Programming 2 : Lecture 2.

Nested Classes and Enums

- **Nested Classes pages 1-7**
  - *Static nested classes*
  - *Nested classes*
  - *Local classes*
  - *Anonymous classes*

- **`enum` Types pages 7-11**
- **`Number` classes pages 11-12**
- **Ellipsis operator ...page 12**

Nested classes

- *A class within another class is called a nested class or inner class*
- *There are four types of nested class in Java (in C++ there is just one type)*
- *They provide a logical way of grouping classes and lead to more comprehensible code*

Nested Classes

- A class defined within another class is called a ***nested class***
- There are four types of nested classes
  1. **Static nested classes**. These do not have access to members of the enclosing class.
  2. **Non-static nested classes** or **inner classes**. They have access to members of the enclosing class, even if they are declared private.
  3. **Local inner classes**. These are defined within a method
  4. **anonymous inner classes.** These are defined without a name

Benefits of Nested Classes

*"The use of nested classes in Java is a constant source of confusion for many programmers."**

1. They provide a way of logically grouping classes that are only used in one place.
2. They allow for increased encapsulation.
3. They can lead to more readable and maintainable code.
4. They are used to implement UML composition relationships

***Quote from Hardcore Java by Robert Simmons Jr**

1. Static Nested Classes

•Static nested classes are defined within another class with the reserved word static before the class definition

•they are not associated with an object, rather they pertain to a class

```
class OuterClass {
            ...
            public static class NestedClass {
            ...
            }
}
```

```
OuterClass.NestedClass in=
                new OuterClass.NestedClass();
```


•The name of the static nested class is **`OuterClass.NestedClass`**

```
public class OuterClass {
        NestedClass field;
        ...
        public void aMethod(){
                NestedClass local=new NestedClass ();
        }
        public static void staticMethod(){
                NestedClass local=new NestedClass ();
        }
        public static class NestedClass {
                ...
        }
}
```

`NestedClass` may be a field of **`OuterClass`**

`NestedClass` can be declared in a method or static method

•If a static nested class is public, you can create instances of the nested class outside **`OuterClass`** with this code

```
OuterClass.NestedClass in= new OuterClass.NestedClass();
```

Static Nested Class Example

```
public class ColourPalate{
      private Colour[] myPalate;
      int size=0;
      int maxSize=100;
      public ColourPalate(){
            myPalate=new Colour[maxSize];
      }
      public void addRGB(int r, int g, int b){
            myPalate[size++]=new Colour(r,g,b);
      }
      public static class Colour{
            int red;int green;int blue;
            public Colour(int r, int g, int b){
                  red=r;green=g;blue=b;
            }
            public int toGreyScale(){
                  return red+green+blue;}
      }
}
```

ColourPalate has an array of Colour objects

Static nested class

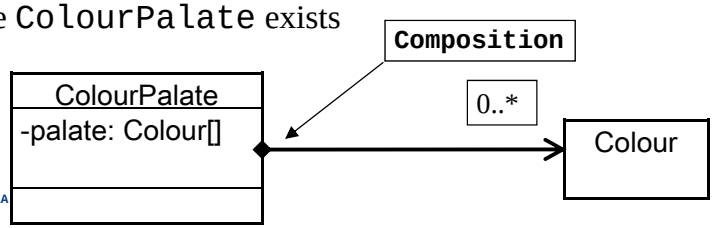
Static Nested Class Example

Because Colour is public, we can use it anywhere (if we import it)

```
import ClourPalate.Colour;

public class SomeClass{
      public static main(String[] args){
            Colour black = new Colour(0,0,0);
      }
}
```

The relationship we are modelling here is **Composition**. `ColourPalate` contains `Colour` objects that exist as long as the `ColourPalate` exists



Using Static Nested Classes

- Of all the nested classes, static nested classes are the easiest to understand and use.
- Use them unless the nested class must have access to the fields of a particular object of the top-level class.
- Nested classes can access the various static members of the enclosing class
- You can use these small helper classes outside the top-level class if you make them public
- Nested classes can also be private and protected
- Nested classes can be final or abstract
- **C++ only has this type of nested class**

2. Non-Static Nested Classes: Inner Classes

- Non-static nested classes are usually called ***inner classes***

```
class OuterClass {
            ...
            class InnerClass {
            ...
            }
}
```

Inner Class

The only difference in syntax is the absence of the keyword static

Inner classes are always associated with a particular object.

2. Non-Static Nested Classes: Inner Classes First example

```
public class OuterClass {
         int a;
         int b;
         public OuterClass(int x, int y){
              a=x;
              b=y;
       }
       public class InnerClass {
         int sum;
         int findSum(){
              sum=a+b;
              return sum;
         }
       }
}
```

Inner Classes cannot exist in isolation

Inner classes are always associated with a particular object and hence can reference The attributes of the outer class

```
OuterClass out=new OuterObject(1,2);
OuterClass.InnerClass in=out.new InnerClass();
int s=in.findSum();
```

Returns 3

2. Non-Static Nested Classes: Inner Classes

Inner Classes cannot exist in isolation

```
public class OuterClass {
        InnerClass field;
       ...
       public void aMethod(){
              InnerClass local=new InnerClass ();
       }
       public static void staticMethod(){
       // ERROR:     InnerClass local=new InnerClass ();
       }
       public class InnerClass {
              ...
       }
}
```

InnerClass may be a field of **OuterClass**

InnerClass CANNOT be a field of a static method on its own

Inner classes are always associated with a particular object. To create one outside of the outer class you have to do this

```
OuterClass out=new OuterObject();
OuterClass.InnerClass in=out.new InnerClass();
```

Inner Class Example

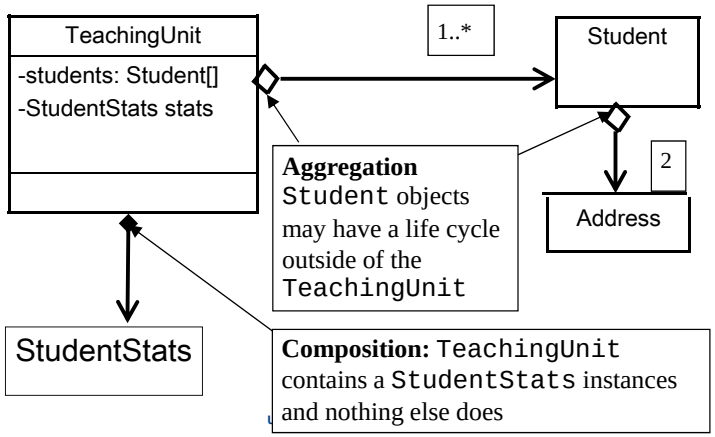
```
public class TeachingUnit {
      Student[] s;
      StudentStats stats=new StudentStats();


      private class StudentStats{
      double maxMark;
      public void findStats(){
            maxMark=s[0].getUnitMark();
            for(int i=1;i<s.length;i++){
                  double x=s[i].getUnitMark();
                  if(x>maxMark)
                        maxMark=x;
      }
      public double getMax(){return stats.maxMark;}
}
```

The inner class accesses the data of the enclosing class

Inner Class UML

Generally inner classes model **composition**



Using Inner Classes

Inner classes should only be used outside the top level class if they implement an interface.

```
OuterClass out=new OuterObject();
SomeInterface in=out.factoryMethod();
```

This returns an object that implements **`SomeInterface`** and is obviously connected to the object **`out`**

Using an interface in this way is a common pattern that encapsulates the inner class. The returned object performs some task on the outer object it is associated with.
We will see this with iterators

3. Local Inner Classes

•Local inner classes are defined within a method. Multiple instances can be created, **but only in the method within which the class is defined.**

```
public void someMethod() {
      class LocalInnerClass {

      }
      LocalInnerClass lm1=new LocalInnerClass();
      LocalInnerClass lm2=new LocalInnerClass();

}
```

•Local inner classes are hardly ever used. Most Java programmers probably don't even realise they exist!

4. Anonymous Inner Classes

•Anonymous inner classes are defined within a method and can only be created **once**

```
public void someMethod() {
     SomeInterface s=new SomeInterface(){
     public int interfaceMethod(){return 0;}
   };
}
```

They are almost always a throw away instantiation of some interface. Anonymous inner classes are more common than local inner classes, but still quite unusual.

Uses for Anonymous Classes

You will most likely see anonymous classes in swing or threaded applications

```
JButton button= new JButton("My Button");
button.addActionListener(new ActionListener(){
      public void actionPerformed(ActionEvent e){
            //do stuff do here
      }
};
```

```
new Thread(
      new Runnable() {
            public void run()
            { // do stuff
            }
      }
).start();
```

Nested Class Example: Comparator

Remember we used the **`Comparator`** interface for sorting

For example, the sort method in `util.Arrays` can be passed a `Comparator`

```
static void sort(Object[] a, Comparator c)
```

The **`Comparator`** interface contains a single abstract method called compare

```
java.util Interface Comparator<T>
```

```
int compare(T o1, T o2)
```

It makes sense to define Comparators as nested classes. We could use any of the four nested class patterns to do this.

Pattern 1:
Static Nested Class

This is the best pattern for Comparators

```
import java.util.*;

public class Student {
int age;
String name;

public static class ComparebyAge implements Comparator{
   public int compare(Object a, Object b){
      return (((Student)a).age-((Student)b).age);
   }
}
public static class ComparebyName implements Comparator{
   public int compare(Object a, Object b){
return ((Student)a).name.compareTo(((Student)b).name);
   }
}
}
```

Pattern 2: Inner Class

Not so good.
This means we have to create a Student object if we want to call these methods in another class

```
import java.util.*;

public class Student {
int age;
String name;

public class CompByAge implements Comparator{
   public int compare(Object a, Object b){
      return (((Student)a).age-((Student)b).age);
   }
}
public class CompByName implements Comparator{
   public int compare(Object a, Object b){
return ((Student)a).name.compareTo(((Student)b).name);
   }
}
}
```


Using Pattern 1 and 2

```
Student[] myClass;
myClass=loadStudents("myFile");

Comparator compName=new ComparebyName();
Arrays.sort(myClass,compName);
Arrays.sort(myClass,new ComparebyAge());

Comparator inner = myClass[0].new CompByName();
Arrays.sort(myClass,inner);
Arrays.sort(myClass, myClass[0].new CompByAge());
```

Static nested objects

Inner objects associated with **`myClass[0]`**

Here, inner classes are confusing because the comparator has nothing to do with **`myClass[0]`**

Pattern 3: Local Inner Class

```
public static void main(String[] arg){
  Student[] myClass;
  myClass=loadStudents("myFile");
//Sort by mark in ascending order
  final class CompareByMark implements Comparator{
  public int compare(Object a, Object b){
      return (int)(((Student)b).avMark-
                   ((Student)a).avMark);
      }
  }
  Arrays.sort(myClass,new CompareByMark());
}
```

Local inner class defined **within** a method

This class can only be used within the methods where the local class is defined. It can be used more than once

Pattern 4: Anonymous Inner Class

```
public static void someMeth(){
  Student[] myClass;
  myClass=loadStudents("myFile");
//Sort names in age ascending order as a one off
      Arrays.sort(myClass,
      new Comparator(){
      public int compare(Object a, Object b){
              return (((Student)b).age-((Student)a).age);
            }
      });
      Comparator c=new Comparator(){
       public int compare(Object a, Object b){
        return (((Student)b).score-((Student)a). score);
        }
       });
      Arrays.sort(myClass,c);

}
```

This creates an object of a completely one off type

We can store the one off object in an Interface reference

Inheritance with Nested Classes

```
public class BaseClass {
   int anInt;
   InnerBase ib1 =new InnerBase();
   protected class InnerBase{ protected double x;}
}
```

```
public class SubClass extends BaseClass{

  InnerBase ib2   =     new InnerBase();
  InnerChild ic1  =     new InnerChild();

  public class InnerChild extends InnerBase{
      public String str;
  }
}
```

Inner classes AND static nested classes can be extended in a subclass.

An instance of Subclass now contains all of the following

```
SubClass sub=new SubClass();
sub.anInt;
sub.ib1.x;
sub.ib2.x;
sub.ic1.x;
sub.ic1.str;
```

There is a folder of files demonstrating Nested classes on blackboard

Inner Interfaces

Inner Interfaces are allowed in other classes and Interfaces

```
Interface LinkedList{
     void add(Object e);
     void remove(Object e);

     interface ListNode{
          void connect(ListNode e);
     }
     interface ListIterator{
     }
```

This is ok if you expect it to be used only from the outer class. You are essentially creating an interface `LinkedList.ListNode`

Inner interfaces are static and public.

Nested Classes Summary

If an instantiation of a class relates to the outer class as a whole and not to a specific object, then make it a static nested class*

We will see static classes **`ListNode`** and **`TreeNode`** that are related to **`LinkedList`** and **`BinaryTree`**

If an instantiation of a class needs access to a particular outer class Objects data, then it should be an non-static inner class

We will see **`Iterator`** classes that relate to a specific data structure, so may be inner classes.

Local inner classes and anonymous classes are used to create one off instances of specialised classes and are rarely used. You will most commonly see them in swing applications to create one off listeners

(*) In C++ there is only one type of nested class which is equivalent to Java static nested classes. The other Java nested classes are a feature you *can* use, but they are not essential.

Enum Type

- *enum types are a means of modelling a variable with a discrete, finite number of values*
- *They provide type safety, modularity, clarity and flexibility*
- *In Java, they are in fact instances of anonymous inner classes*

Enumerating

Suppose we want to model enumerated data types

Country: Can be one of {England, Spain, France}
Season: {Spring, Summer, Autumn, Winter}
Card Suit: {Clubs, Spades, Diamonds, Hearts}
Card Value: {Two, Three, …, Ten, Jack, Queen, King, Ace}
Gene Values: {A,C,G,T}

Prior to java 1.5, the programming pattern you would use to model these would involve declaring global statics

```
public static final int SEASON_WINTER = 0;
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_AUTUMN= 3;
```


Enumerating

```
public static final int SEASON_WINTER = 0;
public static final int SEASON_SPRING = 1;
public static final int SEASON_SUMMER = 2;
public static final int SEASON_AUTUMN= 3;
```

•**Not typesafe -** you can pass in any other int value where a season is required, or add two seasons together (which makes no sense).
•**Brittleness -** If a new constant is added between two existing constants or the order is changed, clients must be recompiled. If they are not, they will still run, but their behaviour will be undefined.
•**Printed values are uninformative -** Because they are just ints, if you print one out all you get is a number, which tells you nothing about what it represents, or even what type it is.

Enumerating

Don't confuse `enum` types with the `Enumeration` interface, which is an old form of Iterator

```
public class EnumExamples {
enum Season{WINTER, SPRING, SUMMER, AUTUMN}
enum Suit{CLUBS, SPADES, DIAMONDS, HEARTS}

static public boolean isSummer(Season s){
      return s==Season.SUMMER;}

public static void main(String[] args){
      Season thisSeason=Season.SUMMER;
      Suit myCard=Suit.CLUBS;
      if(isSummer(thisSeason))
            System.out.println("Its Summer ="+thisSeason);
}
```

Convention to capitalise the constants

When to Use Enum

You should use enum types whenever you need to represent a fixed number of constants. They make your code more readable

Days of the week
```
enum Days{MONDAY, TUESDAY,…, SUNDAY}
```
Months of the year
```
enum Months{JAN,FEB,MAR,…,DEC}
```
Colours
```
enum Colours{RED,GREEN,BLUE}
```
Planets
```
enum Planets{MERCURY,VENUS,MARS}
```

In C++, enums are just integers, and can be treated as such. In Java, they are in fact each an instantiation of an anonymous class

Playing Cards Example

```
public class Deck{
    public enum Rank {TWO, THREE, … KING, ACE }
    public enum Suit { CLUBS, DIAMONDS, HEARTS, SPADES }

    private ArrayList<Card> deck;

      public static class Card {
        private Rank rank;
        private Suit suit;
        private Card(Rank rank, Suit suit) {
          this.rank = rank;
          this.suit = suit;
        }
        public Rank rank() { return rank; }
        public Suit suit() { return suit; }
        public String toString()
        { return rank + " of " + suit; }
}
```

Nested Class

enum attributes


```
public static class Card {
private Rank r;
private Suit s;
private Card(Rank r, Suit s) {
        this.r = r;
        this.s = s;
    }
```

TYPE SAFE: If you accidentally invoke the constructor with the parameters reversed, the compiler will inform you of your error.

INFORMATIVE. Don't have to rely on variable names to understand meaning

Contrast this to the int enum programming pattern, in which the program would fail at run time.

```
public static class Card {
private int r;
private int s;
private Card(int r, int s) {
        this.r = r;
        this.s = s;
    }
```

Enum Extra Features

Enum is in fact a class and has several useful built in methods

1. The static method `values()` returns the possible values as an array

```
Rank[] vals=Rank.values();
```

2. The method `ordinal()` returns the rank of a particular enum as an integer

```
Rank r= Rank.TWO;
int s=r.ordinal();
```

Rank = TWO has ordinal =0

Enum Iteration

We can use the `values()` method to iterate over all values with `for…each`

```
public Deck()
   {
     deck=new ArrayList<Card>();
     for(Rank r : Rank.values())
           for(Suit s: Suit.values())
                 deck.add(new Card(r,s));
     shuffle();

}

public Card deal(){
   return deck.remove(0);
}
```


Advanced Usage of Enums

Each enum type is in fact the instantiation of an anonymous inner class.

```
enum Grade{FIRST, TWO_ONE, TWO_TWO, THIRD, FAIL}
```

This means I can do strange things with enum. Suppose I want to associate a number to each enum type (e.g. the grade boundary).

```
enum Grade{
FIRST(70), TWO_ONE(60), TWO_TWO(50), THIRD(40), FAIL(0);
      final int boundary;
      Grade(int x){
            boundary=x;
      }
      public double getBoundary(){return boundary;}
}
```


Advanced Usage of Enums

Each of these creates an instance of Grade, the value in the brackets is passed to the construtor

```
enum Grade{
FIRST(70), TWO_ONE(60), TWO_TWO(50), THIRD(40), FAIL(0);

      final int boundary;
      Grade(int x){
            boundary=x;
      }
}
```

this bit defines the class data and methods for an instance of Grade

```
Grade g=Grade.FIRST;
System.out.print("g = "+g.boundary);
```


Advanced Usage of Enums

```
public enum Planet {
  MERCURY (3.303e+23, 2.4397e6),
  VENUS (4.869e+24, 6.0518e6),
  EARTH (5.976e+24, 6.37814e6);
private final double mass; // in kilograms
private final double radius; // in meters

Planet(double mass, double radius) {
   this.mass = mass; this.radius = radius;
}
public static final double G = 6.67300E-11;
```

Advanced Usage of Enums

```
public double surfaceGravity() {
   return G * mass / (radius * radius); }

public double surfaceWeight(double otherMass) { return
   otherMass * surfaceGravity(); }

public static void main(String[] args)
{
   double earthWeight =85;
   double mass =
   earthWeight/Planet.EARTH.surfaceGravity();
   for (Planet p : Planet.values())
      System.out.println("Your weight on Planet:"+p+" ="+
   p.surfaceWeight(mass));
}
```

Advanced Usage of Enums

You can actually make `enums` mutable.

```
enum Grade{
FIRST(70), TWO_ONE(60), TWO_TWO(50), THIRD(40), FAIL(0);
      int boundary;
      Grade(int x){boundary=x;}
      public void setBoundary(int x){boundary=x;}
}
```

```
Grade g=Grade.FIRST;
Grade g2=Grade.FIRST;
g2.setBoundary(33);
```

this changes the value for ALL instances of FIRST.

If you need this kind of flexibility, you generally wouldn't use enums. However, it can be useful with threads because enums are a good way of modelling the Singleton software engineering pattern.

What Are Enums?

Each value of an enum is an instantiation of an anonymous inner class.

```
enum Grade{
FIRST(70), TWO_ONE(60), TWO_TWO(50), THIRD(40),FAIL(0);
      final int boundary;
      Grade(int x){
            boundary=x;
      }
```

Is equivalent to this:

```
static public abstract class Grade{
     final int boundary;
     private Grade(int b){ boundary=b;}
     public static final Grade FIRST = new Grade(70){};
     public static final Grade TWO_ONE = new Grade(60){};
     public static final Grade TWO_TWO = new Grade(50){};
     public static final Grade THIRD = new Grade(40){};
     public static final Grade FAIL = new Grade(0){};

 }
```

`enum` Summary

`enum` types are a mechanism for modelling variable with a discrete number of values `enum Colours{RED,GREEN,BLUE}`

The **benefits** of **`enum`** are:

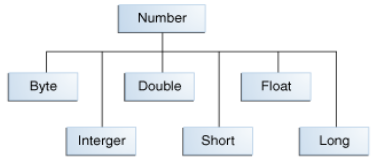
1. **Type Safety**: errors with usage can be detected at compile time rather than run time
2. **Clarity:** easier to understand code with **`enum`**
3. **Modularity:** we can add values to an existing **`enum`** (e.g. add a colour) without changing any code that uses the enum type
4. **Flexibility:** built in functions like **`values()`** help us write tidy code.

Each value for an **`enum`** is in fact an instance of an anonymous inner class, and hence can have methods and variables associated with it

`Number` Class and Ellipsis Operator …

- *`Number` classes are primitive wrappers*
- *The ellipsis operator … allows methods to have a variable number of arguments*

`Number` class: Primitive Wrappers



The abstract class Number is the superclass of classes **`Byte`, `Double`, `Float`, `Integer`, `Long`** and **`Short`**.

use a **`Number`** rather than a primitive:

1.As an argument of a method that expects an object (often used when manipulating collections of numbers). E.g. **`ArrayList<Integer>`**

2.To use constants defined by the class, such as **`MIN_VALUE`**

3.To use class methods for converting values to and from other primitive types, for converting to and from strings, and for converting between number systems (decimal, octal, hexadecimal, binary).

`Number` class: Primitive Wrappers

- You can use primitives and Numbers interchangeably.
- **This inbuilt feature is called "Boxing"**
- You can also use them with primitive operators *(built in operator overloading)*

```
Integer a=10;
Integer b=2;
int c,d=20;
b=d;
a=d;
c=a+b*c-d;
```

So why use primitives at all?
Because there is an overhead associated with using the primitive wrappers (at least 4 bytes for the reference)

classes **`BigDecimal`, `BigInteger`** provide a mechanism for protecting agains overflow

`ellipsis` Operator

Java allows methods with a variable argument list

```
public static void average(double...numbers)
```

This use of an **ellipsis** was new in java 1.5. It allows the user to enter a variable number of arguments of type

Note that in Java you **cannot** assign default values to parameters (you can in C++)

Variable Argument Lists

```
public static double average(double...numbers){
 double s=0;
 for(int i=0;i<numbers.length;i++)
     s+=numbers[i];
  return s/numbers.length;
}

public static void main(String[] args){
  double d1=100,d2=33,d3=333;
  double mean;
  mean=average(d1);
  mean=average(d1,d2);
  mean=average(d1,d2,d3);

}
```

numbers is just an array

The JVM works out how big numbers needs to be and puts your arguments in the array

Summary

NESTED CLASSES: Class within another class

ENUM TYPES: representing a restricted type data

NUMBER CLASSES: Wrappers for primitives

ELLIPSIS: Variable argument lists for a method