Programming 2

Lecture 1: Java Revision

1. Basics

Compilation, *types, operators, control structures,* ***Classes and Methods,*** *references, static methods, flow of control*

2. Arrays: *primitives, objects and multidimensional*

3. Inheritance: *basic syntax, programming features, using inheritance*

See package Programming2Revision for examples

See Programming 1 lecture notes and Lewis and Loftus

Programming Languages

http://lang-index.sourceforge.net/

http://spectrum.ieee.org/computing/software/the-2015-top-ten-programming-languages

Rank	Name	Share	Last month's share	Last year's share
1	C	17.668%	15.868%	16.825%
2	Java	14.720%	15.450%	20.381%
3	Objective-C	8.230%	8.516%	9.221%
4	C++	6.770%	7.544%	7.912%
5	Basic	5.457%	5.955%	7.592%
6	PHP	4.401%	4.144%	4.247%
7	Python	3.658%	3.363%	3.616%
8	C#	3.269%	3.444%	4.598%
9	Perl	2.566%	2.455%	2.459%
10	Ruby	1.918%	1.392%	1.576%

Programming Language Categories

http://en.wikipedia.org/wiki/List_of_programming_languages_by_type

1. **Low level:** talk directly to the processor (assembly or machine languages: e.g. GAS the GNU assembler)
2. **High level:**
    1. Procedural (C, Fortran, Pascal, VB)
    2. Object Oriented (Java, C++)
    3. Scripting (PHP, javascript)
    4. Declarative/Functional (Prolog,Lisp, F#, Haskell)

    + many more (e.g. Off-side rule languages like Python**)**

Programming Language Differences

http://en.wikipedia.org/wiki/List_of_programming_languages_by_type

1. Compilation process
2. Basic syntax
3. Basic structure (e.g. typing rules)
4. Execution process
5. Data structures
6. Software engineering features

Compilation

Interpreted languages

Source code is directly run line by line via an interpreter (e.g. javascript, PHP)

Hybrids

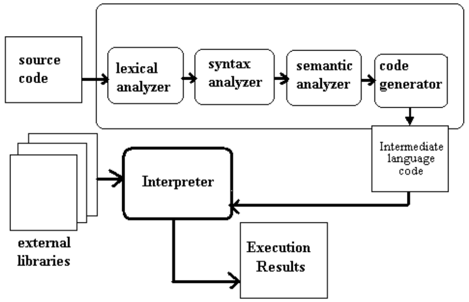
Source code compiled into an intermediary language, which is then interpreted (e.g. Java, C#)

Compiled languages

A compiler creates an executable program from the source

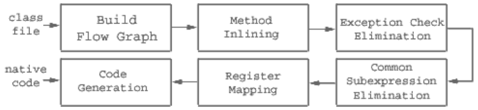
(e.g. C, C++)

Java Process: Hybrid Compiler/Interpreter



Java compiles source into bytecode. The unit of compilation is a class and the bytecode is stored in a .class file. Bytecode is portable. When a class is executed, bytecode is compiled and executed by the Java Runtime Environment (JRE). The main method of an application can be changed without recompiling the bytecode, since everything is compiled on the fly.

The JRE uses Just In Time compilation



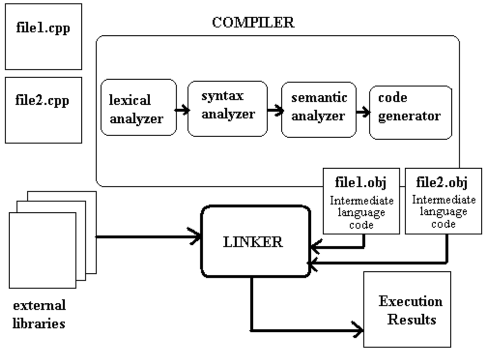
The JRE is everywhere. There are not many compilers for making bytecode from other languages

Lewis and Loftus Chapter 1

http://www.codeproject.com/KB/dotnet/clr.aspx

C/C++ Process

C++ compiles source code into object files then links into machine dependent executable (via assembly language)



http://www.codeproject.com/KB/dotnet/clr.aspx

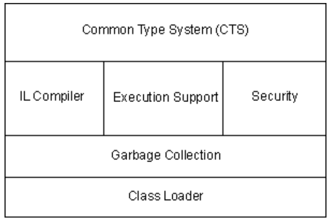
C# Process: Two stage compiler

C# compiles source code into IL (intermediate language) equivalent to bytecode, but the unit of compilation is an assembly (DLL file)

This means that if you change the main class in C#, you have to recompile the whole project. The .net environment converts the assembly into executables

The CLR is an environment in which .NET applications that have been compiled to IL can be run.

It does **not** interpret. It forms an executable. Not sure where the JIT compiler comes into it then!



Common Type System (CTS)		
IL Compiler	Execution Support	Security
Garbage Collection		
Class Loader		

Common Language Runtime (CLR)

.net generally on windows only. Languages: VC, VB.net, ASP.net, C# + third party

http://homepages.com.pk/kashman/dotnet.htm#_dotnet_operating_system

Java Basic Syntax Review

See Programming Semester 1 Lecture 2-4

Lewis and Loftus Chapters 2, 5 and 6

Primitive Types

Type	**Default**	**Space (bits)**	**Range**
`boolean`	false	1	[true, false]
`byte`	0	8	[-128, 127]
`char`	\u0000	16	[\u000, \uFFFF]
`double`	0.0	64	10^{-324} to 10^{308}
`float`	0.0	32	10^{-46} to 10^{38}
`int`	0	32	[-2147483648, 2147483647]
`long`	0	64	-2^{63} to $2^{63}-1$
`short`	0	16	[-32768, 32767]

```
int count;
boolean  test;
double sum;
```

Notes on primitive types

1. In Java, primitive variables are only accessible by the variable name, not by a pointer/reference. **C++ has primitive variable pointers**.
2. Java is strongly typed, which means variables must first be declared before they can be used (**unlike Matlab**).
3. Java performs type checks. This means that if you make assignments such as those below, the code will not compile

```
int a=10;
boolean  b;
double x,y=1,z;
```

`x=a;` ← int to double is allowed

`a=y;` ← Not allowed, need to **cast** `a=(int)y;`

`z=b;` ← Not allowed even with cast

C++ does not perform these checks.

4. Java initialises all primitives to zero. However, the compiler will not let you assume a variable has been initialised

`y=z;` ← Error, z may not have been initialised

Operators

These are all the same in C, C++ and C#

Assignment =

Arithmetic Operators `+ - * / %`

Logical Operators `== && || !`

Compound Operators `++ -- += /= *= -=`

Bit Operators & `|` ~ ^ << >> >>>

http://java.sun.com/docs/books/tutorial/java/nutsandbolts/op3.html http://docs.oracle.com/javase/tutorial/java/nutsandbolts/operators.html

Operator Precedence

Operators	Precedence
postfix	*expr*++ *expr*--
unary	++*expr* --*expr* +*expr* -*expr* ~ !
multiplicative	* / %
additive	+ -
shift	<< >> >>>
relational	< > <= >= instanceof
equality	== !=
bitwise AND	&
bitwise exclusive OR	^
bitwise inclusive OR	|
logical AND	&&
logical OR	||
ternary	? :
assignment	= += -= *= /= %= &= ^= |= <<= >>= >>>=

Notes on Operators

1 Because of the type checking, if you write something like

```
if(a=10){ }
```

instead of

```
if(a==10){ }
```

The compiler will detect the error. **C/C++ does not do this check**

2. Operator precedence: arithmetic ahead of logical.

`if(a==7+3){}` tests whether a equals 10

If in doubt, use brackets

```
if(a==(7+3)){}
```

3. `++ --` Compound Operators can either be pre or post

```
int a=10; b, c=10;
b=a++;
b=++c;
```

Post increment. a has the value 11, b has the value 10

Pre-increment. c has the value 11, b has the value 11

Bit Operators

http://docs.oracle.com/javase/tutorial/java/nutsandbolts/op3.html

`&   |   ~   ^   <<   >>   >>>`

operators that perform bitwise and bit shift operations on integral types

```
byte a=0,b=42,c=10, d=127;
```

Classic interview question: write an algorithm to determine whether a number is a power of 2 or not

See 1M0Y Semester 1 Lecture 3-4

Conditional Control Structures

`if` statements

```
if(count==1)
     minValue=x;
else
     minValue =y;
```

`switch` statements

`?` the conditional operator

```
minVal=x<y? x:y;
```

Conditional Control Structure Examples

```
if(count==1)
      minValue=x;
else
      minValue =y;
```

`switch` statement

`?` the conditional operator

Notes on Conditional Control Structures

•Logical expression for an **if** statement must evaluate to a boolean (different in C/C++, booleans are just integers)

•Prior to Java 7, switch statement expression must be an int or char (same as C/C++). Post Java 7, **String switches are now allowed.**

•Switch will fall through if you don’t include breaks

Repetition Control Structures

`for` statements

`while` statements

```
int count =0;
while(count<20)
{
      //Do stuff
      count++;
}
```

`do … while` statements

`for … each` statements

Notes on Repetition Control Structures

- you can `break` out of a loop, but it is ugly.
- you can `continue` in a loop to end the current iteration, but it is confusing
- `for … each` loops work with `Iterable` collections (we revisit later) but cannot be used to alter the contents of the array you are iterating across

Common interview question: write a method to find kth smallest element in unsorted array.

Classes and Objects

References

Methods and static methods

Flow of control

Lewis and Loftus

Classes and Objects

•A non-primitive data type is defined using a class. A class is a **template**

```
public class Student{
     String name;
     int score;
}
```

•An **object** is an **instance** of a **class**

```
bob = new Student();
```

References

•**To create a student object we declare a student *reference***

```
Student bob, alice;
```

•Then call new to allocate new memory for a student object

```
alice = new Student("ALICE",88);
bob = new Student("BOB",55);
```

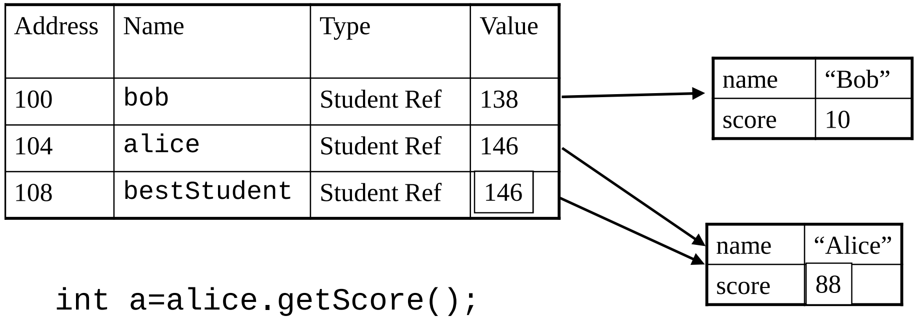
How references work

•**References are variables that can store the location in memory of an object.**

```
Student bob;
bob= new Student(“Bob”,10);
```

Address	Name	Type	Value
0x00000	bob	Student Reference	0x00004
0x00004		Student	name “Bob” score 10

```
Student bob=new Student(“Bob”,10);
Student alice=new Student(“Alice”,99);
Student bestStudent= alice;
bestStudent.setScore(88);
```



Address	Name	Type	Value
100	bob	Student Ref	138
104	alice	Student Ref	146
108	bestStudent	Student Ref	146

name	“Bob”
score	10

name	“Alice”
score	88

```
int a=alice.getScore();
```

Returns 88 NOT 99

Methods and Classes

- A class definition consists of:
  - the instance variables for an object of that class; and
  - the methods that can performed on an object.

```
public class Student{
     protected String name;
     protected int score;

     public int getScore(){ return score;}
     public void setName(String s)
     {
          name=s;
     }
}
```

Static Variables and Methods

- Static variables and methods are associated with the **class** rather than the **object**

```
public class Student{
//Static variable
     private static String university=“UEA”;

//Static method
     public static void setUni(String s)
     {
          university=s;
     }
}
```

Static vs Dynamic

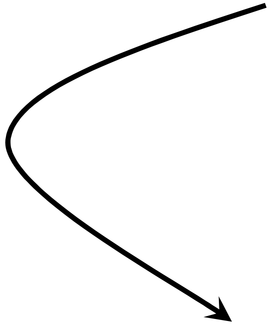
```
public class Student{
   public static int numStudents=0;
   private String name;
   private int score;
   public Student(){
         name="";
         score=0;
   }
   public Student(String n, int s){
         n;
         score=s;
   }
   public void setName(String s){name =s;}
   public static void setNumStudent(int a){numStudents=a;}
   public Student higherScore (Student a){
         if(score>a.score)
               return this;
         return a;
   }
}
```

Test Class

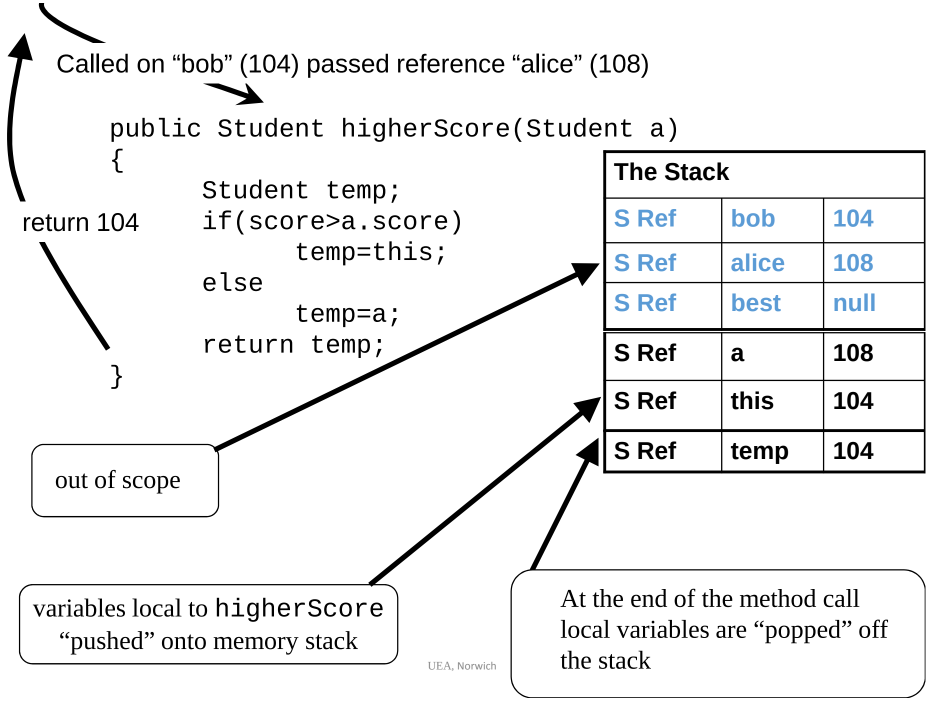
```
public class Student {
//Other stuff here

   public static void main(String[] args)
//This is a test harness
   {
// Access the static variable nosStudents.
         Student.numStudents=2;
//Call static method
         Student.setNumStudent (20000);

         Student bob, alice, bestStudent;

         bob=new Student(“Bob”,65);
         alice=new Student(“Alice”,78);
         best=bob.higherScore(alice);

   }
}
```

Flow of Control

```
public class StudentTest{
public static void main(String[] args)
{
      Student bob, alice, best;
      bob=new Student(“Bob”,65);
      alice=new Student(“Alice”,78);
      best=bob.higherScore(alice);

  }
}
```



The Stack		
Student Ref	bob	104
Student Ref	alice	108
Student Ref	best	null



Called on “bob” (104) passed reference “alice” (108)

```
public Student higherScore(Student a)
{
      Student temp;
      if(score>a.score)
            temp=this;
      else
            temp=a;
      return temp;
}
```

return 104

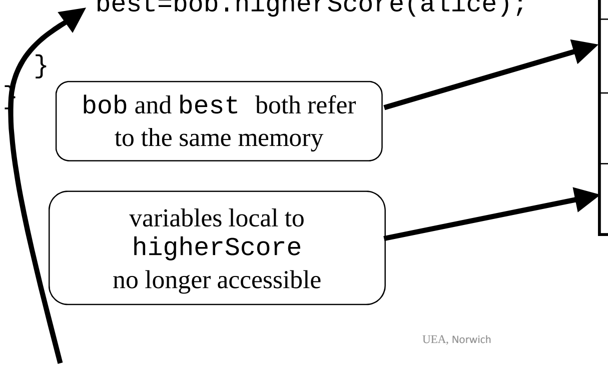
The Stack		
S Ref	bob	104
S Ref	alice	108
S Ref	best	null
S Ref	a	108
S Ref	this	104
S Ref	temp	104

out of scope

variables local to higherScore “pushed” onto memory stack

At the end of the method call local variables are “popped” off the stack

```
public class StudentTest{
public static void main(String[] args)
{
      Student bob, alice, best;

      bob=new Student(“Bob”,88);
      alice=new Student(“Alice”,78);
      best=bob.higherScore(alice);

  }
}
```



The Stack		
Student Ref	bob	104
Student Ref	alice	108
Student Ref	best	104

bob and best both refer to the same memory

variables local to higherScore no longer accessible

Basics with Java vs C++

•In Java primitives are only accessible directly through the variable name. You cannot pass a method a pointer to a primitive. **C++ has references to primitives (pointers)**

•In Java objects are always accessed by reference. You cannot pass a whole object to a method.

•In Java we have **static methods** and **dynamic methods**. In C++, these are called **functions** and **member functions**

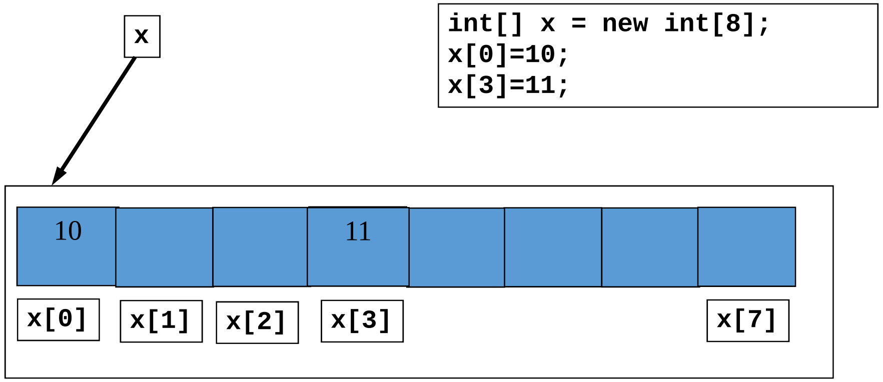
ARRAYS

1. Arrays of Primitives:
2. Arrays of References:
3. **Multi-Dimensional Arrays**
4. ArrayList

1. Arrays of Primitives

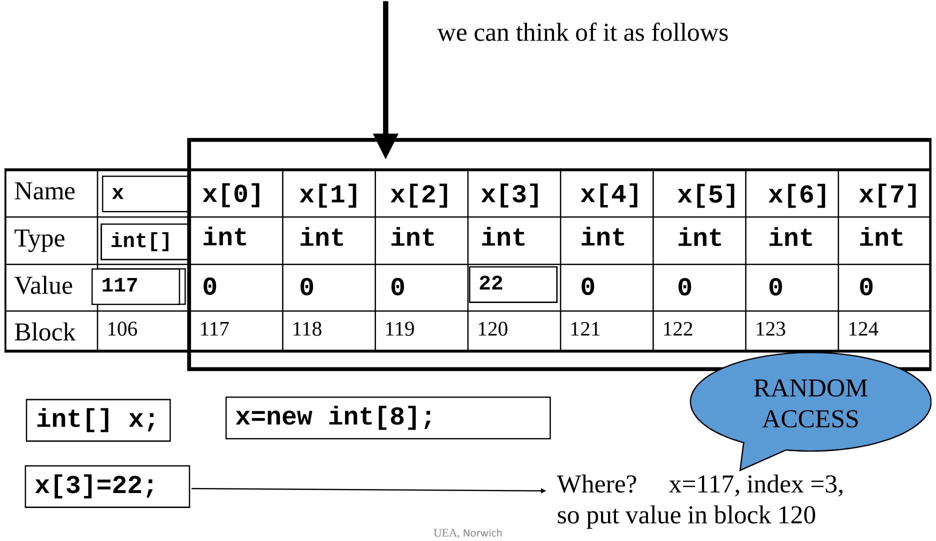
Arrays are collections of elements contiguous in memory that implement a list with random access

```
int[] x = new int[8];
x[0]=10;
x[3]=11;
```



Hard Disk

Value	00011111	00011000	00000111	10101011	10011101	00000000	00010101	11101111	10101010	00101111
Block	106	107	108	109	110	111	112	113	114	115



we can think of it as follows

Name	x	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
Type	int[]	int	int	int	int	int	int	int	int
Value	117	0	0	0	22	0	0	0	0
Block	106	117	118	119	120	121	122	123	124

```
int[] x;
```

```
x=new int[8];
```

```
x[3]=22;
```

Where? x=117, index =3, so put value in block 120

Name	x	x[0]	x[1]	x[2]	x[3]	x[4]	x[5]	x[6]	x[7]
Type	int[]	int	int	int	int	int	int	int	int
Value	107	5	34	63	92	111	140	169	208
Block	106	107	108	109	110	111	112	113	114

We use a loop to access all the elements of an array

```
boolean[] pass =
new boolean[size];
char[] name = new char[size];
for(int i=0;i<x.length;i++)
{
      pass[i]=false;
      name[i]='A';
      x[i]=i*30+(5-i);
}
```

i=0
x[0]=0*30+5-i=5;

i=1
x[1]=1*30+5-1=34;

i=7
x[7]=7*30+5-7=208;

2. Arrays of Objects

```
Object[] objectArrayReference;
House[]   pottergate;
```

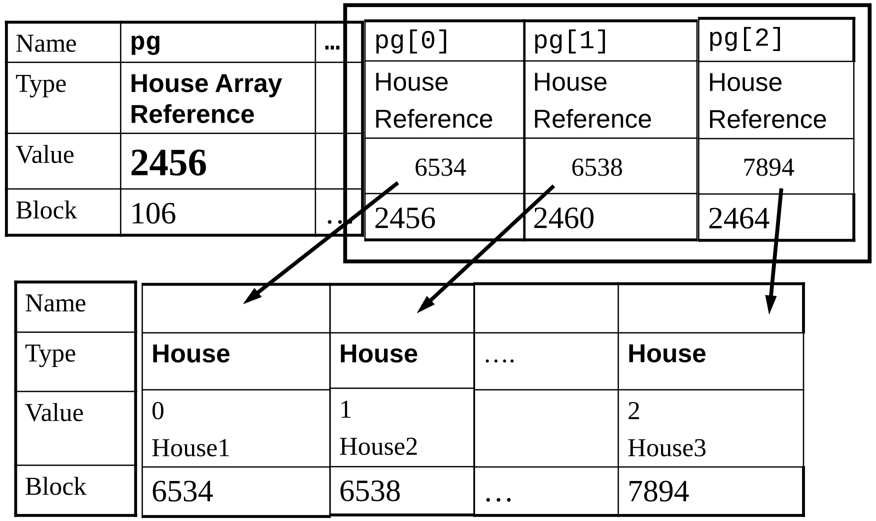
Declares a reference to an array of Object references and a a reference to an array of House reference

Name	pottergate			
Type	House Array Reference			
Value	null			
Block	106	107	108	109

```
pg=new House[3];
```

Name	pg	…	pg[0]	pg[1]	pg[2]
Type	House Array Reference		House Reference	House Reference	House Reference
Value	2456		NULL	NULL	NULL
Block	106	…	2456	2460	2464

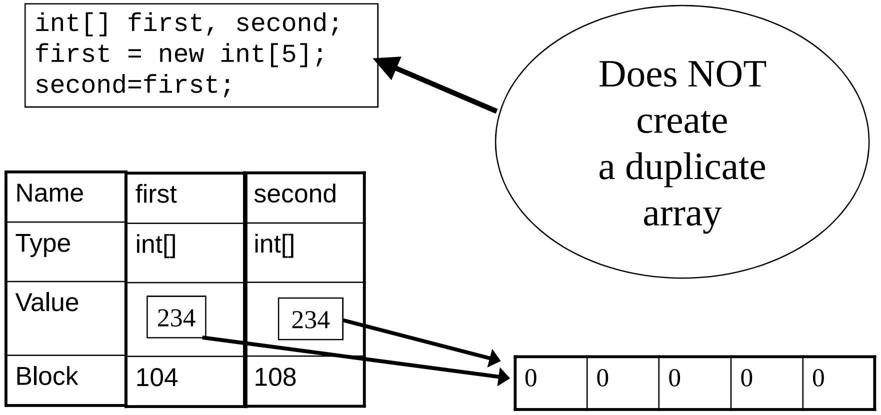
```
for(i=0;i<pg.length;i++)
     pg[i]=new House(i,”House”+(i+1));
```



Name	pg	…	pg[0]	pg[1]	pg[2]
Type	House Array Reference		House Reference	House Reference	House Reference
Value	2456		6534	6538	7894
Block	106	…	2456	2460	2464

Name				
Type	House	House	….	House
Value	0 House1	1 House2		2 House3
Block	6534	6538	…	7894

3. Copying Arrays and Array References

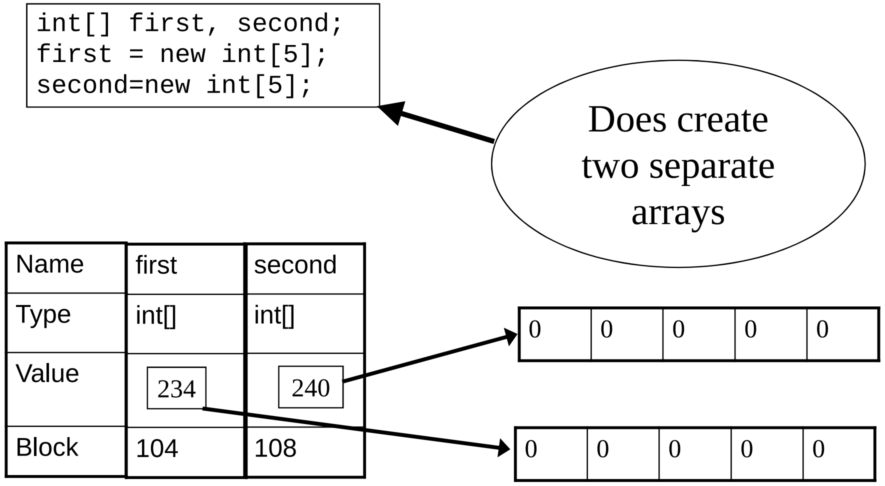


```
int[] first, second;
first = new int[5];
second=first;
```

Does NOT create a duplicate array

Name	first	second
Type	int[]	int[]
Value	234	234
Block	104	108

Copying Arrays and Array References



```
int[] first, second;
first = new int[5];
second=new int[5];
```

Does create two separate arrays

Name	first	second
Type	int[]	int[]
Value	234	240
Block	104	108

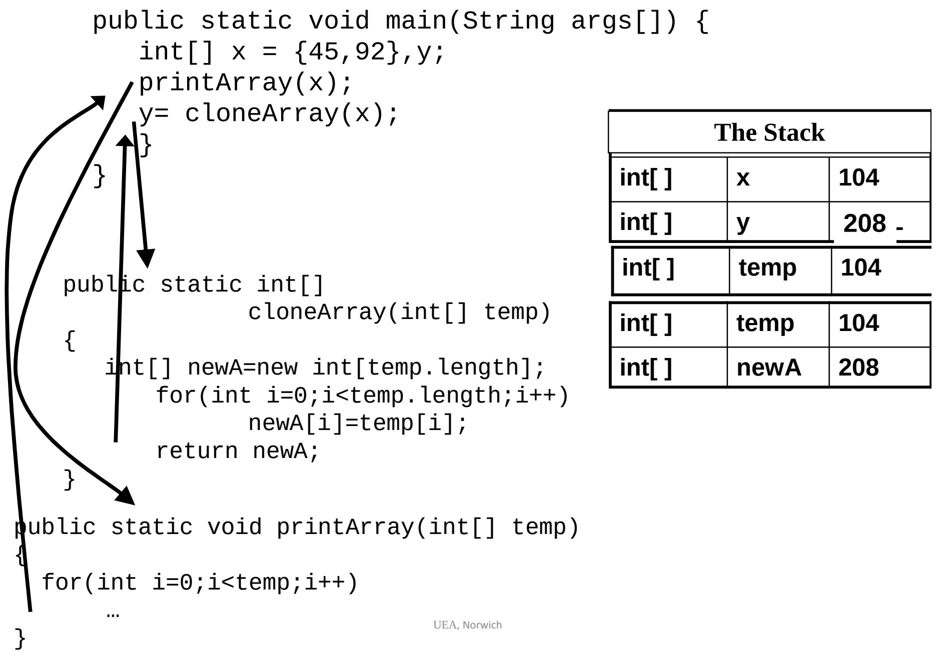
Passing Arrays to Methods

```
public class ArrayIllustration {
public static void printArray(int[] temp)
{
  for(int i=0;i<temp;i++)
         System.out.println("Element"+i+" = "+temp[i]);
}
public static int[] cloneArray(int[] temp)
{
   int[] newA=new int[temp.length];
   for(int i=0;i<temp.length;i++)
        newA[i]=temp[i];
   return newA;
}
```

Array reference argument

Returns an array reference

Could use `System.arraycopy`
arraycopy(Object src, int srcPos, Object dest, int destPos, int length)



```
public static void main(String args[]) {
   int[] x = {45,92},y;
   printArray(x);
   y= cloneArray(x);
   }
}
```

```
public static int[]
            cloneArray(int[] temp)
{
   int[] newA=new int[temp.length];
      for(int i=0;i<temp.length;i++)
            newA[i]=temp[i];
      return newA;
}
```

```
public static void printArray(int[] temp)
{
  for(int i=0;i<temp;i++)
      …
}
```

The Stack

int[ ]	x	104
int[ ]	y	208 -
int[ ]	temp	104
int[ ]	temp	104
int[ ]	newA	208

Multi Dimensional Arrays

2-Dimensional Arrays

- The most natural way to model tabular data is by using an array

Data: Number of students in CMP

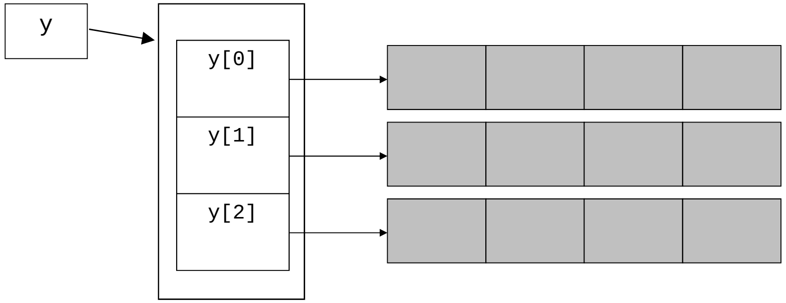
	Year		
Course	1	2	3
G500	80	75	70
G505	10	9	11
G600	20	18	21

Note the shortcut initialisation. This can only be done at declaration

```
2-D Array
int[][] studentCount = { {80,75,70},
                         {10,9,11},
                         {20,18,21} };
```

```
Object[][] y = new Object[3][4];
```

y is a reference to an array of object arrays



- Rows do not have to be contiguous in memory
- Note this is fundamentally different to the way C/C++ stores multidimensional arrays

Multi-D Arrays

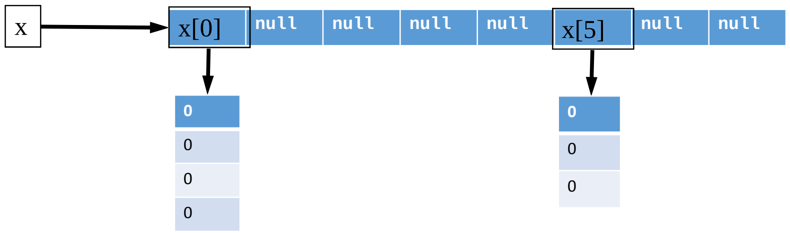
Java allows ragged arrays. A 2-D array is actually an array of array references

```
int[][] x = new int[7][];
```

```
x[0] = new int[4];
```

```
x[5] = new int[3];
```

Array of int[] references



This is NOT possible in C++
where everything is really just a 1-D array

- Tabular data of objects can also be modelled with ArrayList and with ArrayList of ArrayList

```
Object[][] y = new Object[3][3];
```

```
ArrayList<Object[]> ar= new ArrayList<Object[]>(3);
for(int i=0;i<3;i++)
     ar.add(new Object[3]);
```

```
ArrayList<ArrayList<Object>> ar2=
            new ArrayList<ArrayList<Object>>();
for(int i=0;i<3;i++)
     ar2.add(new ArrayList<Object>(3));
```

Higher Dimensional Arrays

```
3-D Array
Pixel[][][] mriScan = new Pixel[654][654][654];
```

```
ArrayList<ArrayList<ArrayList<Number>>> mri=
 new ArrayList<ArrayList<ArrayList<Number>>>();
```

Bit of a hideous nightmare!

```
11-D Array
Pixel[][][][][][][][][][][] superString;
```

that’s just silly

Array Points to Note

1. All Java arrays are Objects. **`Object o1=new int[10];`**

2. All Java arrays contain an extra integer `length.` This is not true of C.

3. Arrays are an implementation of List that allows for constant time access but requires linear time modification (see data structures).

3. `ArrayList` is part of the Collections package and is a java wrapper for Arrays.

4. In C++, array names are simply pointers to the first element, and you can do strange pointer arithmetic operations. **This is not the case in Java!**

5. This alternative declaration is valid

```
     int x[] ;
     double y[];
```

But is not recommended (old style C declarations).

6. Shorthand initialisation is only valid at declaration

```
int[] x={1,2,5,6,11};
int[] y;
y={1,2,5,6,11};
```

OK

INCORRECT

`ArrayList` Class

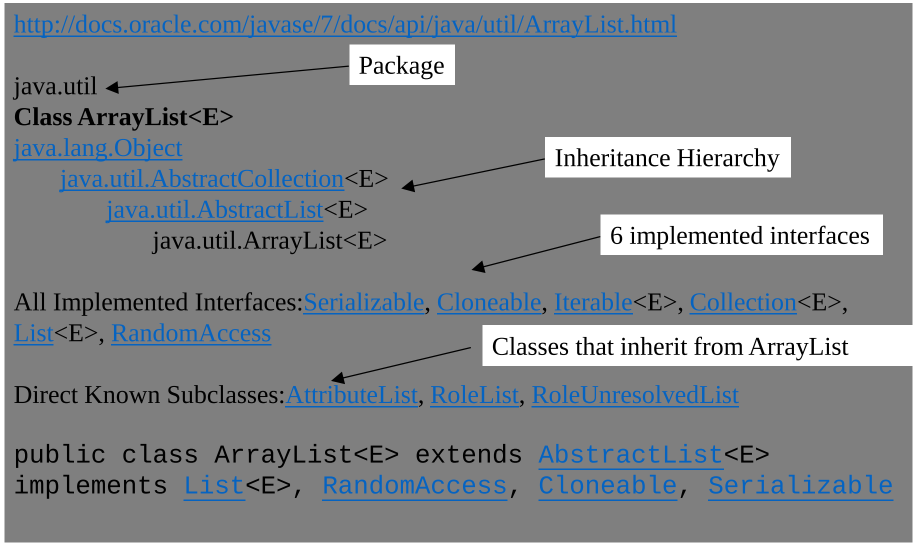
The `ArrayList` class is part of the java.util package of the Java standard class library and part of the Collections framework package, which we cover in detail in week 3.

It implements a List as a maximum size array which grows a maximum of 12 items if the size is exceeded.

```
public class ArrayList<E> extends AbstractList<E> implements List<E>,Cloneable,
Serializable, RandomAccess {
private static final long serialVersionUID = 8683452581122892189L;
private transient int firstIndex;
private transient int lastIndex;
private transient E[] array;
```

ArrayList generic (covered in detail later).

`ArrayList` API



http://docs.oracle.com/javase/7/docs/api/java/util/ArrayList.html

java.util

Class ArrayList<E>

java.lang.Object
    java.util.AbstractCollection<E>
        java.util.AbstractList<E>
            java.util.ArrayList<E>

All Implemented Interfaces:Serializable, Cloneable, Iterable<E>, Collection<E>, List<E>, RandomAccess

Direct Known Subclasses:AttributeList, RoleList, RoleUnresolvedList

```
public class ArrayList<E> extends AbstractList<E>
implements List<E>, RandomAccess, Cloneable, Serializable
```

`ArrayList` Methods

Acccessor Methods

get(int index): Returns the element at the specified position in this list

set (int index, E element). Replaces the element at the specified position in this list with the specified element

contains(Object o): Returns true if this list contains the specified element. More formally, returns true if and only if this list contains at least one element e such that `(o==null ? e==null : o.equals(e))`

Structural Modifier Methods

add(E element): Appends the specified element to the end of this list.

add(int index, E element): Inserts the specified element at the specified position in this list.

remove(int index): Removes the element at the specified position in this list.

remove(Object o): Removes the first occurrence of the specified element from this list, if it is present.

ensureCapacity(int minCapacity): Increases the capacity of this ArrayList instance, if necessary, to ensure that it can hold at least the number of elements specified by the minimum capacity argument.

Using `ArrayList`

Using RAW TYPES for examples. Generics later

```
import java.util.ArrayList;
…
ArrayList test;
test = new ArrayList(10);
Player topScorer = new Player(“Walcott",14);
String champions = "Arsenal";
test.add(topScorer);
test.add(champions);
String str1 = test.get(0).toString();
String str2 = (String)test.get(1);
test.set(1,"Football");
test.set(2,"Club");


test.ensureCapacity(100);
Object b=test.set(11,“Ozil"); //runtime error
Object b=test.get(11);
//this should surely return null but instead crashes!
```

allows access to the class

creates a 10 max size array

Objects in an ArrayList cannot be accessed with an index and square brackets. The method get is used instead

Capacity is different to size!

Inheritance

1. Basic Syntax:
   - 1.1 Terminology
   - 1.2 Inheritance vs Composition
   - 1.3 Final Classes
   - 1.4 Access modifiers
   - 1.5 Constructors
   - 1.6 Static variables and methods
   - 1.7 Pros and Cons
2. Programming Features:
   - 2.1 Overriding Methods
   - 2.2 Polymorphism and Type Compatibility
   - 2.3 Dynamic Binding and Polymorphism
   - 2.4 Abstract Methods and Classes
   - 2.5 Interfaces
3. Using Inheritance:
   - 3.1 Code reuse
   - 3.2 As an alternative to selection
   - 3.3 To decouple an application from the implementation
   - 3.4 To write general methods

Inheritance Syntax

```
public class BaseClass{
//fields and methods here
}

public class SubClass extends BaseClass {
//subclass specialisation
}
```

Any object of type `SubClass` will automatically have the fields and methods of `BaseClass`

```
/*file bankAccount.java **/
public class BankAccount{
      protected double balance;
      public void setBalance(double b){   balance=b;}
}

/*file currentAccount.java */

public class CurrentAccount extends BankAccount{
      public double overdraft;
      public void setOD(double b){overdraft=b;}

      public static void main(String[] args){
             BankAccount b= new BankAccount();
             CurrentAccount c=new CurrentAccount ();

            c.setBalance(100.00);
            c.setOD(1000);
////        b.setOD(99);
}
```

this is ok because c **IS A** `BankAccount` hence c has a field called `balance`, and can call the method `setBalance`

cannot do this because not all bank accounts are current

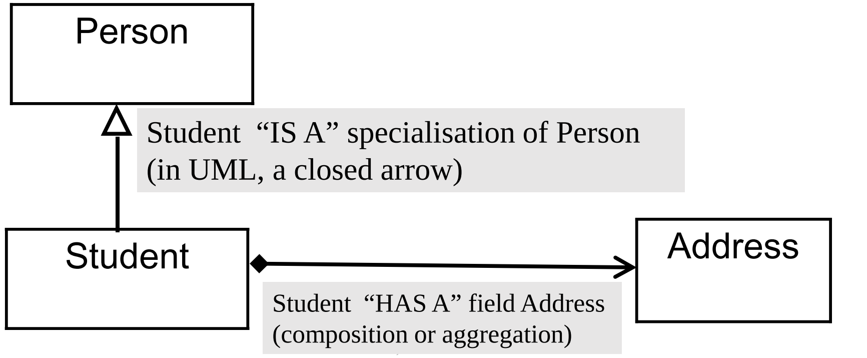
Inheritance Terminology

- A class used to derive a new class is referred to as a *base class*, *parent class* or *superclass.*
- A new class inheriting from a base class is referred to as a *derived class*, *child class* or the *subclass*
- A subclass *extends* a base class
- The relationship between the subclass and the base class is that the subclass "IS-A" particular example of a type of the base class

Inheritance vs Composition

- Inheritance is a different relationship to composition.

INHERITANCE: Class B “IS A” specialisation of Class A
COMPOSITION: Class B “HAS A” field of Class A



Student “IS A” specialisation of Person (in UML, a closed arrow)

Student “HAS A” field Address (composition or aggregation)

Final Classes

- You can make it so a class cannot be extended by declaring the class `final`

```
public final class FinalClass{
//You cannot now extend this class
}
```

- Making classes final can improve the efficiency of your code

Access Modifiers

- All public and protected fields and methods are inherited by the subclass
- Base class private fields are not accessible in the sub class. Objects of the subclass still contain these variables though.

```
public class Shape {
    private double area;
    protected void setArea(double a){area=a;}

 public class Rectangle extends Shape{
       protected double height;
       protected double width;
       public void findArea(){
 //          area= height*width;
             setArea(height*width);
 }
```

Cannot access the private field directly

OK

Constructors

- **Constructors are NOT inherited in Java (can be in C++)**
- this can cause you problems if you do not define a default constructor, and if you call other methods in the constructor.

```
public class Shape {
    private double area;
    public Shape(double d){area=d;}
```

```
public class Rectangle extends Shape{
    double height;
    double width;
public Rectangle(double h, double w){

    super(h*w);

    height=h;
    width=w;
}
```

Will not compile, because there is no call to a base class constructor

Either insert default constructor in `Shape` or call a `Shape` constructor with **`super`**

Static Variables and Methods

- Static variables and methods are **not** inherited

```
Public class BankAccount{
      public static double MAXBALANCE=100000000.
      public static void setMaxBalance(double d){
            MAXBALANCE=d;
      }
}
Public class CurrentAccount extends BankAccount{
…
}
```

- There is only ONE double called `BankAccount.MAXBALANCE.`

* You can access static variables and methods from object references, but you shouldn't .

Inheritance Pros

1. It allows a high degree of code reuse
2. It encourages encapsulation and data hiding
3. It allows stable modularity by removing the need to rewrite code

Inheritance Cons

1. It can encourage you to write over complex, over designed code
2. It can encourage sloppy use of data structures because of implementation hiding
3. It introduces overheads into your programs performance

2. Further Inheritance Programming Points

2.1 Overriding Methods
2.2 Polymorphism and Type Compatibility
2.3 Dynamic Binding and Polymorphism
2.4 Abstract Methods and Classes
2.5 Interfaces

1. Method Over-riding

Sub class can change the definition of a method in the base class

base class

```
public class Shape{
     public double area()
     {
          return -1;
     }
     public double perimeter()
     {
          return -1;
     }
}
```

“dummy” methods

```
public class Rectangle extends Shape{
      private double length=10;
      private double width=10;
      @Override
      public double area()
      {
            return length*width;
      }

      public static void main(String[] args)
      {
            Shape s =new Shape();
            Rectangle r=new Rectangle();
            double a1=s.area();
            double a2=r.area();
            double p1=s.perimeter();
            double p2=r.perimeter();
      }
}
```

sub class

Optional javadoc

method area has been over-ridden

returns -1

returns 100

Both return -1 from a call to the Shape method, since perimeter has not been overriden

Parent methods can be called using `super`

```
public class Person{
      protected String name;
      public String toString(){
            return name;
      }
}

public class Student extends Person{
      protected String studentID;
      public String toString()
      {
            String temp=super.toString();
            return temp+","+studentID;
      }
}
```

Over-ridden method

call to `Person.toString()`

This is the way to enhance a method, rather than redefine it completely

`final` Methods

- All methods by default can be overridden.
- If you declare a method **`final`** then it cannot be overridden

```
public class BankAccount{
      protected double balance;
      public final double setBalance(double b){
            balance=b;
      }
}
```

- Making a method final improves the efficiency of your code
- In C++ (and C#), all methods (member functions) are by default final. To make it so you can override a method, you declare it `virtual`

2.1 Overriding Methods
2.2 Polymorphism and Type Compatibility
2.3 Dynamic Binding and Polymorphism
2.4 Abstract Methods and Classes
2.5 Interfaces

Polymorphism: The ability to store references to different types of objects

```
Student s;
Postgrad bob = new Postgrad();
Undergrad alice=new Undergrad();

s=bob;
s=alice;
Student s = new Student();
```

Postgrad and Undergrad are sub classes of Student

`Student` references can thus store the location of `Postgrad` and `Undergrad` objects

~~`bob=s;`~~
~~`bob=alice;`~~

`Postgrad` CANNOT store `Student`

Polymorphism

- •A reference can store the memory location of any subtype of that class
- •All classes inherit from the built in class `Object`
- •Hence an `Object` reference can store the location of an instance of any class.

```
Object anyObject;
int[] ar=new int[10];
Integer i=new Integer(10);
String s=“Arsenal”;
anyObject=i;
anyObject=s;
anyObject=ar;
```

•C++ does not have this built in inheritance hierarchy. This is one of the main differences in the languages

2.1 Overriding Methods
2.2 Polymorphism and Type Compatibility
2.3 Dynamic Binding and Polymorphism
Chapter 9 Lewis and Loftus
2.4 Abstract Methods and Classes
2.5 Interfaces

Dynamic Binding and Polymorphism

```
Shape s;
s = new Rectangle();
double r = s.area();
```

`s` is static type `Shape`, so does this call `Shape`?

NO: `s` has **DYNAMIC TYPE** `Rectangle`, so it calls this

```
public class Shape{
        public double area()
        {
                return -1;
        }
}
```

```
public class Rectangle extends
Shape{

        public double area()
        {
           return length*width;
        }
```

Dynamic Binding and Polymorphism

- The *static* type of s is `Shape`, but its *dynamic* type is `Rectangle`
- The dynamic type is used to determine which method to call.
- The ability of a reference variable to store references to several different types is called **polymorphism**
- When operations are applied to a polymorphic variable, the operation appropriate to the dynamic type is selected by **dynamic binding** or **late binding**
- This is **the** key feature of OO programming

Note: the static type determines the static variables and methods

2.1 Overriding Methods
2.2 Polymorphism and Type Compatibility
2.3 Dynamic Binding and Polymorphism
2.4 Abstract Methods and Classes
2.5 Interfaces

Abstract classes and methods

`Shape` is now an **abstract class**. This means a `Shape` object cannot be created

```
public abstract class Shape{
     public abstract double area();
}
```

`area` is an **abstract method**. This means that any class extending `Shape` **must define a method `area`**

In C++ pure virtual functions are equivalent to abstract methods

Abstract classes and methods

```
public class Rectangle extends Shape{
     private double length;
     private double width;
     public double area()
     {
          return length*width;
     }
```

`Rectangle` extends `Shape` hence it **must define a method `area`**

In Java, any class with an abstract method must be explicitly declared abstract (not necessary in C++)

2.1 Overriding Methods
2.2 Polymorphism and Type Compatibility
2.3 Dynamic Binding and Polymorphism
2.4 Abstract Methods and Classes
2.5 Interfaces

Interfaces

- In Java, an interface is a collection of abstract methods
- Interfaces allow classes to inherit multiple sets of abstract methods

`interface` not `class`

```
public interface Shape{
     double area();
     double perimeter();
}
```

Abstract methods, but you don’t need the key word `abstract`

Interfaces

`implements` not `extends`

```
public class Rectangle implements Shape{
      private double length;
      private double width;
      public double area()
      {
            return length*width;
      }
```

No equivalent in C++

classes can only inherit from a single class

classes can implement any number of interfaces

Java built in interfaces

1.Comparable Interface

```
public interface Comparable {
     public int compareTo(Object o);
}
```

- Returns 1 if the calling object is “greater than” the one passed as an argument
- Returns -1 if this is “less than” the argument
- Returns 0 if they are equal

Because `Student` implements `Comparable` it must define `compareTo`

```
public class Student implements Comparable{

…
     public int compareTo(Object other)
     {
       if(this.regNos>((Student)other).regNos)
          return 1;
       if(this.regNos==((Student)other).regNos)
          return 0;
       return -1;
     }
}
```

Note the need for casting. Generics will allow us to overcome this

Inheritance In Java vs C++

JAVA: A class can only extend one other class

C++: multiple inheritance is allowed

JAVA: Everything inherits from Object

C++: No built in inheritance hierarchy

JAVA: All methods **can be** overridden unless specifically made final

C++: All member functions **cannot be** overridden unless specifically made virtual

JAVA: Interfaces are a special construct that consist of a collection of abstract methods

C++: No equivalent language construct

1. Understanding Inheritance
2. Inheritance Programming Points
3. **Using Inheritance**

3. Using Inheritance

The following is applicable to any programming language that provides inheritance

3.1 Code reuse
3.2 As an alternative to selection
3.3 To decouple an application from the implementation
3.4 To enable general methods

The basic software engineering motivation for inheritance is to make classes as loosely decoupled as possible, and to reduce the number of potential unforseen consequences.

Code Reuse

- The most obvious benefit of inheritance is that you don’t have to rewrite all the code from the base class
- However, you get the same effect through composition and its less confusing.
- You should only use inheritance when the relationship between the classes you are modelling warrants it
- In practice, you will use composition much more than inheritance. *Most of the programming patterns involve composition*.

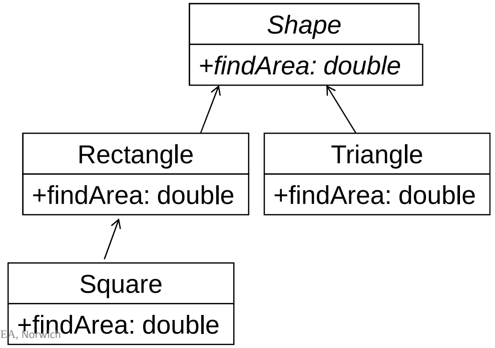
3.2 Separating an application from the classes it uses

- Suppose we want to use shapes in a range of graphics programs.
- We will use rectangles, squares, triangles and circle.
- These may be used in a large number of different applications
- There are two ways to model the shape

With a variable in `Shape` to indicate shape type

```
public class Shape{
      char shapeType;
//Square =s, Triangle =t
//Circle=c, Rect =r
      double area(){
…
      }
}
```

With inheritance



Object Based Programming: Selecting between alternatives

```
public class Shape{
 char shapeType;
 double h,w,r;
 double area(){
  switch(shapeType){
      case ‘r’: case ‘s’:
        return h*w;
      case ‘t’:
        return 0.5*h*w;
      case ‘c’:
        return Math.PI*r*r;
      }
  }
}
```

Note there may be a large number of methods that need to do this selection

This design pattern can get very big very quickly (code bloat)

If we add another Shape to our class, we will have to alter every method where this selection occurs.

```
Usage:
Shape s= //Get shape from somewhere
double area=s.area();
```

Control passed to method `area` where selection occurs

Object Oriented Programming: Selecting by dynamic binding

```
interface Shape{
 double area();
//Etc
}
```

```
Public class Rectangle implements Shape{
 double h,w;
 double area(){return h*w};
}
```

```
Public class Triangle implements Shape{
 double h,b;
 double area(){return 0.5*h*b};
}
```

```
Public class Square extends Rectangle {
//Specific square stuff here
}
```

If we add another Shape to our inheritance hierarchy, we do not have to change any of the existing code.

```
Usage:
Shape s= //Get shape from somewhere
double area=s.area();
```

Selection happens by dynamic binding **before** method `area` is called

3. Using Inheritance

3.1 Inheritance as an alternative to selection

3.2 Changing implementation without changing application

3.3 Creating general methods

3.2. Changing implementation without changing application

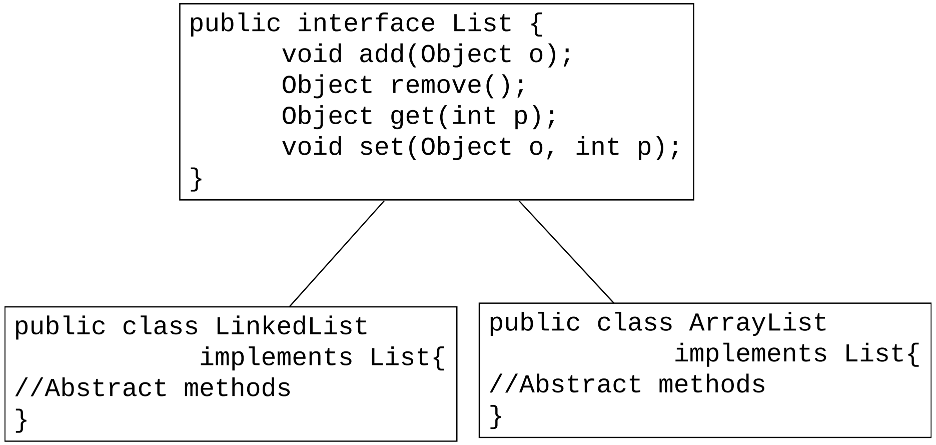
Suppose you want to write an application that will manipulate a `Collection` of objects

For example, we might be writing a program to manage a print queue

```
Public class printManager{
   List myData;
   public printManager(List m){
   myData =m;
}
   public void processJob(Object o){
//Add, remove, get and set values in the list
/* This could be very complex code
   myData.add()      myData.remove()
   myData.get()      myData.set()
   sorting, */
}
```

3.2. Changing implementation without changing application

The application `printManager` needs a List. The actual implementation of `List` is irrelevant to the method `processJob`



```
public interface List {
      void add(Object o);
      Object remove();
      Object get(int p);
      void set(Object o, int p);
}
```

```
public class LinkedList
            implements List{
//Abstract methods
}
```

```
public class ArrayList
            implements List{
//Abstract methods
}
```

3.2. Changing implementation without changing application

A program that uses printManager can alter the data structure (i.e. the implementation) without having to change the application

```
public static void main(){
PrintManager office, home;

office = new PrintManager(new ArrayList());
// Use in some way, m.processJob(“First”);

home = new PrintManager(new LinkedList());
// Use in a different way, hence LinkedList better

}
```

This is good because the best data structure to use will be different depending on how it is used.

3. Using Inheritance

3.1 Inheritance as an alternative to selection

3.2 Changing implementation without changing application

3.3 Creating generally applicable methods

3.3 Creating general methods

Suppose we would like to write a class to provide tools to sort arrays using the selection sort algorithm

	8
	5
	2
	6
	9
	3
	1
	4
	0
	7

```
Task: Selection Sort: Sorts Array T[1...n] into ascending order
integer pos;
element min;
for i=1 to n-1 loop
        Find position of smallest element in T[i .. n]
                pos=i; min=T[i]
for j=i+1 to n loop
        if T[j]<min
                pos=j; min=T[j]
        Swap element at smallest position with element at i
                T[pos] = T[i]; T[i]=min
```

3.3 Creating general methods

Ideally we would like to be able to write a single method that could sort an array of any type of Object. We don’t want to write a different method for each class we define

```
public class MySorting{
        public static void selectionSort(Student[] s){
        …
        }
        public static void selectionSort(House[] s){
                …
        }
        public static void selectionSort( Card[] s){
                …
        }
```

This code is not particularly useful!

The key requirement for sorting is to be able to compare two objects. If we pass an array of `Comparable` objects, then we can **guarantee** the method `compareTo` is implemented

```
public class MySorting{
public static void selectionSort(Comparable [] n){
                int min,i,j;
                Comparable temp;
                for(i=0;i<n.length-1;i++){
                        min=i;
                        for(j=i+1;j<n.length;j++){
                                if( n[j].compareTo(n[min])<0)
                                        min=j;
                        }
                        if(i!=min){
                                temp=n[i];
                                n[i]=n[min];
                                n[min]=temp;
                        }
                }
        }
```

This method can sort *any* array of objects (of the same type) which implement the comparable interface

The restrictions are:

1. It cannot be used on an array of objects (*not all objects are comparable*)
2. To use with primitive types would have to use a wrapper class *(this is common in java)*
3. **We can only compare the objects in a single way, and after compilation we cannot change this.** Frequently the application will determine which field we want to sort an array by (e.g. spread sheet program). We may want to sort our array of Students by score, alphabetically, by age, etc.

General Sort Routine 2: Functors

- An alternative is to pass as an argument the method that will be used to compare
- In C++ you would use a *function pointer*
- In java, you wrap the compare method in a new class that implements the `Comparator` interface

```
public interface Comparator {
    int compare(Object o1, Object o2);
}
```

```
public class MySorting{

public static void selectionSort(Object[] n, Comparator
cmp ){
            int min,i,j;
            Object temp;
            for(i=0;i<n.length-1;i++){
                  min=i;
                  for(j=i+1;j<n.length;j++){
                  if(cmp.compare(n[j],n[min])<0)
                        min=j;
                  }
                  if(i!=min){
                        temp=n[i];
                        n[i]=n[min];
                        n[min]=temp;
                  }
            }
      }
```

```
class CompareByScore implements Comparator{
public int compare(Object obj1, Object obj2){
      return ((Student)obj1).score-((Student)obj2).score;
}
}

class CompareByName implements Comparator{
public int compare(Object obj1, Object obj2){
      Student s1 = (Student)obj1;
      Student s2= (Student)obj2;
      return s1.name.compareTo(s2.name);
}
}
```

Functors are usually defined as nested classes (we will cover this later)

```
To use selectionSort with a functor
Object[] myCourse= new Student[5];
Comparator functor1 = new CompareByScore();
//Create array of students
//...
// Sort by score

Sort.selectionSort(myCourse, functor1);
//Sort by name

Sort.selectionSort(myCourse,new CompareByName());
```