;
   y = 10;
};
```

```
if (num < 0)  num = -num;
if (code  &&  x > y)
     ans = a;
else
  rep = a;
```

```
if (res)   {
     x++;
     res = res  &&  x < 10;
} else {
  x--;
  res = res  &&  x > 20;
};
```

If statements can be nested:

- Elses match nearest open If
- Java suffers from the dangling else problem.

```
if (cond1)
    if (cond2)
       x = 1;
    else
      x = 2;
```

- Take care that indentation doesn't mislead.
- Can use braces to change if necessary.
- Always a good idea to use braces for the *then* and *else* parts.

The **switch** statement is the other Java selection method:

```
switch (value) {
  case v1 : sequence-of-statements;
  case v2 : sequence-of-statements;
  ...
  case vn : sequence-of-statements;
  default  : sequence-of-statements;
};
```

Variable `value` is evaluated.

- Each of the cases is examined in turn.
- If a case matches `value` exactly, **all** the statements in the switch statement from that point on are executed.
- If no case matches, optional default section executed.

Typically use the `break` keyword to only have some of the statements executed.

```
switch (day) {
            case 0 : rate = 2.0;
                 break;
            case 1 : rate = 0.8;
                 break;
            case 2 : rate = 1.0;
                 break;
            case 3 : case 4 :
                rate = 1.2;
                 break;
            case 5 : case 6 :
                rate = 1.8;
                 break;
}
```

Using the switch statement:

- Take care – usually want to have a **`break`** at the end of each case, but easy to forget and compiler doesn't catch.

  `switch` is insecure in this way.

- Usually good practice to put in a default section: it might just be error handling code to deal with unexpected values.

- Always put a **`break`** at the end of the final non-default case section.

The **`switch`** statement is very useful, but used too often can lead to complex code:

- use of polymorphism in object-oriented programs reduces need to use it.

- use table-based methods.

These latter approaches are more compact, often more efficient, easier to read and to maintain.

10 Loops

Three loops in Java, **`while`** , **`for`** , and **`do`** .

10.1 While loops

General form of the **`while`** loop is:

```
while (cond)
        statement
```

Evaluate the condition; if it's true the statement is executed and then we repeat . . .

- Good idea to use { `statement-sequence` } for body of loop;

- Note the contents of the while loop need never be executed if the condition is initially false.

Task: read in sequence of integers terminated by -1 (sentinel) and find the average.

```
int num, sum = 0, n = 0;
double ave;
Scanner in = new Scanner(System.in);
num = in.nextInt();
while (num! = 1) {
    sum+ = num;
    n++;
    num = Scanner.nextInt();
}
ave = (double)sum/n;
```

Normal execution:

- Check condition, execute *entire body*, repeat

Two ways of extra-ordinary change of control:

- **`continue`**: rest of body of loop missed, go to beginning of loop again.
- **`break`**: leave loop immediately.

Useful sometimes, *but* breaks structured programming.
Here is a good example where you might use this feature

```
int num, sum = 0, n = 0;
double ave;
while ( true ) {
        num = in.nextLine();
        if (num == −1) { break; }//Exit here
        sum+ = num;
        n + +;
}
ave = (double )sum/n;
```

Be very careful if you find yourself doing this more often. Rule of thum — only use once per loop

10.2 For loops

General format is

```
for (init; cond; change)
    statement;
```

Semantics are:

1. The *init* statement is first executed;
2. If *cond* is true the body is executed;
3. After body is executed, `change` is executed;
4. Goto step 2

In 1.5 there is a construct analogous to Python's *for* statement. We'll look at it after doing arrays.

```
for (i = 1; i <= 10; i++) {
    n = n + i - 3 * n;
    System.out.println
      (i + "computer and its mouse went to mow a meadow");
};
```

```
for (i = 10; i >= 1; i--)...
```

```
for (i = 0; i < 10; i++)...
```

Real computer scientists count from zero!!!!
Example

```
static void compfn () {
   int x;
   double y1, y2;
   for (x=0; x<100; x=x+10) {
      y1 = 3.1*x*x - 38*x + 15;
      y2 = 2*x*x   +  2*x + 1;
      System.out.print("x = "+x+"; y1= "+y1+"; ``+
                       "y2="+y2+"; ");
      System.out.println(y1>y2?"+":"-");
   }
}
```

10.3 Do loops

Similar to the **while** loop except the condition testing is done at the end.

```
 do {
...
} while (cond);
```

- Can use break and cont;
- Take care in indentation.

10.4 Summary

Any loop can be coded as any of the three!

- The **for** loop is the most powerful – don't try to do too much, strive for simplicity.

11 Reference/Complex data types and references

The use of references is one of the basic features of the semantics of Java.
For primitive types, when a variable is declared,

- The compiler allocates memory big enough to store a value of that type;

- The data for the variable is actually stored at the variable's address in memory.

For all other types: objects, arrays, etc,

- The compiler allocates memory only for a *reference* to the object;
- The data for the object is not stored at the variable's address in memory – only the *reference* to such an object can be stored here.
- Memory must subsequently be allocated for the object.

Key points:

- What we see as a declaration of an object, is a declaration of a reference capable of referring to the underlying object;

Memory for underlying object must be allocated and values set up appropriately.

- Explicit allocation using the **new** keyword;
- Assignment to an existing object;
- Class constructor – a method that automatically is called when an object is declared.

11.1 String class

Capable of storing strings — Not primitive in Java, in automatically imported `java.lang` class.

```
String myname,x,y,z; // Just the ref!

z = new String ("The Zoo");
myname = "Fred the Penguin";
x = myname;
y = myname + " is here";
```

or
`String car = "Red car"`

- **`String`** has many string manipulation routines – read up about them in a reference.
- Examples:`comparesTo` and `equals`.

Why do a and b end up with different values?

```
String  s1, s2;
boolean a, b;

s1 = new String ("Wits");
s2 = new String ("Wits");

a = s1.equals(s2);
b = s1 == s2;
```

11.2 Simple Java arrays

An array is an *homogeneous* collection of objects, the individual elements of which can be accessed by index.

- The first item is indexed by zero;
- The last item by $n - 1$ if the length of the array is n

```
Type array_name[];
array_name = new Type[n];
```

- May need to worry about the memory allocation of individual items if the type is not primitive.

```
int rainfall[], sum;
rainfall = new int [12];
sum = 0;
for (int m = 0; m < 12; m++) {
```

```
    rainfall[m] = in.nextInt();
    sum+ = rainfall[m];
}

int rainfall[][];
rainfall = new int [12][31];
```

- Array size can be determined at run-time;
- Can have multiple dimensions

For more complex objects, may need to take care of memory allocation:

```
Animal pets[];
String d;
pets = new Animal[12];
for (int m = 0; m < 12; m++) {
    d = in.nextInt();
    pets[m] = new Animal(d);
}
```

11.3 The enhanced `for` statment

There is a form of the **`for`** loop analogous to the **`for`** loop in Python.

```
for(variable : collection)
   statement
```

- This allows looping over an array or a collection
- The variable takes in turn the values of each value in the collection and the corresponding statement is exected.

Note:

- You can iterate over a primitive Java array.
- You can iterate over an object of any class which implements the *Iterable* interface.

```
int data [] = {1,2,3,4,5,6};
int other[];

for (int i: data)
   System.out.print(i+" ");

System.out.println();

other = new int [] {45,11,34};
for (int i: other)
  System.out.print(i+" ");
System.out.println();
```

Note the following:

- The example shows how arrays can be initialised explicitly. Note how in the one form the **`new`** keyword is used and in the other it isn't.
- The loop control variable must declared in **`for`** statement itself. Its scope is only within the loop! You may not have a variable of the same name also declared locally.
- It is a bad idea to change the value of the array/collection.
- I have illustrated enhanced for loops using arrays of integers, but the construct generalises. First, you can have arrays of other types (even arrays). Second, you can do the same thing with any class that implements the *Iterable* interface. This includes container classes like ArrayList and List.

12 Methods and method calls

All classes are broken up into methods

- Choice of methods is critical part of program design;
- Summary: Method a named, separately callable section of code.

Provides a number of key features

- control abstraction: enables us to design programs that map naturally to problem and problem solution;
- methods do one thing! 1-page rule.
- code re-use: within a program
- code re-use: provides a way we can access code written previously.

First look how a method executes and then how we call it.
The general form of a Java method is

```
modifiers  returntype  method-name ( args) {

  sequence-of-statements

}
```

- Example modifiers: **`static`** , **`public`** , **`private`** .
- Methods must have a return type describing the type of the value returned (**`void`** if none).
- The function may take arguments. Each argument must be typed. Each time the function is called, the arguments are instantiated with values by the caller.

```
int cubeabs(int x) {
   // return the absolute value of the cube
   int y;

   y = x * x * x
   if (y < 0)
      y = -y;
   return y;}
```

This could be called by

```
   ...
   myval = x -  cubeabs(3) + cubeabs(-2);
   z = cubeabs(myval);
```

Scope rules determine where a variable can be seen.

- May be many declarations of variable x. Given an occurrence of x in the program, which declaration does it refer to?

General rule:

- An identifier is in scope within the syntactic unit in which it is declared, from the point at which it is declared.
  - . . . except where hidden by an inner declaration
- Extensions to this rule for access of members of a class – will see this later.

```
public class TrivialApplication {

    static int i,j,k,x=3;

    static int moggle (int x)
       { int i, m;
         i = x-5;      m = 2*x+3*i;
         j = 7;
         return m-i;
       };

    public static void main(String args[]) {
       int i, n;
       System.out.println( "Hello World!" );
       i = 3;    j = 5;
       k = moggle(5);
       System.out.println("Values: i " +
                          i + ";  j: " + j + ";  k:" + k);
        }
}
```

12.1 Calling a method

When a method is called, the system does the following:
Set up and transfer

- makes a record of where the method is called from (so we can return there after executing the method);

- allocates space for any local variables of the method;
  NB: each call of the method (usually) has its own variables;
- allocates space for formal parameters/arguments
- passes the parameters to the method
  binds the actual parameters to the formal parameters

Execution

- Control *transfers* to the code of the method
- The code of the method is executed step-by-step until control reaches the end.

Completion and return
Once the method completes, any return value is passed back to the caller, and execution returns to the caller.

Binding of parameters

- Many mechanisms possible
- In Java, there is only one **call by value**
- Value of actual parameter copied to formal parameter;
- Change to the formal parameter does *not* change actual;
- The function can returns a value of the function/method
- Change a 'global' variable
  - Should only change instance variables of object;
  - Can be dangerous

But remember except for Java primitive types, our variables are *references* rather than the object itself.

```
void Example(int m[]) {
      m[0] = 100;
      m[1] = 200;
};
...
int m[] = {0, 1};
...
Example(m);
```

Since *references* passed, consider what the method does to determine what changes happen.

13 Dynamic and static memory allocation

Static memory allocation

- Compiler makes decision about memory allocation;
- As program starts to run, memory is allocated and stays allocated for entire run of program;
- Example: vars declared with `static`

Dynamic memory allocation

- Memory that can only be allocated when program runs
- Example: (usually) local variables to methods, objects allocated with *new*

Declaring variables statically

- Use the **`static`** keyword
- NB: only one copy of the variable exists
- Advantage: more efficient, may have right functionality
- Disadvantage: not always right semantics

Two key points to remember for static variables objects:

- *static instance variables* of a class belong to class, not individual variable — all objects access the same thing. Example: Math.PI
- with *dynamic*, instance variables belong to the individual objects not the class.

Local variables for methods

Although declared in one place, the space allocated for a variable is done dynamically.

- For each call of a method, space is allocated (the activation record, remember)

- When the method terminates, the space is freed
- For each "current" method there will be an activation record
- So if A calls B which calls C, then there will be an activation record for each method.

NB: What is "current" is determined dynamically not textually

By dynamic, I mean what is actually being executed – not just what is in the text. Just because there is a call of B inside method A doesn't mean that in each call of method A that B will also executed.

- For recursive methods, there will be an activation record for each active call to the method.

  So if A calls B which calls B which calls B then there will be an activation record for A and *three* activation

```
void A() {
   B(3);
}

void B(int n) {
   System.out.println(n);
   if (n>0) B(n-1);
}
```

Try the following example . . .

```
public class Fib {

    static int fib(int n) {
        int lhs, res, i=0;
        if (n == 0 || n == 1)
            { res = 1; }
        else {
            lhs = fib (n-1);
            res = lhs + fib(n-2);}
        i++;
        System.out.println("numcalls= "+";  i="+ i);
        return res;
    }
```

```
    public static void main(String [] args) {
        int ans;
        ans = fib(4);
        System.out.println("Result is "+ans);
    }

}
```

14 Defining classes

A class models some concepts in the real world:
Need to choose:

- Set of methods that are appropriate for the class or objects of the class;
  - Public interface
  - 'Helper' methods
- Instance or state variables
  - Usually private
  - Can be public

Usually many classes that make up a program definition

- Encourage code re-use and use of libraries

```
modifiers class {
  instance_variable_declarations;
  method_declarations
}
```

Declarations include: modifiers and type
Note: We'll see later how generics can be declared.
Scope of methods and variables

- public: can be accessed by any other class.

- private: can only be accessed by other methods of the class.
- protected: can only be accessed by other methods of the class or by subclasses.
- package: can be accessed by any other files in the same package (*the same directory*).

  Directory structure very important for Java applications

Inside a class, a member (variable or method) can be used normally.

- e.g if *name* is a variable in the *Animal class*, inside the Animal class we can just refer to *name* directly.

  So, if *SetName* is a method in Animal:

```
void SetName(String s) {
      name = s;
}
```

Outside the class, they have to be qualified either with the class name (for static members) or object name for others.

- e.g. Math.PI; Math.sqrt; g.read()

14.1 Constructors

A *constructor* is a method that is automatically called when the method is declared:

- Has the same name as the class
- Should be public
- Should not have a return type
- Can have arguments

Constructors very useful, used often.

- Can also have a destructor:

- When the garbage collector reclaims the object, **finalize** routine used;
- Useful, but not needed often in Java because the GC works well.

```
class AnClass {
      String name;
      int num_ms,measures[];

      public AnClass(String name,int ms) {
             num_ms = ms;
             this.name = name;
             measures = new int [ms];
      }

      String nameOf() {
             return name;
      }
}
```

Used as follows:

```
public class SAnimal {

    public static void main(String [] args) {
        AnClass a;
        AnClass c = new AnClass("Joe", 10);
    }
}

class Animal {
    String name, identifier;
    int    year;

    public Animal(String n, String id, int year) {
        name = n;
        identifier = id;
        this.year = year; }

    void eats() {
```

```
        System.out.println("Diet unknown"); }

    void dob() {
        System.out.println("**ANIMAL** "+name+" born "+year); }

    String ID () {
        return (name + ": (" + identifier +")");  }
}
```

Somewhere else...

```
void check() {
        Animal a;
        a = new Animal("FooBear", "F3123", 1997);
        a.dob();

        Animal b = new Animal("BlobBear", "G777", 2003);
```

14.2 Overloading methods

Can have a number of methods with the same name!

- Differentiated by different types of arguments
  - But cannot redefine on basis of return type!
- Compiler can determine which method is meant by the type information

  e.g. standard operations like +, ...
- Used often for constructors, but used elsewhere too.

```
    public AnClass(String name, int ms) {
        num_ms = ms; this.name = name;
        measures = new int [ms];
    }

    public AnClass(String name) {
        num_ms = 10; this.name = name;
        measures = new int [10];
    }

...
AnClass a = new AnClass(``penguin'',20);
AnClass b = new AnClass(``seagull'');
```

15 Inheritance

One of the key features of OOP is inheritance:

- Can define one class to be a sub-class of another
- We say a sub-class extends the super-class
- Object of sub-class has all the state information and methods of the super-class except for:
  - Any changes
  - Any additions
- Any functionality not implemented in the sub-class is inherited from the superclass
- *Powerful technique for code re-use*: if existing class almost does what's wanted, just extend definition

```
class Student {

    protected String name, faculty, cellphone;

    public Student(String n, String f, String c) {
        name =n ;
        faculty = f;
        cellphone = c;
    }

    public String NameOf () {
        return name;
    }

    public String CellNumber() {
        return cellphone;
    }

}


class ScienceStudent extends Student {
```

```
    private int pointsobtained;

    public ScienceStudent(String n, String c) {
        super(n, "science", "085 "+c);
        pointsobtained = 0;
    }

    public String CellNumber() {
        return ("Tel: "+cellphone);
    }

    public void AddPoints(int result) {
        pointsobtained += result;
    }

}


public class StudentTest {


    public static void main (String [] args) {
       Student g1 = new Student("Boole","CLaM","9874321");

       ScienceStudent s1 =
           new ScienceStudent("Turing", "7655432");

       s1.AddPoints(10);
       System.out.println(s1.CellNumber());
       System.out.println(s1.NameOf());


    }

}
```

16 The Java API

Java comes with libraries — classes organised in packages.

- "Standard libraries": eg. java.*
- Sun

- GCC
- Other

For example, `java.net` has the `URL` class.

- Packages can be hierarchical: `org.xml.sax.helpers`

To be an effective Java programmer you must know some basic packages well and have an idea of what else is there. Otherwise, you are just re-inventing the wheel.

16.1 Some key Java classes

`Arrays`

- Searching, sorting, initialising

Collection classes:

- ArrayList
- Stack
- Hashtable

StringTokenizer

17 Generics

Generics allow us to parameterise classes with types

- When a class is declared, type parameters are specified
- Allow us to generalise the class
- When the class is used, it is *instantiated* with a specific type.

Introduced in Java 1.5 — can do without it, but makes programming easier and safer.
We look at an example first, to show why it is useful.

One of the standard Java classes is the `Stack` class.

- In the `java.util.Stack` package.
- Can create
- pop
- push
- empty
- peek

Example that works:

```
Stack s = new Stack();

s.push("apple");
s.push("pear");
s.push("banana");
s.push("cherry");
while (! s.empty())
   System.out.println(s.pop());
```

Produces in order: cherry, banana, pear, apple
Example that doesn't work

```
String z;

s.push("apple");
s.push("pear");
s.push("banana");
s.push("cherry");
while (! s.empty()) {
   z = s.pop();
   System.out.println(z);
}
```

These two examples are another example of the difference between compile-time and run-time type checking. The `println` method can print *any* type: you might just get the address stored as the reference, but it is legal to print any variable of any type.

First example In the `Stack` class, the `pop` method returns an object of compile-time type `Object`. The compile-time system is happy because there is a `println` method for objects of type `Object`. No problem. At run-time, the run-time system can see that the object is actuall a `String` and so calls the `println` method defined on the `String` class (actually the `println` method calls the `toString` method) and so we get useful results.

The second example fails because the compile-time system in not happy to allow you to assign an `Object` reference to a `String` variable. So even, though the underlying type is a `String` the compile time system fails.

Remember, that objects of the Stack class can be objects of anything: there is no requirement that they are all of the same type: you can have a stack that contains a string and animals and so on...

The solution is that you have do type coercion.

```
while (! s.empty()) {
    z = (String) s.pop();
    System.out.println(z);
}
```

You tell the compile-time system what the underlying type is.

- Seldom need a stack that is truly heterogeneous.
- Over-uses polymorphism.
- Unsafe – the compile-time system can't check
- Big, big schlep to have to type-coerce – very ugly code

So, use generics: parameterise generics.

- classes can be given a *type parameter* — in angle brackets.

```
Stack<String> s = new Stack<String>();

s.push("apple");
s.push("pear");
s.push("banana");
s.push("cherry");
while (! s.empty())
    System.out.println(s.pop());
```

We use angle brackets as a way of specifying the type. In the example above, `s` is Stack of Strings only.

Generics are very useful, and if used simple are fairly easy to use. However, there are complexities if you are doing complex things, when creating your own generics. If doing so, you should read up more about generics.

18 What else?

Will see

- Javadoc
- Junit
- Interfaces
- Iterators
- Exceptions

Lots not covered:

- Regular expressions (`Pattern`)
- Windows,
- GUI,
- applets,
- multithreading,
- networking,
- Java Beans, ...

19 References

- Preiss. *Data Structures and Algorithms with Object-Oriented Design Patterns in Java*.
- Bishop. *Java Gently*
- Cay S. Horstmann; Gary Cornell. *Core Java 2 Volume I - Fundamentals, Seventh Edition*. Prentice-Hall
- David Flanagan. Java in a Nutshell, 5th Edition. O'Reilly.
- Goodrich *et al.* Data structures and algorithms in Java.
- `http://ug.ms.wits.ac.za/~coms2003/api`
- `http://ug.ms.wits.ac.za/~coms2003/javadocs`
- `http://java.sun.com/docs/books/tutorial/index.html`