CompSci 230 S2
Object Oriented Software Development

Nested Classes

Review Quizzes

- Consider the following:

```
abstract class A {
  public void templateMethod() {
    operation1();
    operation3();
  }
  abstract public void operation1();
  public void operation3() {
    System.out.print("A-op3 ");
  }
}
class B extends A {
  public void operation1() {
    System.out.print("B-op1 ");
  }
}
```

- What is the output of the fol

```
B b1 = new B();
b1.operation1();
```

- What is the output of the following code?

```
A b1 = new B();
a1.operation3();
```

- What is the output of the following code?

```
A b1 = new B();
a1.templateMethod();
```

Review Quizzes

- Consider the following:

```
interface FinancialAidEligible { }
abstract class Person {
  int ID;
  public int getID() { return ID; }
}
class Student extends Person {}
class Undergraduate extends Student implements FinancialAidEligible {}
```

- Which of the following statements is/are LEGAL

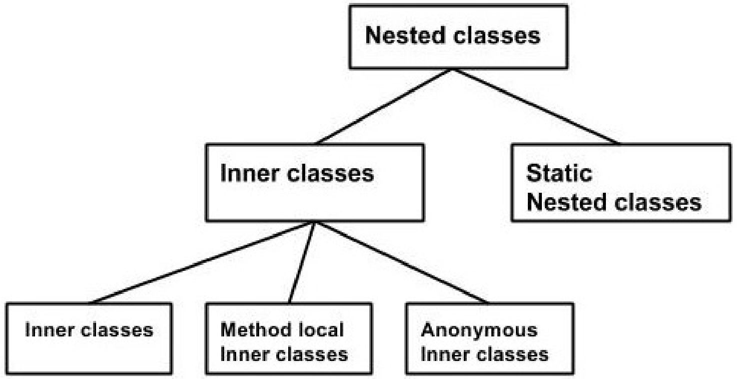
```
I.  FinancialAidEligible p1 = new Undergraduate();

II.FinancialAidEligible p2 = new FinancialAidEligible();

III.FinancialAidEligible p3 = new Student();

IV.  FinancialAidEligible[] people = new FinancialAidEligible[10];
```

Agenda & Reading

- Topics:
  - Introduction
  - Static (Static Nested Classes)
  - Non-Static (Inner Classes)
    - Member Classes
      - □ MyStack with Inner Member Class
      - □ MyStack Without Nested class
    - Local Classes
    - Anonymous Classes
      - □ MyStack with Inner Anonymous
      - □ WindowAdapter
- Reading
  - Java how to program Late objects version (D & D)
    - Chapter 12
  - The Java Tutorial:
    - Nested Classes

1.Introduction
Nested Classes

- A class defined inside another class
  - Some classes only make sense in the context of another enclosing class
    - A GUI event handler cannot exist by itself, only in association with a GUI component it handles events for (Example:ActionListener)
  - We use nested classes to reflect and enforce the relationship between two classes
- A nested class can be declared as:
  - Static (Static Nested classes)
  - Non-Static (Inner class)



```
public class MyRegularClass {
 ...
  class MyInnerClass {
      ...
  }


}
```

Outer class

Inner class

Inner class:
Complete access to the fields and methods of its enclosing class, including those declared private

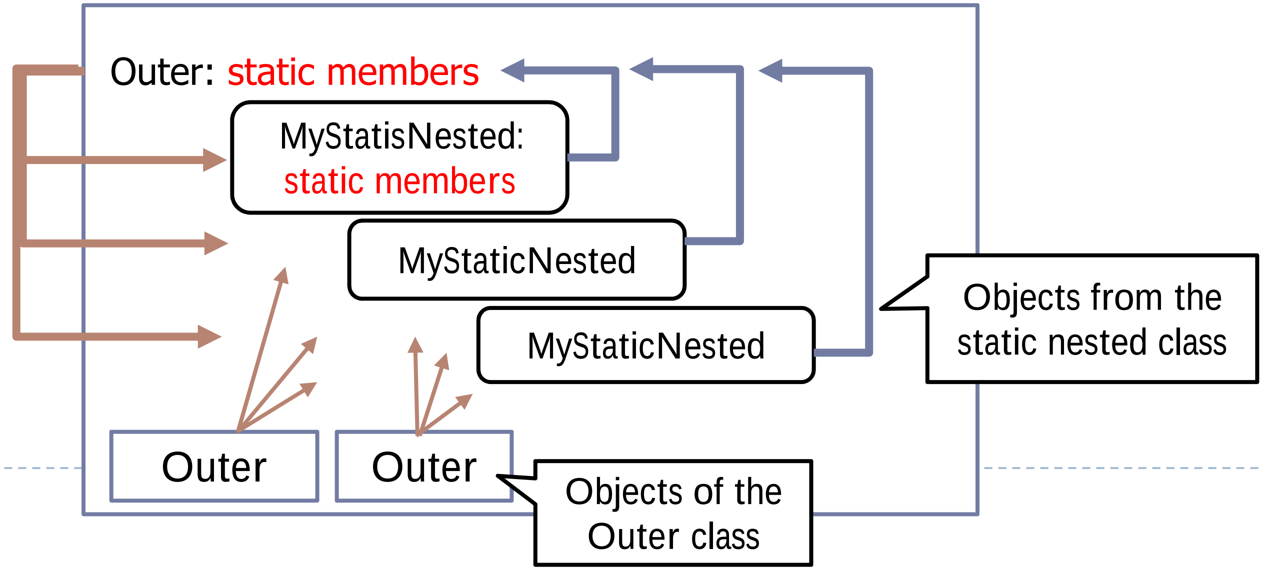
.class created for the inner class:
MyRegularClass$MyInnerClass.class

1.Introduction
Why use Nested Classes?

- It is a way of **logically grouping classes** that are only used in one place:
  - If a class is useful to only one other class, then it is logical to **embed** it in that class and keep the two together. i.e. helping classes
- It increases **encapsulation**:
  - Consider two top-level classes, A and B, where B needs access to members of A that would otherwise be declared private. By hiding class B within class A, A's members can be declared private and B can access them. In addition, B itself can be hidden from the outside world.
- It can lead to more **readable** and **maintainable** code:
  - Nesting small classes within top-level classes places the code closer to where it is used.

2.Static Nested Classes

- If you want to make objects of a nested class type **independent** of objects of the enclosing class type, you can declare the nested class as **static**:
  - can access **static members and variables** of its outer class (even declared as **private**)
  - **cannot** refer to any instance members
  - is associated with its enclosing/outer class
    - behaviour as any static member of the outer class
    - may be instantiated/accessed without an instance of the outer class



Objects from the static nested class

Objects of the Outer class

2.Static Nested Classes
Creating objects of the static nested class

- You can declare **objects of this nested** class type **independent** from any objects of the **outer class**, and regardless of whether you have created any outer objects or not.
  - i.e. don't need to create an object of the Outer class, like the other static member.

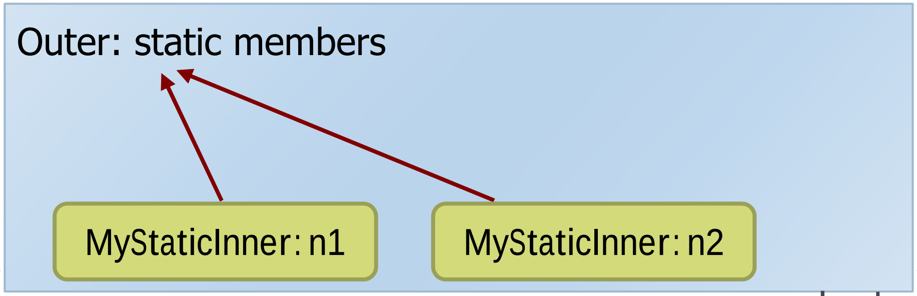
```
//from any other classes
Outer.MyStaticInner n1 = new Outer.MyStaticInner();
```

Outside the outer class, access it by its outer class name

```
// within the Outer class itself
 MyStaticInner n2 = new MyStaticInner();
```

- Note: No access to instance members of the enclosing class (outer)

```
public class Outer {
  private int x;
  private static int count = 10;
  ...
  public static class MyStaticInner {
    ...

  }
}
```



2.Static Nested Classes Accessing Methods and Fields

Outer.java

- The enclosing class (Outer) has full access to the static nested class (MyStaticInner)

```
public class Outer {
  private int x=1;
  private static int count = 10;

  ...
  public static void outerStaticMethod() {
    MyStaticInner b = new MyStaticInner();
    System.out.println("b.x=" + b.x + ",b.count=" + b.count);
  }

  public static class MyStaticInner {
    private int x=2;
    private static int count=20;
    ...
  }

  public static void main(String[] args) {
    ...
    outerStaticMethod();
  }
}
```

Create a new instance of the static nested

```
b.x=2,b.count=20
```

2.Static Nested Classes Accessing Methods and Fields

- The Static Nested class (MyStaticInner) can access to the **static members** of the outer class.
  - e.g. count, g()
- But it **cannot** access to instance members (Outer.x)

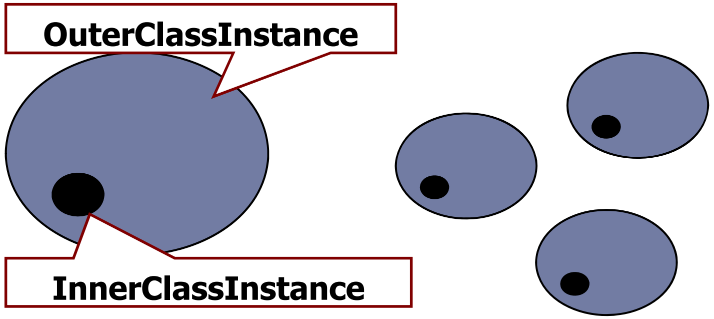
```
public class Outer {
  private int x=1;
  private static int count = 10;
  public static void g() { System.out.println("g"); }
  ...
  public static class MyStaticInner {
    private int x=2;
    private static int count=20;
    public void  instanceMethod() {
      System.out.println("x=" + x);  //itself
      System.out.println("count=" + count); //itself
      System.out.println("Get Outer.count=" + Outer.count);
      MyOuter.g();
    }
    ...
  }
  public static void main(String[] args) {
    Outer.MyStaticInner n1 = new Outer.MyStaticInner();
    n1.instanceMethod();
    ...
```

Create a new instance of the static nested

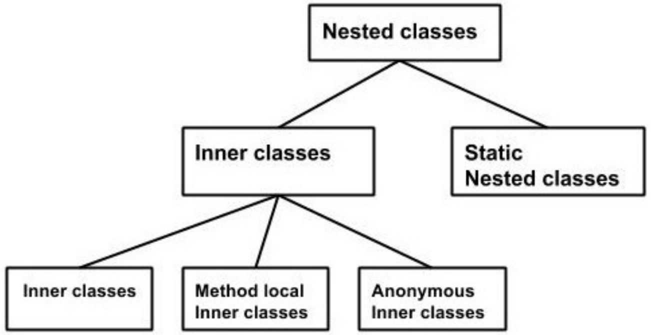
3.Inner Classes

- A nested class that is associated with an **instance** of its outer class, is known as an inner class.

```
public class MyRegularClass {
 ...
  ...class MyInnerClass {
       ...
  }
}
```



- Non-Static (Inner Classes)
  - Member Classes :A member class is defined within the body of a class
  - Local Classes:A local class is a class defined within a method
  - Anonymous Classes:A local class is declared implicitly by creating a variable of it



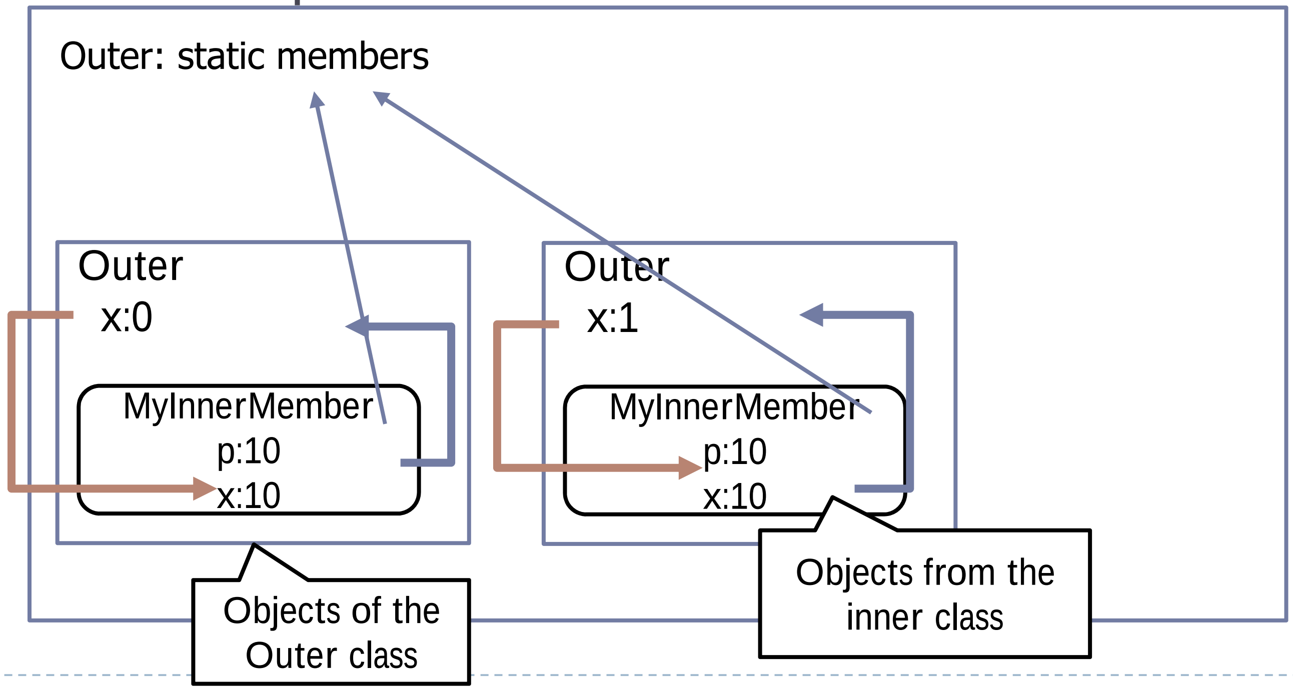
4.Member Classes

- Definition
  - A member class is not declared static
  - A member class is defined within the body of a class
- Rules
  - Member classes **cannot** declare **static variables and methods**
  - The member class has access to all instance and class variables and objects of the outer class or any other inner classes, including members declared private
  - The outer class has access to all the variables and methods in the inner class

```
public class Outer {
 ...

   class Member {
       ...
   }
}
```

4.Member Classes

- An instance of an inner class is effectively inside an existing instance of the outer class
  - An inner class has **access** to all the methods and variables associated with the instance of the outer class including members with the private access modifier.



4.Member Classes
Creating objects of the inner class

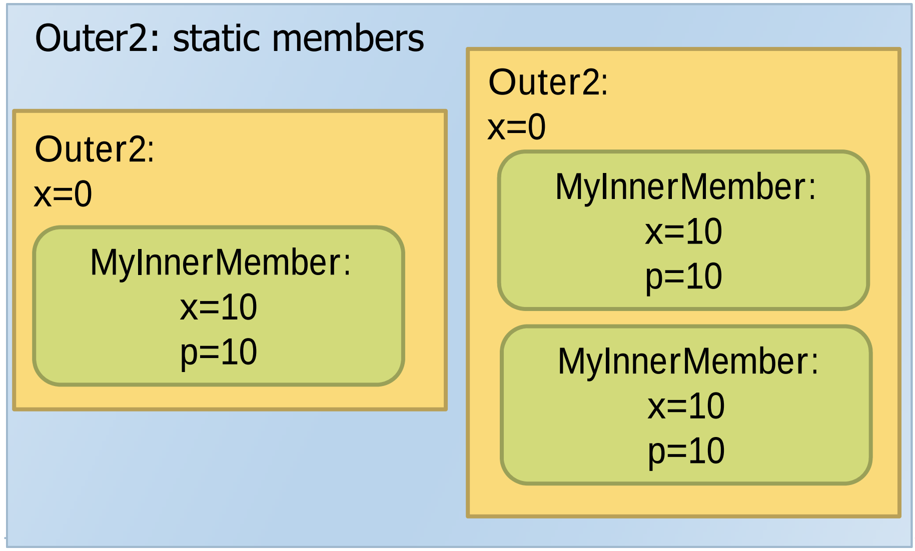
- Every instance of an inner class is linked to the enclosing instance that create it.

```
// within the Outer class itself
MyInnerMember n = new MyInnerMember();
```

```
//from any other classes
Outer2 obj = new Outer2();
Outer2.MyInnerMember m3 = obj.new MyInnerMember();
```

- Note: More than one inner instance can be associated with its enclosing instance.

```
public class Outer2 {
  private int x;
  private static int count;
  ...
  public class MyInnerMember {
    ...

  }
}
```



Outer2: static members

Outer2:
x=0

MyInnerMember:
x=10
p=10

Outer2:
x=0

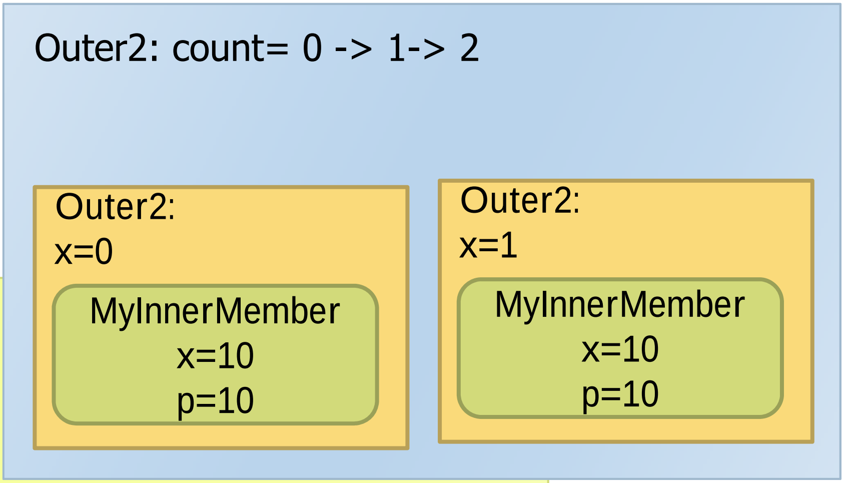
MyInnerMember:
x=10
p=10

MyInnerMember:
x=10
p=10

4.Member Classes
Accessing Methods and Fields

Outer2.java

- The enclosing class (Outer2) has full access to the member class
  - i.e.ALL instance members



Outer2: count= 0 -> 1-> 2

Outer2:
x=0
MyInnerMember
x=10
p=10

Outer2:
x=1
MyInnerMember
x=10
p=10

```
public class Outer2 {
  private int x;
  private static int count;
  public Outer2() {
    x=count++;
    MyInnerMember n = new MyInnerMember();
    System.out.println("n.x=" + n.x + ", n.p" + n.p);
  }
  public class MyInnerMember {
    private int p = 10;
    private int x = 10;
    ...
  }
  public static void main(String[] args) {
    Outer2 m1 = new Outer2();
    Outer2 m2 = new Outer2();
}
```

Create a new instance of the member class

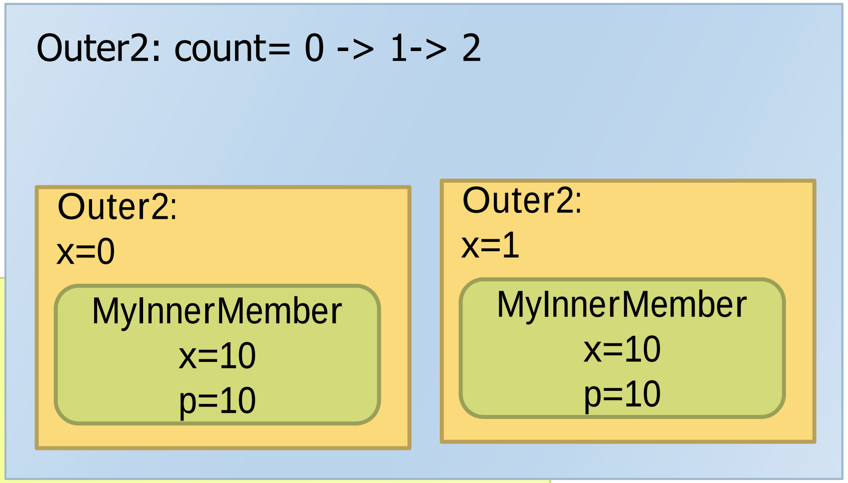
No static variables should be defined inside the member class

```
n.x=10, n.p10
n.x=10, n.p10
```

Note: You can't create inner class objects without first creating an outer class object

4.Member Classes
Accessing Methods and Fields

- The member instance can access to all instance and static **members** of the outer class (even declared as private).



```
public class Outer2 {
  private int x;
  private static int count;
  public Outer2() {
    x=count++;
    MyInnerMember n = new MyInnerMember();
    n.instanceMethod();
  }
  public class MyInnerMember {
    private int p = 10;
    private int x = 10;
  }
  public void instanceMethod() {
      System.out.print("x=" + x + ",p=" + p );
      System.out.println(" x=" + Outer2.this.x + ",count=" + Outer2.count);
    }
...
```

```
x=10,p=10 x=0,count=1
x=10,p=10 x=1,count=2
```

Exercise 1

- What is the output of the following program?

```
class TestMemberOuter1{
 private int data=30;
 class Inner{
  void message(){System.out.println("data is "+data);}
 }
 public static void main(String args[]){
  TestMemberOuter1 obj=new TestMemberOuter1();
  TestMemberOuter1.Inner in=obj.new Inner();
  in.message();
 }
}
```

5.Local Classes

- A local class is a class defined within a method
- A local class exists until end of that method/block only (hidden from everywhere else)
- Use: if a class is needed only inside **one method** to do special work, and need not be visible anywhere else
- Rules
  - Never declared with an access specifier-- scope is always restricted to the block in which they are declared (cannot be declared public, protected, private or static)
  - Cannot include static variables and methods
  - Can access the fields of the containing class and the **local variables** of the method they are declared in. (from JDK 8)

```
public class Outer {
 public void method() {

    class Inner {
        ...
    }
 }
}
```

5.Local Classes
Accessing Methods and Fields

- Can access the fields of the containing class and the local variables of the method they are declared in. (from JDK 8)

```
public class Outer3 {
  private int x=1;
  private static int count;
  public void OuterInstanceMethod(int p) {
    final int q=100;
    int r = 20;
    class MyLocalClass {
      private int x = 10;
      private int y = 10;

      public void instanceMethod() {
        System.out.print("x=" + x + ",y=" + y );
        System.out.println(" x=" + Outer3.this.x + ",count=" + Outer3.count);
        System.out.print("p=" + p + " q=" + q +" r=" + r );
      }
    }
    MyLocalClass obj = new MyLocalClass();
    obj.instanceMethod();
...
```

```
Outer3 m1 = new Outer3();
m1.OuterInstanceMethod(5);
```

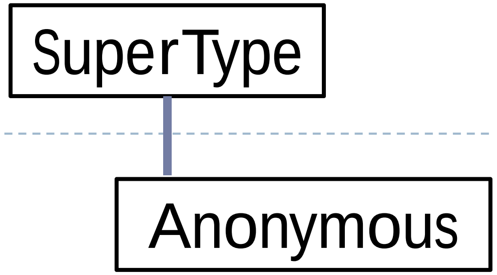
x=10,y=10 x=1,count=0
p=5 q=100 r=20

6.Anonymous Classes

- Definition:
  - a local class that is not given a name, but instead is declared implicitly by creating a variable of it
- An object to be created using an **expression** that combines **object creation** with the **declaration of the class**
  - i.e. declare and instantiate a class at the same time.
- Use them if you need to use a local class only **ONCE**.
- Anonymous classes are commonly used in AWT
- Syntax

```
public class Outer {
    new SuperType(constructor args) {
    // ... class body

    };
}
```

6.Anonymous Classes



SuperType

Anonymous

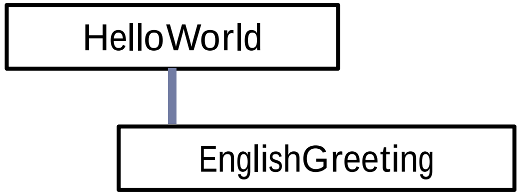
- Syntax

```
public class Outer {
   new SuperType(constructor args) {
   // ... class body

   };
}
```

- SuperType can be an **interface** (that the anonymous class **implements**) or a **class** (that the anonymous class **extends**)
  - The form of the new statement:
    - □ Declares a **new anonymous class** that **extends** a given class or **implements** a given interface,
    - □ **creates** a new instance of that class, and
    - □ **returns** it as the result of the statement
- Note: the class body can define methods but cannot define any constructors

6.Anonymous Classes
Local class Vs Anonymous class

- Remember: Anonymous is a local class that is not given a name!
- Example:
  - HelloWorld interface
  - Case 1: Using a Local class
  - Case 2: Using an anonymous class



Create an instance of an anonymous class (no name) which implements the HelloWorld interface

```
public void sayBye() {
    HelloWorld i  = new HelloWorld() {
       // ... class body

    };
    i.greet();
}
```

Local class

The local class implements the HelloWorld interface

Create an instance of local class

```
public void sayHello() {
   class EnglishGreeting implements HelloWorld {
       ...
   }
   HelloWorld english = new EnglishGreeting();
   english.greet();
}
```

6.Anonymous Classes
Local class Example

- Local inner classes can be declared anywhere a local variable can be declared and have the same Method Scope.
- Local inner classes can only be instantiated from **within** the method they are declared in.
  - When instantiating a local inner class, the instantiation code must come after the local inner class declaration.

```
public void sayHello() {
  class EnglishGreeting implements HelloWorld {
    String name = "world";
    public void greet() {
      System.out.println("Hello " + name);
    }
  }
  HelloWorld english = new EnglishGreeting();
  english.greet();
}
```

```
interface HelloWorld {
  public void greet();
}
```

```
Hello world
```

6.Anonymous Classes
Anonymous class Example

- Anonymous inner classes as the terminology implies have no name.
  - You can't execute the instanceof test against anonymous inner classes or any process that requires the name of the class.
- Anonymous inner classes can be coded anywhere where an expression is legal, so keep the code to a minimum to maintain readability.
- Anonymous inner classes can't implement multiple interfaces.

```
public void sayBye() {
  HelloWorld i = new HelloWorld() {
    String name = "world";
    public void greet() {
      System.out.println("Bye " + name);
    }
  };
  i.greet();
 }
```

```
Bye world
```

6.Anonymous Classes
Multiple Interfaces

- It is simply a less flexible way of creating a local inner class with **one instance**.
- But if you want …
    - a local inner class which implements multiple interfaces or
    - which implements interfaces while extending some class other than Object or
    - which specifies its own constructor …
  - You should create a regular named local inner class.
- there is a trivial workaround
  - Using an interface extending both of them:

```
interface InterfaceB  {
  public void g();
}
interface InterfaceA {
  public void f();
}

interface InterfaceD {}
```

```
new InterfaceD() {
      public void f() { }
      public void g() { }
    };
}
```

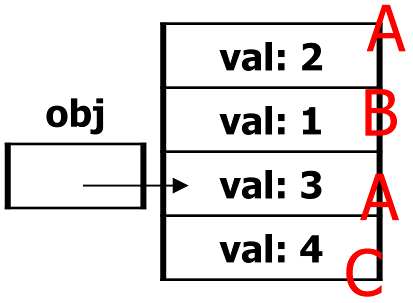
Exercise 2

- What is the output of the following code?

```
B.C obj = new B().new C();
```

```
class A {
    int val;
    A(int v) { val = v; }
}
```

```
class B extends A {
    int val = 1;
    B() { super(2); }

    class C extends A {
        int val = 3;
        C() {
            super(4);
            System.out.println(B.this.val);
            System.out.println(C.this.val);
            System.out.println(super.val);
        }
    }
}
```



Application

- Create an array, fill it with integer values, and then output only values of even indices of the array in ascending order.
- Example:
  - DataStructure class contains:
    - a constructor to create an instance containing an array
  - The DataStructureIterator inner class:
    - implements the Iterator interface.
      - □ Iterators are used to step through a data structure and typically have methods **to test for the last element**, **retrieve** the current element, and **move** to the next element.

Iterator Interface

- An iterator object is a “one shot” object. It is designed to go through all the elements of a Collection once
- Methods:
  - boolean hasNext()
    - //returns true if this iteration has more elements
  - Object next()
    - //returns the next element in this iteration

```
ArrayList<Integer> list;
list = new ArrayList<Integer>();
list.add(3);
list.add(3);
list.add(5);
Iterator<Integer> it = list.iterator();
System.out.print(it.next() + " ");
System.out.print(it.next() + " ");
System.out.print(it.next());
```

```
3 3 5
```

```
while (it.hasNext())
   System.out.print(it.next() + " ");
```

DataStructure

- Problems
  - Need to access private data in DataStructure
- Solution
  - Define DataStructureIterator as inner class

```
class DataStructure implements Iterable<Integer> {
    private final static int SIZE = 15;
    private int[] arrayOfInts = new int[SIZE];
    public DataStructure() {
        for (int i = 0; i < SIZE; i++) {
            arrayOfInts[i] = i;
        }
    }
...
}
```

```
0 2 4 6 8 10 12 14
```

```
DataStructure ds = new DataStructure();
Iterator<Integer> iterator  = ds.iterator();
while (iterator.hasNext())
  System.out.print(iterator.next() + " ");
```

DataStructure & DataStructureIterator

- Example:

```
class DataStructure implements Iterable<Integer> {
    private final static int SIZE = 15;
    private int[] arrayOfInts = new int[SIZE];
    public DataStructure() {
        ...
    }
    public Iterator<Integer> iterator() {
        return new DataStructureIterator();
    }
    public class DataStructureIterator implements Iterator<Integer> {
      private int nextIndex = 0;
      public boolean hasNext() {
        return (nextIndex <= SIZE - 1);
      }
      public Integer next() {
        Integer retValue = Integer.valueOf(arrayOfInts[nextIndex]);
        nextIndex += 2;
        return retValue;
      }
    }
}
```

Anonymous

- Consider the DataStructure class, rewrite the iterator using anonymous

```
class DataStructure implements Iterable<Integer>{
  private final static int SIZE = 15;
  private int[] arrayOfInts = new int[SIZE];
  public DataStructure() {
        ...
  }
  public Iterator<Integer> iterator() {
    return new Iterator() {
      private int nextIndex = 0;
      public boolean hasNext() {
        return (nextIndex <= SIZE - 1);
      }
      public Integer next() {
        Integer retValue = Integer.valueOf(arrayOfInts[nextIndex]);
        nextIndex += 2;
        return retValue;
      }
   };
  }
}
```

Return an instance of a class which implements the Iterator interface

Exercise 3

- Consider the following interface:

```
interface Iterator {
    public boolean hasNext();
    public Object next();
}
```

- And the example program:

```
DataStructure ds = new DataStructure();
Iterator iterator  = ds.iterator();
while (iterator.hasNext())
  System.out.print(iterator.next() + " ");
```

- Implement an OddNumberIterator so that it prints elements that have an odd index value.

```
import java.util.*;
class DataStructure {
  private final static int SIZE = 15;
  private int[] arrayOfInts = new int[SIZE];
  public DataStructure() {
    for (int i = 0; i < SIZE; i++) arrayOfInts[i] = i;
  }
  public Iterator iterator() {return new OddNumberIterator();  }
  public class        ...  }
```

Summary

Type of Nested Class	**Applies To**	**Declared**	**Can be Used**
Static Member	Classes and interfaces	Inside a class as static	By any class
Member (non-static)	Classes	Inside a class (non-static)	Within the member class
Local (named)	Classes	Inside a method	Within the method
Anonymous (local unnamed)	Classes	Inside a method with no name	Within the method

Summary

Type of Nested Class	**Structure**	**Variable Visibility**
Static Member	may have instance or static variables/methods	access only static outer variables and methods
Member (non-static)	no static methods or variables allowed	access outer instance or static variables/methods
Local (named)	no static methods or variables allowed	access – outer instance or static variables/methods – local final variables
Anonymous (local unnamed)	no static methods or variables allowed	access – outer instance or static variables/methods – local final variables