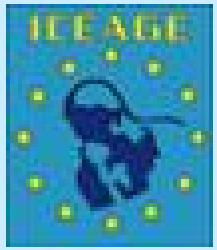
ISSGC
The 4th International
Summer School on Grid Computing



Refresher

Richard Hopkins

rph@nesc.ac.uk

Context

Aim is to give/re-new

- enough understanding of Java to get through the school
  - To write bits of Java yourself
  - Understand bits of Java written by us / you colleagues
- a wider appreciation of the capabilities of Java

- Assume you have some experience of programming in some object-oriented language – Can't teach you the O-O paradigm

What you get is

- This Lecture + supporting material

Supporting Material

- From us

  http://www.gs.unina.it/~refreshers/java

  - Presentation.ppt — This presentation
  - Tutorial.html — A tutorial for you to work through
    Includes a complete example
    Illustrating most of what is covered
    with some exercise for you to do on it

- From elsewhere
  - "Thinking in Java", Bruce Eckel - http://www.mindview.net/Books/TIJ/
  - Java Tutorials - http://www.cas.mcmaster.ca/~lis3/javatutorial/
    http://java.sun.com/docs/books/tutorial/
  - Java APIs reference documentation –
    http://java.sun.com/j2se/1.5.0/docs/api/index.html

Outline

General

- **Introduction to Java**
- Classes and Objects
- Inheritance and Interfaces

Detail

- Expressions and Control Structures
- Exception Handling
- Re-usable Components

Practical

Reference Material

Welcome to the Java World (1)

Goal - Interoperability – the same code runs on any machine/O-S

- The Java compiler produces “bytecode” binary –
  - “Machine code” for the Java Virtual Machine (JVM)
  - Executed by an interpreter on a physical machine

- The same compiled code can be executed on any hardware and software architecture for which there is a JVM interpreter (run-time environment)

- Java is freely downloadable from Sun website
  - Java Development Kit (JDK)
  - Java Runtime Environment (JRE)
- JDK & JRE are not Open Source, but an Open Source implementation is avalaiable (Kaffe)

Java Pluses

- Object-Oriented
  - Everything is an object
  - Multiple inheritance in a restricted form
- Architecture independent
  - The language itself
  - The library (platform) of 1,000+ APIs
- Secure – JVM provides a layer between program and machine – safely execute un-trusted code
- Robust
  - No pointers, only references
  - Dynamic array bound checking
  - Strongly typed
  - Built-in exception-handling mechanism
  - Built-in garbage collection
- Power …

  …and Simplicity
  - Easy to learn (for someone who understands O-O paradigm)

Java Minuses

Those who don't like Java, don't like it because of

- Execution Inefficiency
  - Interpreted (but Just in time compilation helps)
  - Garbage collection
  - Dynamic array-bound checking
  - Dynamic binding
  - Don't use it when timing/performance is critical
- Error diagnostics
  - Full stack trace
- The dreaded CLASSPath

Java@work

- What you need:
  - Java Development Kit
  - A text editor
    - vi or notepad are enough
    - jEdit is a dedicated editor (developed in Java)
    - Netbeans and Eclipse are powerful, free IDE (Integrated Development Environment)
    - Commercial tools: JBuilder, IBM Visual Age for Java

Basic Syntax – Structure, comments

```
// A very simple HelloWorld Java code
public class HelloWorld {
  /*  a simple application
   *   to display
   *  "hello world"  */
  public static void main(String[] args) {
    System.out.println("Hello World!");
  }// end of main
}
```

- Code is structured using
  curly brackets { ...}
- Each non-{ } statement ends with
  a semicolon (as in C/C++)
- Single line comment,
  from // to end of line
- Multi line comment,
  from /* to */

- Identifiers, examples –
  i engine3 the_Current_Time_1 MyClass myString

Rules and conventions at end

- Java is not positional:
  carrige return and space sequences are ignored
  (except in quoted strings and single line comments)
  so lay-out for ease of reading

Basic Syntax – Primitive Types

Type	Size (bits)	Example literals
boolean	1	true false
char	16	'A' '\"' '\\' '\n' '\r' '\u05F0' '\t'
byte	8	integer: -64 123 073 0x4A2F 9223372036854775808L
short	16	
int	32	
long	64	
float	32	floating point: 11.3E-4 73.45 -1.45E+13
double	65	

\n - newline
\r - return
\t - tab
\u -Unicode – 4 hex digits

initial 0 - octal
initial 0x or 0X - hexadecimal
Final L – long

- Default values – 0 (= false)

Basic Bureaucracy


```
// A complex HelloWorld Java code
public class HelloWorld {

 public static void main(String[ ] args)
{
    <code for printing out a greeting>
  }
 class greeting { <method defintions> }
<other class defintions>
}
```

A java "program" consists of

- A public class (HelloWorld)
  - with a "main" method
  - with an array of strings as parameter
    - For the command line arguments
- Other classes

The program is in one or more files

Each file has at most one public class – same name – HelloWorld.java

HelloWorld.java

Steps

- Create/edit the program text file, e.g.
  $ vi HelloWorld.java
- Compile using the command
  $ javac HelloWorld.java 2>HW.err
- Run using the command
  $ java HelloWorld
  (this runs the java virtual machine)

For now

- public = externally accessible
- Otherwise only accessible from within same class definition

CLASSES and OBJECTS

General

- Introduction to Java
- **Classes and Objects**
- Inheritance and Interfaces

Detail

- Expressions and Control Structures
- Exception Handling
- Re-usable Components

- Practical
- Reference Material

Classes and Objects

- A class represents an abstract data type
- An object is an instance of a class
- A class has constructor methods whereby an instance of the class can be created
- A class has attributes – instance variables
- Each instance of a class has its own value for each attribute
- A class has methods
- Every instance of a class can have each method applied to it

Accumulator Example - Definition

- Accumulator
  - Keeps a running total
    - Which can be incremented
  - Tracks how many times it has been used

```
public class Accumulator {
//attributes
double total = 0.0;
int uses = 0;
// methods
public double incr (double i) {
// doing it
uses  =  uses+1;
total = total + i;
return total; }
}
```

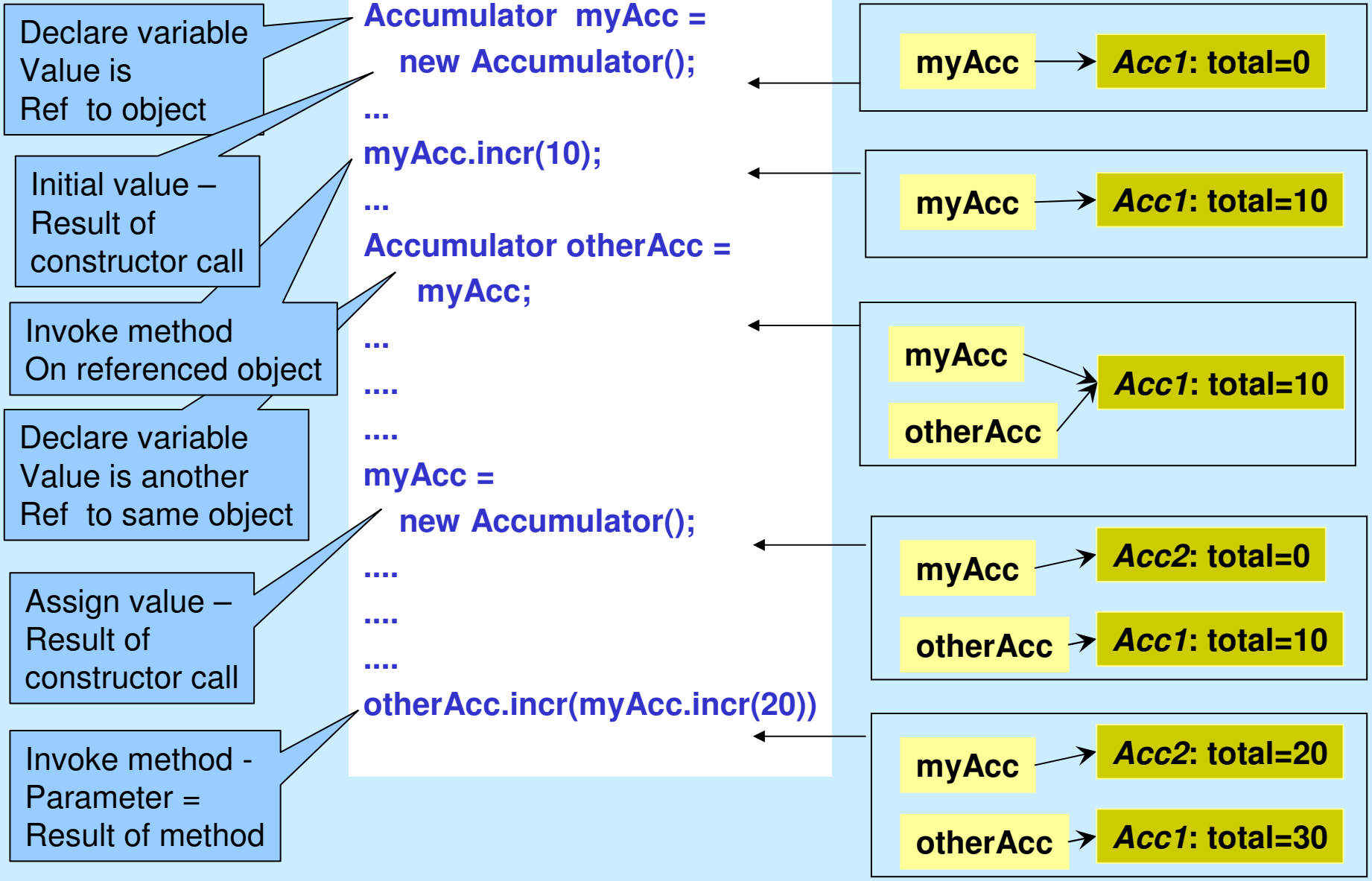
Attributes –
[Visibility] (optional)
type
name
[Initial value]

Method –
[Visibility]
type
name
Parameter * (repeated)
type
name
Body – statement *

Assignment

Exit with result

Accumulator Example - Usage



Values and their Usage

- For – A variable
  A parameter
- Its type is either
  - Primitive – holds a primitive value.
    - Can be used in expressions
    - Can be produced by expressions
  - Reference - holds a reference to an object
    - Can be copied to a variable / parameter
    - Can be produced by constructor call
- Assignment To = From
  - To gets a copy of value of From
    for objects - another reference to same object
  - Same for parameter passing

All simple and intuitive

Unless you are used to a language
with more sophisticated pointers/references !

Special Cases

- null – a reference value that doesn't reference anything
  - Default for references
- this – references the object itself – an implicit parameter to every method, referencing the object on which the method was called
- void – The "nothing" type

Constructors

- The new <class-name> is a method call on a class constructor method
- We can define constructors for doing object initialisation – constructor is a method whose name is the class name

Constructor - gives Initialisation value Implicitly returns object reference

```
public class Accumulator {
double total;

public Accumulator (double i) {
total =  i;}
}
```

usage

```
myAcc =
    new Accumulator(10);
```

- If no constructor declared – get a default one with no parameters which does nothing (except initialise values of class variables)

Constructors (2) & Method Overloading

- Two constructors – one with specified initial value; other with default

Constructor - gives Initialisation value

Constructor - Omits Initialisation value Uses default

```
public class Accumulator {
double total;

public Accumulator (double i) {
total =  i;}

public Accumulator () {
total =  0.0;}
}
```

usage

```
Accumulator  myAcc;
myAcc.incr(10);
myAcc =
    new Accumulator(10);
....
myAcc =
    new Accumulator();
```

- Two methods with same name – Method Overloading
- Must have different “signature” – number and types of parameters
- So which one to use is determined by what parameters are supplied
- General feature, not just constructors

External attribute access

- As a general rule the state of an object should be accessed and modified using methods provided by the class - Encapsulation
  - You can then change the state representation without breaking the user code
  - User thinks in terms of your object's functionality, not in terms of its implementation
- However, if you insist, you can make attributes more accessible – e.g. public

```
public class Accumulator {
public double total = 0.0;
int uses = 0;
// methods
public double incr (double i) {
// increment the total
uses  =  uses+1;
total = total + i;
return total ;}
}
```

usage

```
Accumulator  myAcc =
    new Accumulator();
...
myAcc.total = 10;
....
myAcc.total =
   myAcc.total + 6;
```

Better to have new method – myAcc.reset(10)

(Accidentally) by-passes uses update

Better to use myAcc.incr(6)

Static Variables and Methods

- Normally, have to have an instance
  - An attribute declared for a class belongs to an instance of that class
    - An instance variable
  - A class method can only be invoked on an instance of that class

- Can declare an attribute / method as static
  - Something that relates to the class as a whole, not any particular instance
  - Static variable
    - Shared between all class instances
    - Accessible via any instance
    - Accessible via the class
  - Static method
    - Can be invoked independent of any instance – using class name
    - Cannot use instance variables
    - “main” method must be static

- Think of there being one special class instance holding the static variables and referenced by the class name

Statics – Examples (1)

```
public class Accumulator {

public static double defaultInit;
  // default intial value for total
static int count = 0;
  // number of instances
double total = defaultInit;

public double incr (double i) {
...}

public Accumulator () {
count  = count + 1; }

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

defaultInit – static variable, to configure accumulator with the default initial value for new ones

count – static variable -to Track number of accumulators that exist

Constructor – static method, updating static variable

howMany – static method – accessing static variable

Statics – Examples (2)

```
public class Accumulator {

public static double defaultInit;
  // default intial value for total
static int count = 0;
  // number of instances
double total = defaultInit;

public double incr (double i) {
...}

public Accumulator () {
count  = count + 1; }

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

usage

```
Accumulator.defaultInit = 100;
...
Accumulator  myAcc =
    new Accumulator();
...
int i = Accumulator.howmany();
...
int j = myAcc.howmany();
....
myAcc.defaultInit=30;
```

Using class name

Using instance

Constants (Static)

- Can define a constant using final – can't do anything more with it

```
public class Accumulator {
public static final double root2 = 1.414;
static int count = 0;

public double incr (double i) {
total = total + i;
return total;}
}
```

usage

```
...
myAcc.incr(Accumulator.root2);
```

- A generally useful constant provided by this class – can use anywhere
  - Public – part of the external functionality
  - Static – not instance-specific
- Whenever particular values are used in a class interface they should be provided as constants – coded values, e.g "/" as separator in file name paths
- As a substitute for enumerated types

  http://www.javaworld.com/javaworld/jw-07-1997/jw-07-enumerated.html

Constants (Non-static)


```
public class Accumulator {
public static final double root2 = 1.414;
static int count = 0;
final double defaultIncrement = count;

public double incr (double i) {
total = total + i;
return total;}

public double incr () {
total = total + defaultIncrement;
return total;}
```

Instance constant – each object has its own value, evaluated at object creation

- Method Overloading again

```
Accumulator myAcc =
    new Accumulator();
myAcc.incr(10);
...

....
myAcc.incr();
```

Kinds of variable

- Attributes ("fields")
  - Class variables– one for the class, shared between instance
    - Static or non-static (i.e, constant or variable)
    - Created and intialised when class loaded
  - Instance – separate one for each object
    - Static or non-static (i.e, constant or variable)
    - Created and intialised when class loaded
  - Has default initial value of 0 or null

- Local Variables
  - At any point can define a new variable
    - int temp = 0;
    - Does not have default initial value – un-initialised error
- Parameters
  - acts like a local variable, initialised by the actual parameter

Life and Death - creation

- An instance object is created by invocation of a constructor –

  new Class(…)

  This creates and initialises all the instance (non-static) variables and constants

  If not explicitly initialized, instance variable have default initial value 0 or null

- What about the Class object
  - The home for class (static) variables and constants
  - The target for static methods
  - This is created in the beginning
    - Before any instances are created (except in strange circumstances)
    - Typically when the class is loaded into the JVM
  - That's when class variables and constants are created and initialised
  - Can put in explicit class initialisation code

Life and Death - Destruction

- Java VM does garbage collection
- An object instance is destroyable when
  - Nothing references it
  - Therefore it cannot be accessed
  - Once an object becomes destroyable, the garbage collector may eventually destroy it
    - That releases the memory resources used by the object
- To enable early release of resources, destroy references
- Can put in additional finalisation code

```
Accumulator myAcc =
    new Accumulator();
...
myAcc.incr(10);
....
myAcc = null;
....
```

INHERITANCE and INTERFACES

General

- Introduction to Java
- Classes and Objects
- **Inheritance and Interfaces**

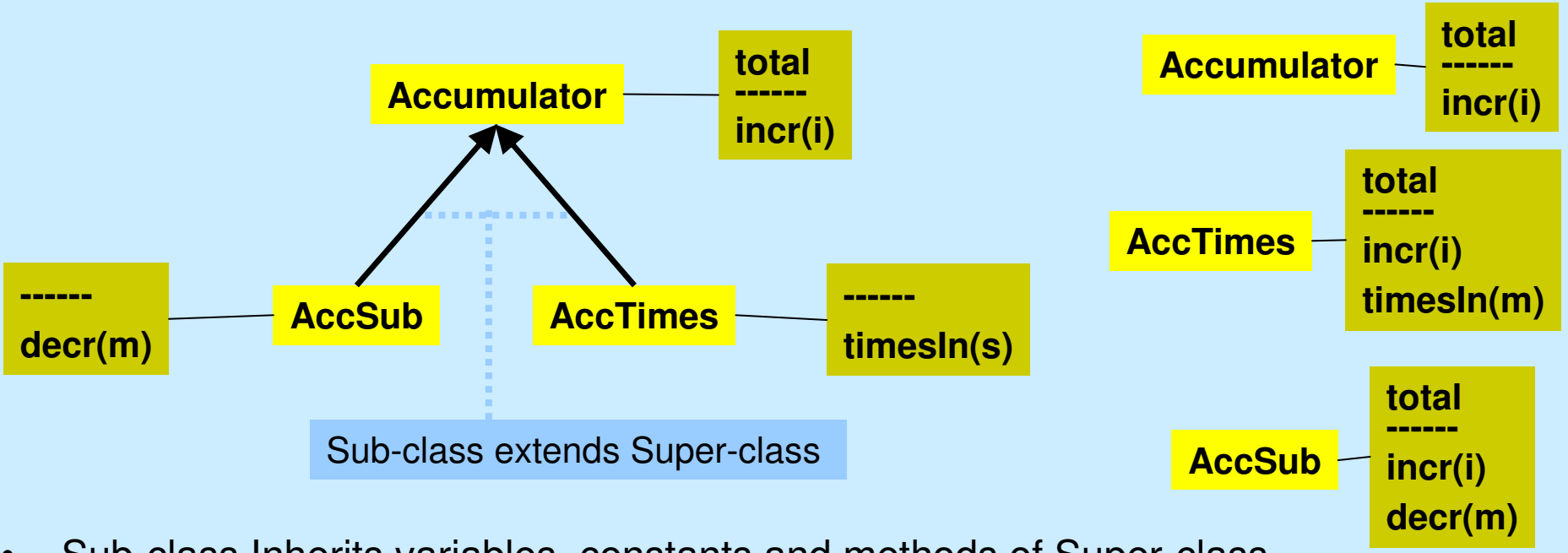
Detail

- Expressions and Control Structures
- Exception Handling
- Re-usable Components

- Practical
- Reference Material

Class Inheritance – principles

- Extended versions of Accumulator
  - AccSub : include method decr(s) – total = total – s
  - AccTimes : include method timesIn(m) – total = total * m



- Sub-class Inherits variables, constants and methods of Super-class
- Sub-class instance can be used any where a super-class instance can
- So inputs to sub-class must include inputs to super-class
  outputs from super-class must include outputs from sub-class

Class Inheritance – Simple Extension

```
public class Accumulator {
double total = 0.0;
public double incr (double i) {
total = total + i;
return total; } }
```

Inherits
total – type and initialisation
incr – signature and implementation

May be in a different file. Inherit from library classes

```
public class AccSub extends Accumulator{


public double decr (double s) {
total = total - s;
return total;
}}
```

Only need what is new

Class Inheritance – Method Overriding

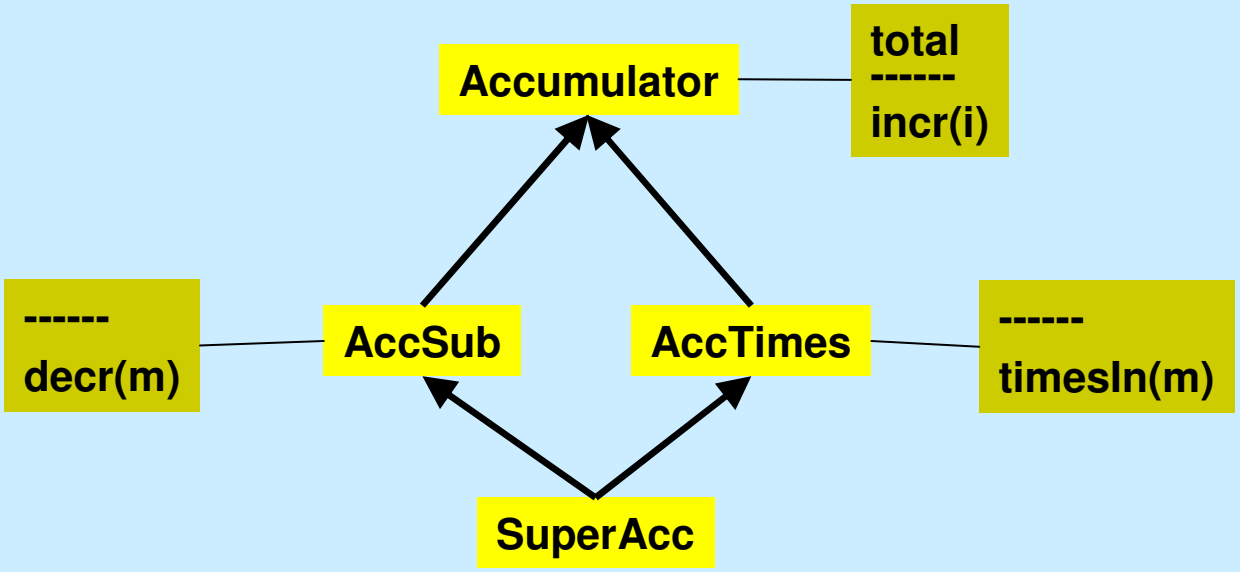
- Inherit the signature, but override the implementation

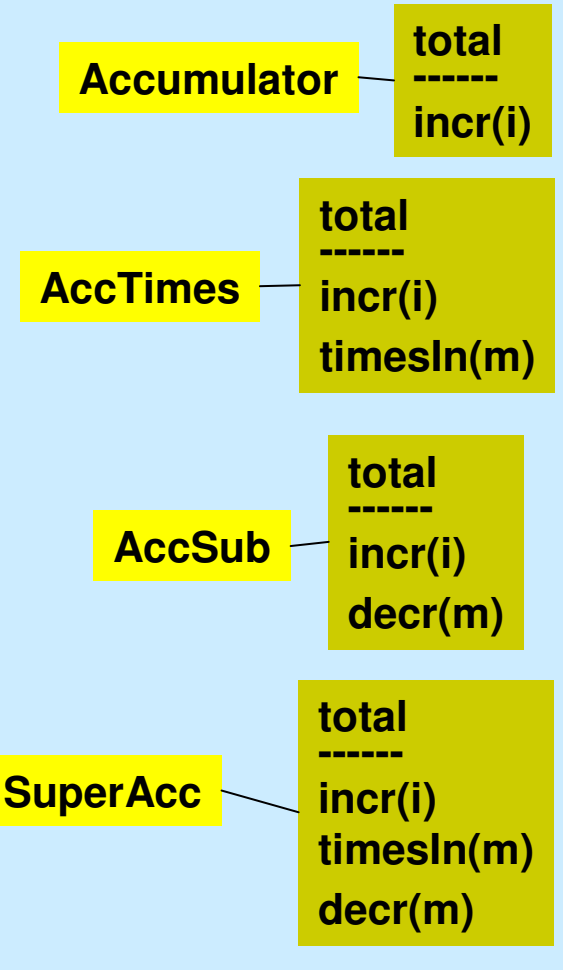
```
public class Accumulator {
double total = 0.0;
public double incr (double i) {
total = total + i;
return total; } }
```

```
public class AccSub  extends Accumulator {
public double decr (double s) {
total = total - s;
return total;}
public double incr (double i) {
return this.decr(-i); }}
```

Incr is re-implemented using **decr**

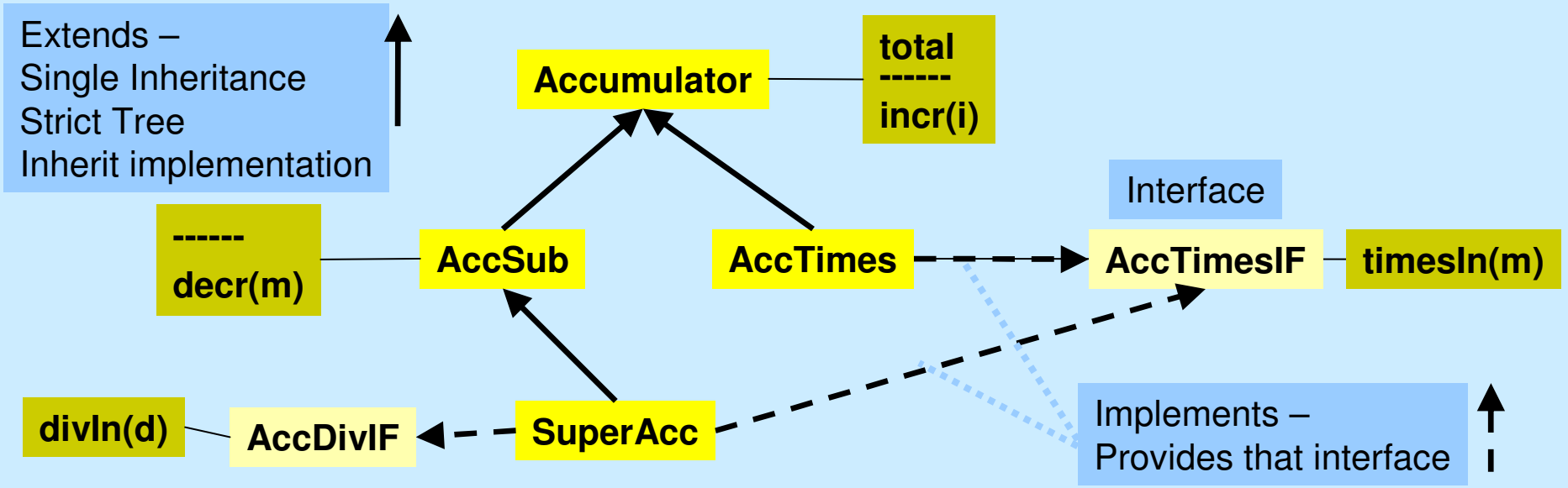
Multiple Inheritance





- SuperAcc inherits from both
  AccSub and AccTimes
- Problem –
  - inherits from Accumulator on two distinct paths
  - what if AccSub and AccTimes both have implementation of incr()
  - Which one does SuperAcc use?

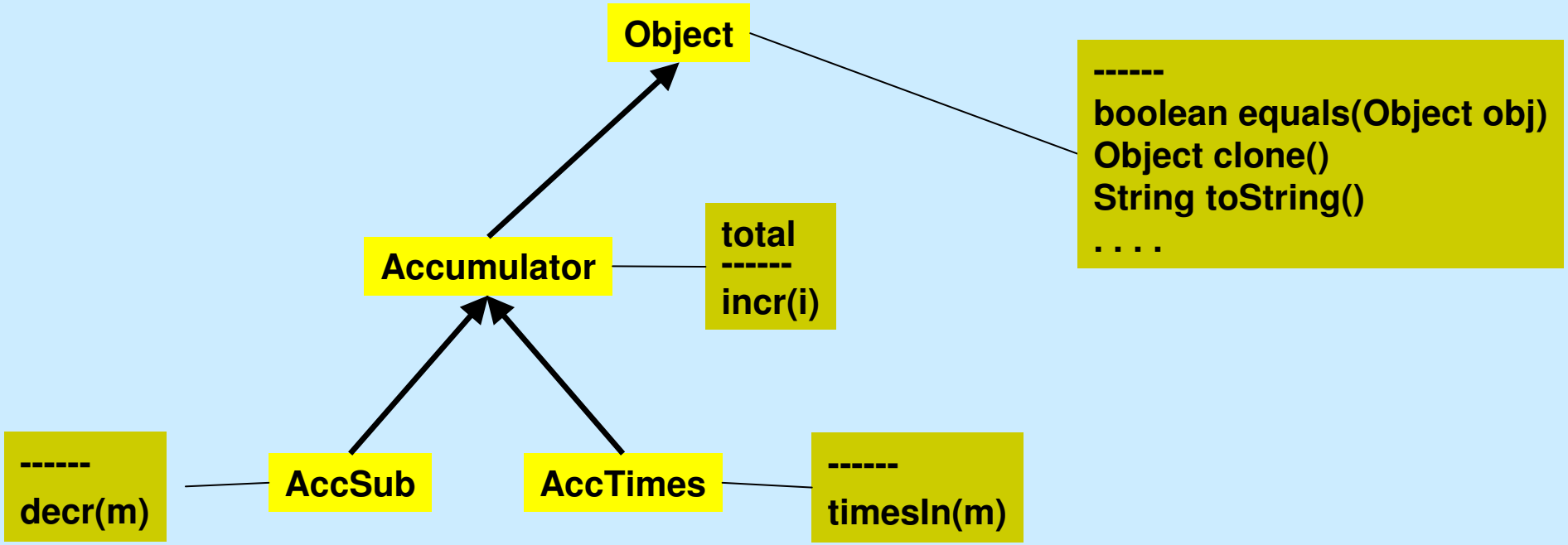
The Java Solution - Interfaces



- The Interface defines
  - Zero or more method signatures
  - Zero or more static constants
- A class can implement several interfaces

The Object Class

- The root of the class hierarchy is “object” – every object is an Object



- equals – test two objects for
  - Identicality – default implementation test for them being the same object
  - Equivalence – maybe overwritten test for something more useful
    - Two accumulators are equivalent if they have same total
- Clone – makes a copy of the object – default implementation gives shallow copy
- toString – to give a displayable representation

Wrapper Classes

- To make a primitive data type into an object

primitive	wrapped
boolean	Boolean
char	Character
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double

Integer IntegerConst = new Integer(17)

Provide useful methods, e.g.

Int input= Integer.parseInt(aString)

See class Integer etc. in Java APIs

Reference Type Conversion

- Widening – can always treat an object as instance of a superclass

```
Object object;
Accumulator accumulator;
SuperAcc superAcc;

superAcc = new SuperAcc();

object = superacc;

accumulator = superacc;

superAcc = object;

object.incr(1);

superAcc= (SuperAcc) object;

( (SuperAcc) object).incr(1);
```

Is a (instance of)

Widening – moving it up the class hierarchy

Narrowing – moving it down the class hierarchy
Compiler can't know object
references an object of class superAcc

A cast for narrowing –Tells the compiler that the thing referenced by object is an instance of SuperAcc
Compiler believes me
If wrong – run-time exception

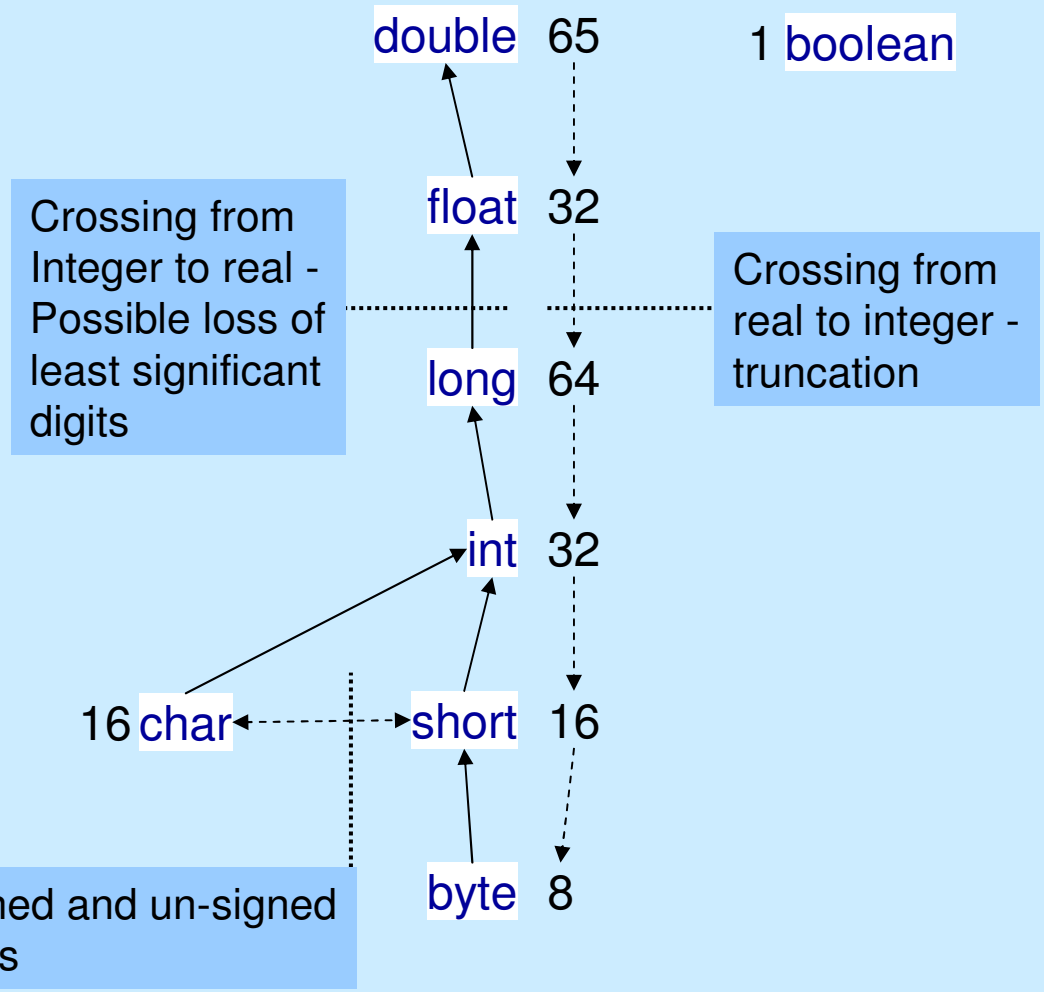
Primitive type Conversion

- Can automatically widen
  a smaller type
  to a bigger one

aFloat = aByte

- Cast
  a bigger type
  to a smaller one

aByte = (byte) aFloat



EXPRESSIONS AND CONTROL STRUCTURES

General

- Introduction to Java
- Classes and Objects
- Inheritance and Interfaces

Detail

- **Expressions and Control Structures**
- Exception Handling
- Re-usable Components

- Practical
- Reference Material

Basic Operators

- * / % multiply, divide, remainder
- + – plus, minus
- + – unary plus, unary minus
- + **string concatenation**
- > >= < <= comparison
  == !=
- ! || && ^ boolean – not, or, and, exclusive or
  (for || and && - conditional evaluation of 2nd argument)

- ++ – – post increment/decrement n++
- ++ – – pre increment/decrement ++n
- + = – = assignment with operation n += 10 (n=n+10)

Expressions

- Precedence and associativity – as expectable
  - When you (or your reader) could be in doubt – use brackets

- Return results

  Every expression returns a value – including an assignment expression

  a = b+= c=20

  right to left associativity – a = (b+= (c = 20))

  assign 20 to c; add the result into b; and assign that result to a.

Conditions

- **Conditional expressions**

```
(x>y ? x : y)  = 4 + ( j > 0 ? k+n : m+o) * 2
```

If x > y assign to x, otherwise assign to y

If J>0 use k+n, else use m+o

- **Conditional statements**

If condition gives trueThen do this

Can omit else

```
if (x>y && j >0)
        { x = 4 + (k+n)*2; }
else if (x>y)
        { x = 4 + (m+o)*2; }
else if (j>0)
        { y = 4 + (k+n)*2; }
else
        { y = 4 + (m+o)*2; }
```

- Conditional expressions can reduce repetition
- Reducing repetition usually makes things
  - Clearer
  - More robust

Switch statements

- Expression based choice over alternatives

```
public class Accumulator {
double total = 0.0;
static char doAdd = `a`;
static char doSub = `s`;
static char doMult = `m`;
public double doAction (byte action, double value) {
   switch (action) {
                case `A` :
                case `a` : total = total + value; break;
                case `S` :
                case `s` : total = total – value; break;
                case `M` :
                case `m` : total = total * value; break;
                default : ... }
   return total ; }
```

myAcc.doAction(`S`, 20)

Evaluate switch expression = ‘S’

Choose case where constant Matches switch value

Fall through

If no match

So, “break” to exit whole switch

While and Do Statements

```
public double powerIn(int p) {
                    // if p<2, do nothing
        double  base=total;
        while (p>1)
        {           total = total * base;
                    p=p-1; }
  return total ; }
```

while
<condition>
<statement>
May do it zero times

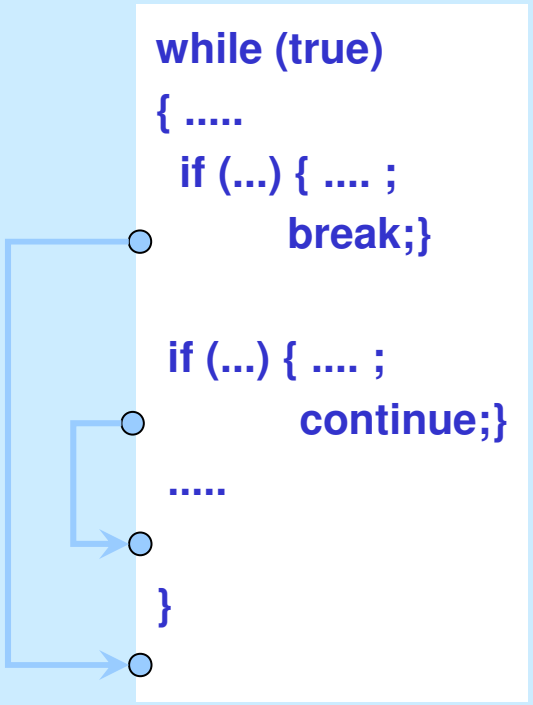
```
public double powerIn(int p) {
                    // assumes p>=2
        double  base=total;
        do
        {           total=total * base;
                    p= p-1; }
        while (p>1);
  return total ; }
```

do
<statement>
while
<condition>
Does it at least once

For Statements, break and continue

```
public double powerIn(int p) {
                    // if p<2, do nothing
        double  base=total;
        for ( int i =2;  i <= p;  i++)
                    total = total * base;
  return total ; }
```

for
(<intialise> // assignment
  <test> ; // boolean
 <update>) // assignment
<statement>
May do it zero times



break – jumps to just after the whole thing – terminate it

continue – jumps to just after the current iteration –

start next iteration if test succeeds

do <update> in for loop

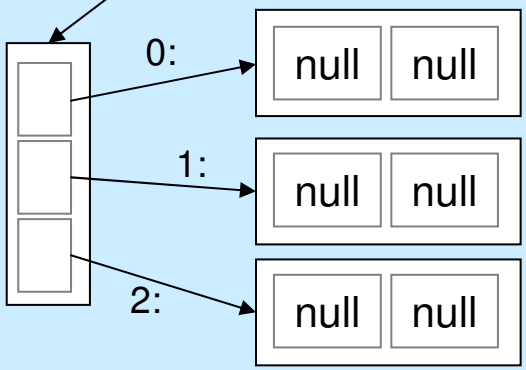
Arrays – declaring and creating

- An array is an object with specialised syntax

Gives a variable for a reference to an array of accumulators – No value yet

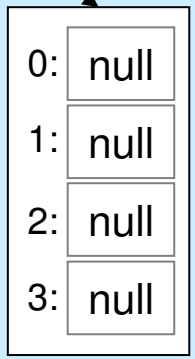
```
Accumulator [ ] myArrayOfAcc;

Accumulator [ ] [ ] my2DArrayOfAcc;

myArrayOfAcc = new Accumulator [4 ] ;

my2DArrayofAcc = new Accumulator [3] [2] ;
```

Gives a variable for a reference to an array of references to arrays of accumulators



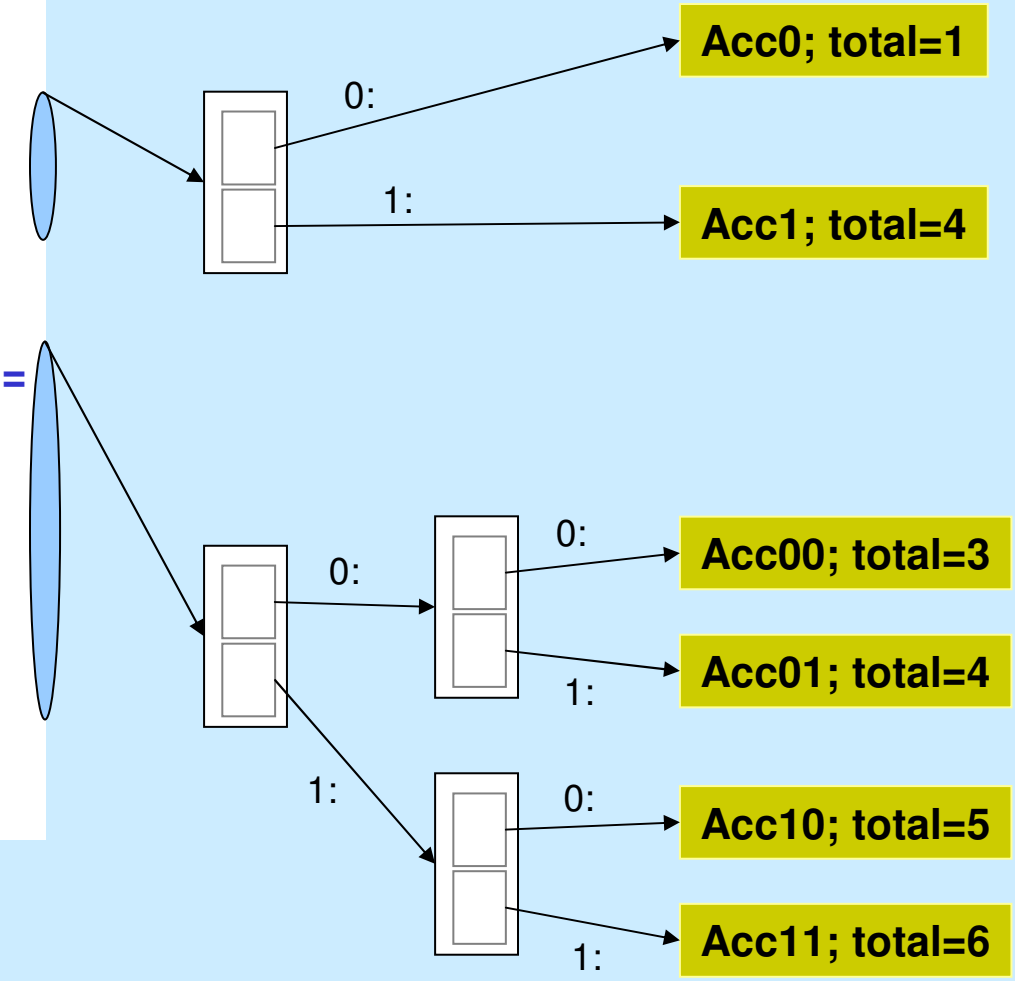
Gives 3-element Array of references to new 2-element arrays

Gives 4-element Array of Appropriate default values – Null or 0
Indexed: 0 – 3



Arrays - Initialising

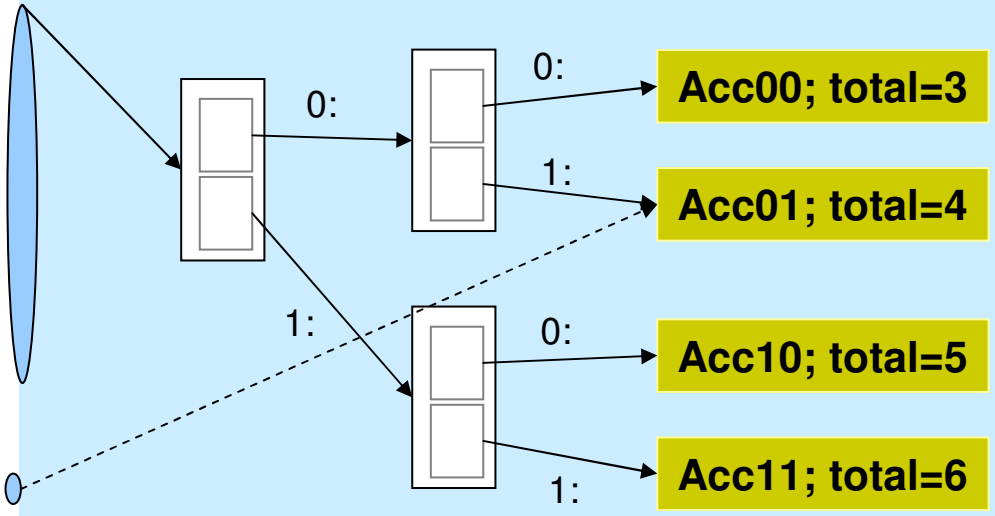
```
Accumulator [ ] myArrayOfAcc =
  { new Accumulator(1),
    new Accumulator(4) };

Accumulator [ ] [ ] my2DArrayOfAcc =
  {
    { new Accumulator(3),
      new Accumulator(4)   } ,
    { new Accumulator(5),
      new Accumulator(6)   }
  } ;
```



Arrays - accessing

- someArray [i] gives the i-th element
- someArrayOfArray [i] [j] means
  (someArrayOfArray [i]) [j] gives the j-th element of the i-th element
- someArray.length the array length
- someArray[i].length the length of the i-th componenet array

```
Accumulator [ ] [ ] my2DArrayOfAcc =
  {
     { new Accumulator(3),
       new Accumulator(4)    } ,
     { new Accumulator(5),
       new Accumulator(6)    }
  } ;
...
(my2DArrayOfAcc [ 0 ] [ 1 ]) . incr(2)
```



EXCEPTION HANDLING

General

- Introduction to Java
- Classes and Objects
- Inheritance and Interfaces

Detail

- Expressions and Control Structures
- **Exception Handling**
- Re-usable Components

- Practical
- Reference Material

Important exception handling concepts (1)

- "exception" means exceptional event – something that disrupts the normal instruction flow
- Use try-catch construct
- Cause the event by throw ing an exception, inside a "try" block
- Detect the event by catch ing the exception, inside a "catch" block

- What is thrown and caught is an exception object
  - Represents the causing event
  - Has a type – the kind of event that happened
  - Has fields – further information about what happened

- There is a class hierarchy of more specialised exception types
  - The top (least specialised) is type Throwable
  - You can declare your own exception type –
    - must extend from (a sub-class of)Throwable

Important exception handling concepts (2)

For some types of exceptions

- The exceptions that can be caused within a method,
  and are not caught by the method itself,
  must be declared as part of the method signature

- An exception is a possible output – inheritance rules apply
  - A sub-class must not introduce more exceptions than its super-class
  - I should be able to safely use the sub-class anywhere I can use the super-class

Exception-throwing example

```
public class AccBadParam
   extends Throwable {...};
public class Accumulator {
...
public double powerIn(int p)
      throws AccBadParam
{
  //previously -  if p<2, do nothing – now exception
 if (p<2)
     throw new AccBadparam
                ("powerIn(p) requires p>=2");
           double  base=total;
           while (p>1)
           {          total = total * base;
                      p=p-1; }
   return total ; }
}
```

Declares a new kind
Of exception

Declares that this method
throws that exception

Constructs and throws
an exception object
Inherits constructor
with message parameter (string)

Control jumps out to
the innermost active try block
which catches
  AccBadParam
or a super-class of it

Exception-catching example

Something
In here
(or called in here)
throws
exception

Catch it, declares a variable
to hold the exception object

```
…
Try {
….
myAcc1.powerIn(n);
….}
catch (AccBadParam e1)
    throw new
       otherExceptionType("...”+e1.getMessage());
catch (Xexception e1)
   { // exception recovery
   }
```

Catch clauses
Are checked
In this order

If none match
then check
containing
try/catch
constructs
In this method
or in
calling method
Etc

Catch some other possible exception

Convert exception
To something
understandable
At outer level

Re-usable Components

General

- Introduction to Java
- Classes and Objects
- Inheritance and Interfaces

Detail

- Expressions and Control Structures
- Exception Handling
- **Re-usable Components**

- Practical
- Reference Material

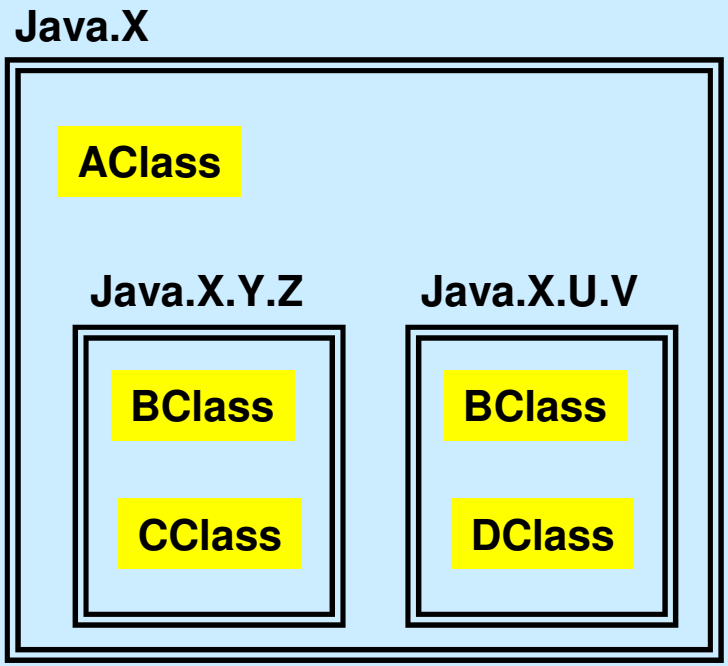
Packages and Naming (1)

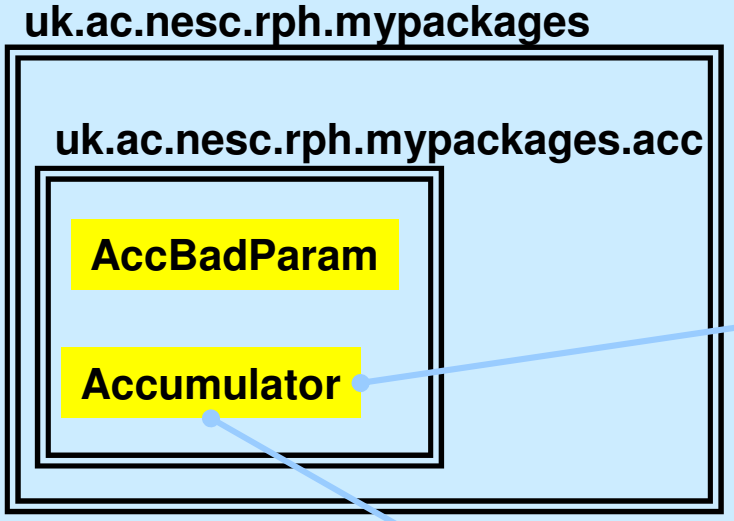
- A major point of OO is to have lots of classes that can be re-used
- Just the Java Platform has over 1,000 classes
- Each class can have many associated named entities
  - Methods
  - Class/Instance Variables
  - Constants

- This leads to a naming problem
  - How to ensure that names are unambiguous

- Solved by having a hierarchy of named packages
  - Each package has a number of classes in it
  - Provides a local namespace for those classes
  - Can have sub-packages
  - Use your domain name (reversed) to prefix your package names

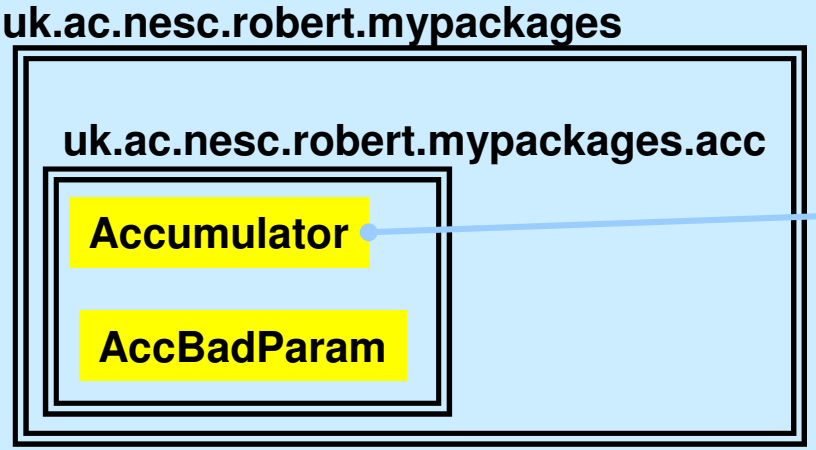
Packages and Naming (2)





Simple Class name **Accumulator**

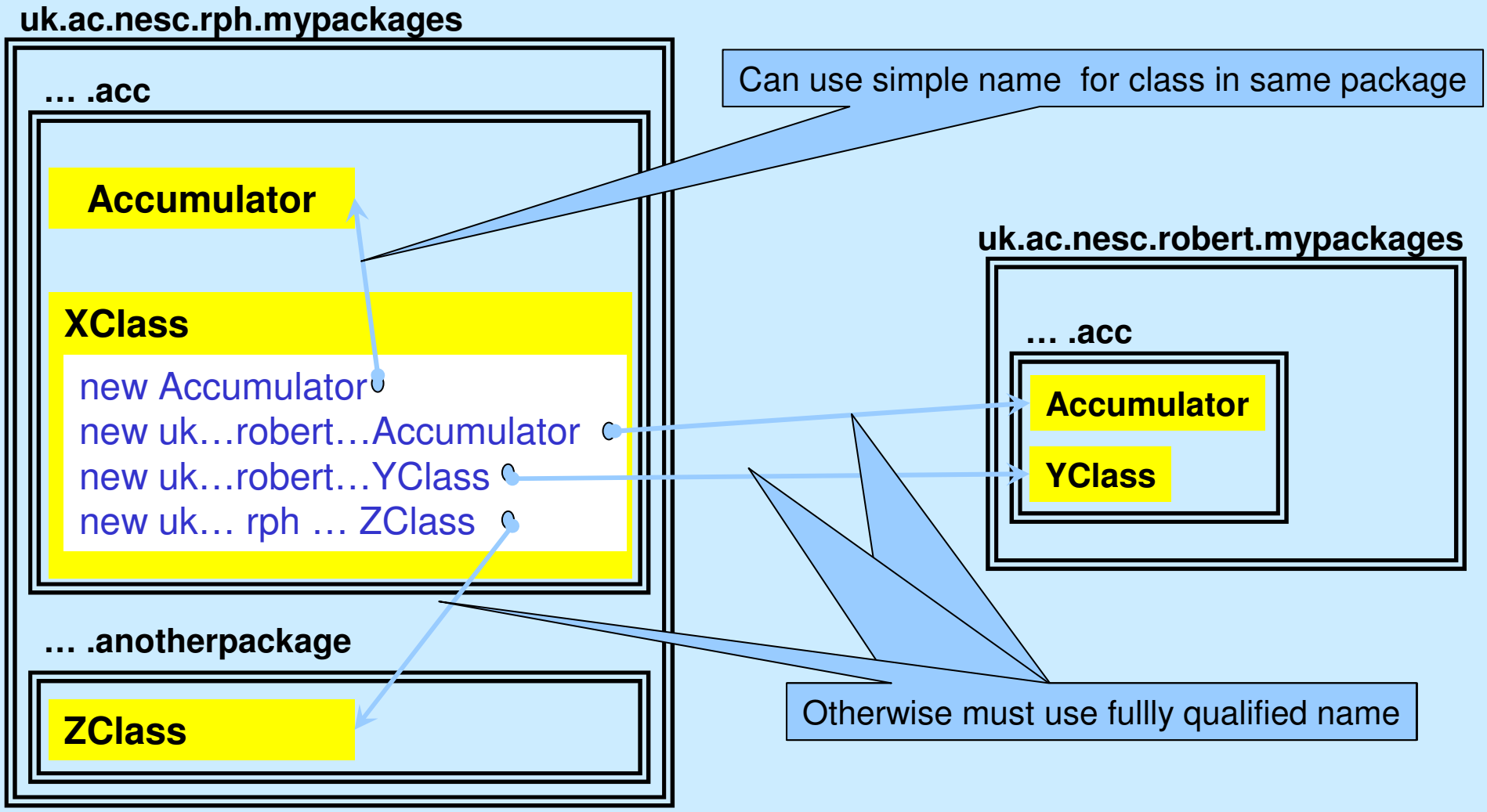
Fully qualified class name
uk.ac.nesc.rph.mypackages.acc.Accumulator



Fully qualified class name
uk.ac.nesc.robert.mypackages.acc.Accumulator

package

Naming Rules



Except that classes in the java.lang package can always be referred to by simple name
e.g. String vs java.lang.String

Imports (1)


```
new Accumulator

new uk.ac.robert.mypackages.acc.Accumulator

new uk.ac.robert.mypackages.acc.YClass

new uk.ac.rph.mypackages.anotherpackage.ZClass
```

- Using fully qualified names for classes from external packages could get to be inconvenient
- Can import a class form a package once
  - Then can refer to it by simple name,
    - Provided there is not another imported class with the same simple name

Imports (2)

Declare what package the class(es) in this file belong to

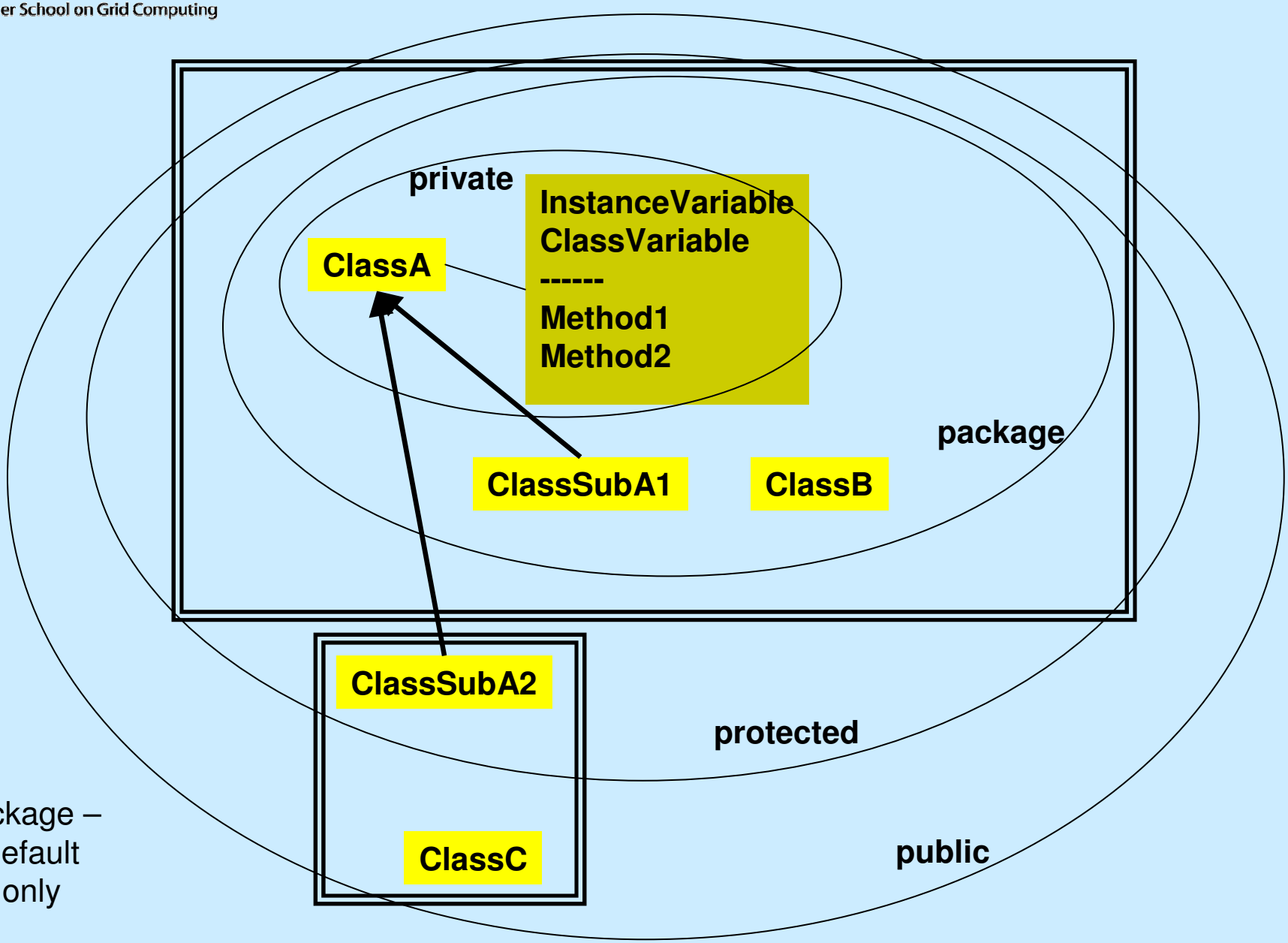
Import specific class from that package

Import all classes from that package

```
package uk.ac.nesc.rph.mypackages.acc;
import uk.ac.nesc.robert.mypackages.acc.YClass;
import uk.ac.nesc.rph.mypackages.anotherpackage.*;
.............
Class …
Class …
```

- In a file
  - First is package name (if any)
  - Next are imports
  - Then one or more classes
  - There may be one public class X for file X.java

Visibility



package – default only

Documentation

- If the components in a package are to be re-used they need documentation – information provided to the programmers who are going to re-use them

  information about the methods etc which are externally accessible.

- Documentation – about what they do and how to use them

Different from

- Commentary - about how they work – for maintenance

- There is a javadoc tool which automatically generates HTML pages of documentation using special comments in the program
- Embedding the documentation in the code means it is more likely to be updated when the code changes

Javadoc comments

- Documentation comments have the form /** <comment> */
- The comment can include @ tags, e.g. @author Richard Hopkins
- These are treated specially in the generated documentation
- The comment immediately precedes the thing it is describing –
  - Class
  - Attribute
  - Constructor
  - Method

```
/** Maintains a value of type double which
  * can be manipulated by the user
  * @author R. Hopkins
 */
public class Accumulator {
double total = 0.0;

/** To increment the accumulator's value
  * @param i the increment
*/
public double incr (double i) {
total = total + i;
return total; } }
```

Java API s

- Java API – Packages which are part of the Java platform

  http://java.sun.com/j2se/1.4.2/docs/api/

- Most useful
  - **java.lang**
  - java.io
  - java.util.*

Java.lang

- Java.lang
  - **Object** – clone() , equals() , toString() , hashCode() , …
  - **Integer** – MAX_VALUE , compareTo() , parseInt() , valueOf() **….**
  - **Double , Byte , Short , Long , Float** – similar
  - **Number**
  - **Boolean** – valueOf(), …
  - **Character** – valueOf(), **….**
  - **Enum**
  - **Math** – E, PI, abs(), sin(), sqrt(), cbrt(), tan(), log(), max(), pow(), random() **…**
  - **Process, ProcessBuilder**
  - **String** – string , getChars , compareToIgnoreCase, **…**
  - **System** – err, in, out, arrayCopy(), currentTimeMillis(), getProperty() , **…**
    - getProperties() documents what they are
  - **Thread** – sleep(), …
  - **Throwable, Exception, Error**

Project Tools

Java Archives - JAR Files

- Bundle mutiple files into a single (compressed) archive file
- As ZIP files – uses same format
- Can have an executable JAR file

Another Neat Tool - Ant …

- is a tool for building projects
  - performs similar functions to `make` as a software project build tool.
- uses a file of instructions, called *build.xml*, to determine how to build a particular project
  - Structurally similar to a `Makefile`
  - Uses XML representation
- is written in Java and is therefore entirely platform independent
  - Can be extended using Java classes

Build File (1)

A Build file defines one (or more) projects

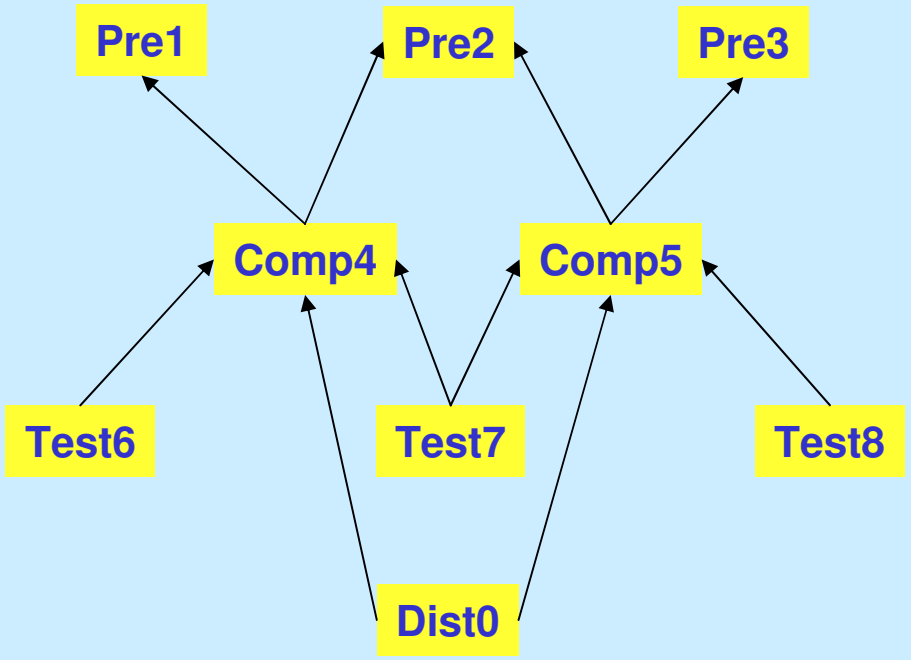
- **Each project defines**
  - **a number of targets**
    - **Each target is an action which achieves the building of something**
      - **Comprises one or more tasks**
  - **Dependencies between targets**

    to achieve target X we must first achieve targets Y, Z, …
  - **Properties – name value pairs,**

    <property name="src" location="MyCalc"/>
    - **so tasks can be parameterised - refer to property name**
    - **property value can be set from within the build fle, or externally as a build parameter**

Project Structure

- Build files gives a DAG (Directed Acyclic Graph) of target dependencies
- E.g PreN – preparation – e.g copy in some files
  CompN – compile some program
  TestN – runs some standard test
  DistN – prepare an archive file for distribution (JAR for Java Archive)



- Everything defined just once
- Do minimum necessary work

e.g. for target test8

ant test8

does Pre2 and Pre3

but not Pre1

won't do Pre2 if its output files are more recent than its input files

e.g. ant Dist0

Pre2 is only run once

Task Definition

- A task is a piece of code that can be executed.
  - A task can have multiple arguments.
    The value of an attribute might contain references to a property.
    These references will be resolved before the task is executed.
  - Tasks have a common structure:

    <name attribute1="value1" attribute2="value2" ... />

    name is the name of the task,
    attribute*N* is the attribute name
    value*N* is the value for this attribute.
  - There is a set of built-in tasks, along with a number of “optional” tasks
  - it is also very easy to define your own.

Example Build File

```
<project name="Assignment" basedir=".">
        <property name="src" location="Assignment/src"/>
        <property name="build" location="Assignment/build"/>
    <target name="init">
        <mkdir dir="${build}"/>
    </target>
    <target name="compile" depends="init" >
        <javac srcdir="${src}" destdir="${build}" />
    </target>
    <target name="dist" depends="compile" >
        <jar jarfile="lib/Assignment.jar" basedir="${build}"/>
    </target>
    <target name="clean" description="clean up">
        <delete dir="${build}"/>
    </target>
</project>
```

Documentation - http://ant.apache.org/manual/index.html

ClassPath (The dreaded)

- The Java compiler and JVM loader need to know what directories to search for the .jav or .class files that it needs
- This is provided by a class path -

  a : separated list of directory names, e.g

  ~/MyProj/MyCalc:/GT4/SRB/src: .....

And Grid Middleware Is complex

This is dreaded because

- In a complex system the class path can be very long
  - Both in number of entries
  - And name for each entry, e.g. /uk/ac/nesc/rph/myProject
- Any jar files used must be explicitly included (you cannot just include a directory containing all relevant jar files)
- If it is wrong – a required file cannot be found – it is very hard to track down the problem

and Grid middleware is comlex

Setting the Class Path

- Directly on the java / javac **command line**
  java –classpath ~/myJava/utilities:~hisJava/oddsAndEnds MyClass 22

  Class path | To run | Arg[0]

- By (re-) setting the $CLASSPATH **environment variable**
  $export CLASSPATH=$CLASSPATH:~/me/extraClasses

- As part of the **build file**

```
<javac srcdir="${src}" destdir="${build}">
    <classpath>
        <pathelement path="${basedir}/lib/Jar1.jar"/>
        <pathelement path="${basedir}/lib/Jar2.jar"/>
        <pathelement path="${basedir}/lib/Jar2.jar"/>
    </classpath>
</javac>
```

- If none is specified a default class path is used that includes the current working directory.

The practical

General

- Introduction to Java
- Classes and Objects
- Inheritance and Interfaces

Detail

- Expressions and Control Structures
- Exception Handling
- Re-usable Components

- **Practical**
- Reference Material

Practical - Directory Structure and commands

- Package name **uk.ac.nesc.rph.myCalc*N***
- Matches directory structure - /uk/ac/nesc/rph/calc

```
rph                    //home – that's me – rph@nesc.ac.uk
  JavaTutorial         // run everything here
        JavaDoc        // .html files
        Data                    // input and output  files
        uk
         ac
           nesc
              rph
                 myCalcN              //package name
                   MyCalculatorN.java      // Step N - source
                   MyCalculatorN.class     // Step N - compiled
```

```
$mkdirhier   uk/ac/nesc/rph/myCalc
$javac   uk/ac/nesc/rph/myCalc/MyCalclulatorN.java     2>MC.err
$javadoc  –d  JavaDoc  uk/ac/nesc/rph/myCalc/MyCalculator*.java
$java  uk/ac/nesc/rph/myCalc/MyCalculatorN  arg0    arg1
```

Practical

- Material is here

http://www.gs.unina.it/~refreshers/java

- Help session – here Monday 12.30 -14.30

Reference

- General
- Introduction to Java
- Classes and Objects
- Inheritance and Interfaces

- Detail
- Expressions and Control Structures
- Exception Handling
- Re-usable Components

- Practical
- **Reference Material**

Basic Syntax - Identifiers

- Identifiers, examples –
  - i
  - engine3
  - the_Current_Time_1
- Identifiers, rules
  - Start with <letter> _ $ £ <.. A currency symbol >
  - Continue with those + <digit>
  - Excluding reserved words
  - No length limitation
- Identifiers, conventions
  - $ etc – for special purposes – do not use
  - HelloWorld – class name,
    - start with u/c, capitalise start of words
  - mainMethod – everything else
    - Start with lower case, capitalise start of words

Basic Syntax – Primitive Types

Type	Size (bits)	Example literals
boolean	1	true false
char	16	'A' '\"' '\\' '\n' '\r' '\u05F0' '\t'
byte	8	integer: -64 123 073 0x4A2F 9223372036854775808L
short	16	
int	32	
long	64	
float	32	floating point: 11.3E-4 73.45 -1.45E+13
double	65	

\n - newline
\r - return
\t - tab
\u -Unicode – 4 hex digits

initial 0 - octal
initial 0x or 0X - hexadecimal
Final L – long

- Default values – 0 (= false)

Basic Operators

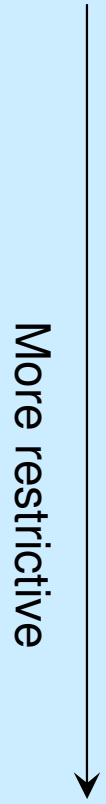
- * / % multiply, divide, remainder
- + – plus, minus
- + – unary plus, unary minus
- + string concatenation
- > >= < <= == != comparison
- ! || && ^ boolean – not, or, and, exclusive or
  (for || and && - conditional evaluation of 2nd argument)

- ++ – – post increment/decrement n++
- ++ – – pre increment/decrement ++n
- + = – = assignment with operation n += 10 (n=n+10)
- ^= b ^= true (b=!b)????

Additional Operators

- ~ integer – bitwise complement
- << integer – left shift
- >> integer – right shift with zero extension
- >>> integer – right shift with sign extension
- & integer – bitwise and
- | integer – bitwise or
- | boolean – unconditionally evaluated Or
- ^ integer – bitwise exclusive or
- *= /= %= <<= >>= >>>= &= ^= |=
  assignment with operation

Visibility – Access Specifiers



- **Public** – Accessible wherever its containing class is – least restrictive.
- **Protected** ---Only accessible to sub-classes and the other classes in the same package.
- **Package access** ---Members declared without using any modifier have package access. Classes in the same package can access each other's package-access members.
- **Private** – only accessible from within the containing class itself – most restrictive

Thanking our sponsors…

Thanking our sponsors…

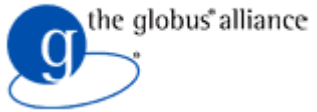
The Grid World moves
Fast as magic

To Know what's happening
STAY ALERT

Micron AMD Allied Telesyn

Microsoft IBM hp invent HITACHI Inspire the Next the globus alliance

Thanking our sponsors…




Java