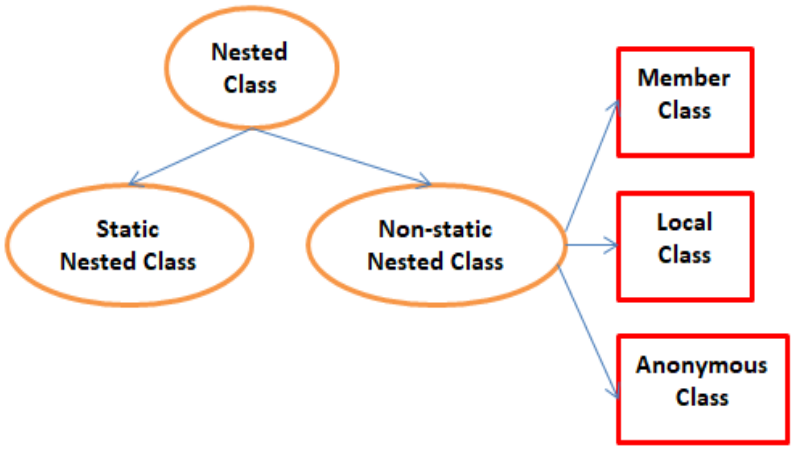
Nested Classes in Java

- Java allows us to define classes inside other classes.
- The class written within a class is called the nested class, and the class that holds the inner class is called the outer class.
- The scope of the nested class is bounded by the scope of its enclosing class.
- Nested classes enable us to logically group classes that are only used in one place, write more readable and maintainable code and **increase encapsulation**.
- If we want to create a class which is to be used only by the enclosing class, then it is not necessary to create a separate file for that. Instead, you can add it as "Inner Class"
- One important thing to notice about an Inner class is that it can be created only within the scope of Outer class. Java compiler generates an error if any code outside Outer class attempts to instantiate Inner class **directly**.
- A nested class is a **member** of its enclosing class. As a member of the OuterClass, a nested class can be declared private, public, protected, or *package private*. (Recall that outer classes can only be declared public or *package private*.)
- Based on implementation nested class can be divided Into following types :




```
class OuterClass {
   ...
   class NestedClass {
      ...     }
}
class OuterClass {
   ...
   static class StaticNestedClass {
      ...
   }
   class InnerClass {
      ...     }
}
```

- Non-static nested classes (inner classes) have access to other members of the enclosing class, even if they are declared **private**.
- Static nested classes have **no access to the instance variables** (public or private) of the surrounding class. This does not mean that static nested classes have no access to instance variables at all. It means that static nested classes have no access to instance variables of the enclosing class "for free", the way the non-static nested classes do.

Advantage of java inner classes

- Nested classes provide a way of logically grouping classes that are only used in one place. It can access all the members (data members and methods) of outer class including private.
- **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.

- **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.
- Inner classes are a security mechanism in Java. We know a class cannot be associated with the access modifier **private**, but if we have the class as a member of other class, then the inner class can be made private. And this is also used to access the private members of a class.

- **Code Optimization**: It requires less code to write.

Examples –

Inner class is best useful when its functionality is tied to its outer class.

- You have a class address whose object should have a reference to city and by design that's the only use of city you have in your application. Making Address and City as outer and inner class exposes City to any of the Class and it will make sure that it is accessed using object of Address.

- You have a Robot class. Now this robot will have some AI brain class, Arm class, legs class etc. Then, When you instantiate Robot class, its AI brain should also be instantiated and likewise others. So, it is better to make all other classes as the inner classes of Robot Class.

1. Static Nested Class

- It is nested class with static modifier.
- Static nested classes can access only static members of its outer class i.e it cannot refer to non-static members of its enclosing class directly. Because of this restriction, static nested class is rarely used.
- As with static members, these belong to their enclosing class, and not to an instance of the class. So you don't need instance of parent class to access the static nested class.
- They can have all types of access modifiers in their declaration.
- They can define both static and non-static members.

```
public class Enclosing {

   private static int x = 1;
   public static class StaticNested {
      private void run() {    // method implementation    }
   }
}
```

```
public void test() {
      Enclosing.StaticNested nested =
            new Enclosing.StaticNested();
      nested.run();
   }
```

Note :When referring we use the outer class name not its instance as it is static inner class.

2. Non-static Nested class

- Non-static Nested class is the most important type of nested class. It is also known as **Inner** class.
- It has access to all variables and methods of Outer class including its private data members and methods and may refer to them directly. But the reverse is not true, that is, Outer class cannot directly access members of Inner class.
- Inner class can be declared private, public, protected, or with default access whereas an Outer class can have only public or default access.
- They can only define non-static members.
- A non-static Nested class that is created outside a method is called **Member inner class**.
- A non-static Nested class that is created inside a method is called **Local inner class**. If you want to invoke the methods of local inner class, you must instantiate this class inside the method. We cannot use private, public or protected access modifiers with local inner class. Only abstract and final modifiers are allowed.

a) Member inner class

```
class Outer
{
    public void display()
    {
        Inner in=new Inner();       in.show();
    }
    class Inner
    {
        public void show()
        {
           System.out.println("Inside inner");
        }
    }
}
```

```
class Test
{
 public static void main(String[] args)
 {
  Outer ot=new Outer();
  ot.display();
 }
}
```

Output :
Inside inner

Note : Here inner class method is invoked inside an outer class method. So no special syntax needed.

```
class Outer
{
  // Simple nested inner class
  class Inner  {
      public void show()
      {
        System.out.println( “nested class method");
      }
  }
}
```

```
class Main {
   public static void main(String[] args)
   {
       Outer.Inner in = new Outer().new Inner();
       in.show();
   }
}
```

Note : Here we instantiate the inner class and use its method. So we need to an instance of outer class to create a inner class instance.

b) Local Class

Local classes are a special type of inner classes – in which **the class is defined inside a method** or scope block. It is also known as **method local class**.

- They cannot have access modifiers in their declaration
- They have access to both static and non-static members in the enclosing context
- They can only define instance members

```
class Outer
{
   int count;
   public void display()
   {
      for(int i=0;i<5;i++)
      {
         class Inner     // Inner class inside loop
         {
            public void show()
            {
                  System.out.println("Inside inner"
                     + (count++));
            }
         }
         Inner in=new Inner();      in.show();
      }
   }
}
class Outer
{
    void outerMethod()
    {
        System.out.println("inside outerMethod");
        class Inner
        {
            void innerMethod()
            {
                System.out.println("inside
                                    innerMethod");
            }
        }
        Inner y = new Inner();
        y.innerMethod();
    }
}
```

```
class Test
{
   public static void main(String[] args)    {
      Outer ot=new Outer();
      ot.display();
  }
}

class MethodDemo
{
    public static void main(String[] args)
    {
        Outer x = new Outer();     x.outerMethod();
    }
}
```

Output
Inside inner 0
Inside inner 1
Inside inner 2
Inside inner 3
Inside inner 4

Example of Inner class instantiated outside Outer class

```
class Outer
{
    int count;
    public void display()
    {
        Inner in=new Inner();      in.show();
    }
    class Inner   {
        public void show()       {
            System.out.println("Inside inner " +
                    (++count));    }
    }
}
class Test   {
   public static void main(String[] args)    {
        Outer ot=new Outer();
        Outer.Inner in= ot.new Inner();     in.show();
   }
}
```

Output : Inside inner 1

Method Local class cannot access a local variable from outer class until it is defined as **final**. In the example below, the compiler generates an error.

```
class Outer
{
    void outerMethod()
    {
        int x = 98;
        System.out.println("inside outerMethod");
        class Inner
        {
            void innerMethod()    {
                System.out.println("x= "+x);
            }
        }
        Inner y = new Inner();      y.innerMethod();
    }
}
class MethodLocalVariableDemo
{
    public static void main(String[] args)    {
        Outer x=new Outer();       x.outerMethod();
    }
 }
```

Output

local variable x is accessed from within inner class; needs to be declared final

But below code doesn't generate any error :

```
class Outer
{
    void outerMethod()
    {
        final int x=98;
        System.out.println("inside outerMethod");
        class Inner
        {
            void innerMethod()    {
                System.out.println("x = "+x);
            }
        }
        Inner y = new Inner();      y.innerMethod();
    }
}
class MethodLocalVariableDemo
{
    public static void main(String[] args)
    {
        Outer x = new Outer();
        x.outerMethod();
    }
}
```

Why only final local variable can be accessed?

- Java doesn't support **closures**, i.e. local variable (variables of method) can't be accessed outside the method, but fields of class can be accessed from outside the class. We know that Methods and local variables live on the **stack**. So, local variables are lost as soon as the method ends.

- Where do object of local inner class live in java?
  As object of local inner class live on the **heap**, objects may be **alive** even after method ends in which local inner class has been defined.

- A local class can only access local variables that are declared final (not change after initialization) in java. When an inner class access final local variables compiler create a **synthetic field** inside the inner class (Synthetic fields are created by compiler and they actually doesn't exist in source code) and also copy that variable into the **heap**. So, these synthetic fields can be accessed inside local inner class even when execution of method is over in java.

c) Anonymous Inner Class

- It is an inner class without a name and for which only a single object is created.
- It is useful when making an instance of an object with certain "extras" such as overriding methods of a class or interface, without creating separate subclass of that class.
- It is useful in writing implementation classes for listener interfaces in graphics programming.
- They cannot have access modifiers in their declaration
- They have access to both static and non-static members in the enclosing context
- They can only define instance members
- They're the only type of nested classes that cannot define constructors or extend/implement other classes or interfaces
- An anonymous class has access to the members of its enclosing class.
- An anonymous class cannot access local variables in its enclosing scope that are not declared as **final**.
- We can use any constructor while creating anonymous class. Notice the constructor argument is being used too.
- Like a nested class, a declaration of a type (such as a variable) in an anonymous class shadows any other declarations in the enclosing scope that have the same name.

Anonymous inner class are mainly created in two ways:

- By extending Class (may be abstract or concrete)
- By implementing Interface

Syntax: The syntax of an anonymous class expression is like the invocation of a constructor, except that there is a class definition contained in a block of code.

```
Class Outer  {
   // Test can be interface,abstract/
      concrete class
   Test tIns = new Test()
   {
       // data members and methods
       public void test_method()
       {
           ........
           ........
       }
    };
}
```

- The anonymous class creation lines look like we are creating an instance of the Test class called tIns. But ***what's actually happening in that code is that an instance of an anonymous class is being created.***
- An anonymous inner class is a subclass.
- This certainly does not look like we are creating a normal instance of a class – because you don't normally see methods being defined at the same time that an instance of a class is created.
- We are creating an instance of a subclass of the Test class. The instance (**tIns**) is actually an instance of an anonymous subclass of the Test class.
- The class that we have created clearly has no name! We jump straight to creating an instance of the class - all we have is a reference variable (**tIns**) for the anonymous inner class.

To understand anonymous inner class, let us take a simple program. Here we have a FlyingMachine class and we want to extend this class and override one of the methods in this class. Without creating the subclass separately we can use the anonymous inner class syntax as shown in the example.

FlyingMachine.java

```
package javaapp2;
import static java.lang.System.out;

//  Base class for different type of flying machine
public class FlyingMachine {
   private int noEngine;
   private int noPassenger;
   public void fly() {
      out.println("No implementation");
   }
   public void setEngine(int ne)  {
      noEngine = ne;
    }
   public int  getEngine()   {   return noEngine;  }
   public void setPassenger(int np)  {
         noPassenger = np;
   }
   public int  getPassenger()   {
         return noPassenger;
   }
   public void bombardment(String fmType ) {
       out.println(fmType + " may not
              bombard missile");
   }
}
```

OuterFly.java

```
package javaapp2;
import static java.lang.System.out;

//   Outer Class
public class OuterFly {
   /*  This creates an anonymous inner class: */
   FlyingMachine flyIns = new FlyingMachine() {
      @Override
      public void bombardment(String fmType )
      {
           out.println(fmType + " will bombard 5
                  missile at a time");
      }
      //  A new method is defined
      public void newMthod(String fmType )
      {
         out.println(fmType + " will not support
              new method");
      }
   };
}
```

- Note : We can also define new methods but this method gives compilation error.

```
public class JavaApp2 {

    public static void main(String[] args) {

      //  Anonimous Inner Class
      OuterFly ofly = new OuterFly();
      ofly.flyIns.setEngine(4);
      ofly.flyIns.setPassenger(2);
      ofly.flyIns.bombardment("Fighter Plane");
      ofly.flyIns.newMethod("Fighter Plane");
   }
}
```

Output

```
Fighter Plane will bombard 5 missile at a time
```

- The **newMethod**() was defined inside the anonymous inner class and not in FlyingMachine. As the instance **flyIns** is of type FlyingMachine it has no idea what the newMethod() is. So this line will give compilation error as shown below.
- We cannot overload a method in inner class too.

```
public static void main(String[] args) {

    //  Anonimous Inner Class
    OuterFly ofly = new
    ofly.flyIns.setEngi
    ofly.flyIns.setPass
    ofly.flyIns.bombard
    ofly.flyIns.newMethod("Fighter Plane");
```

```
cannot find symbol
  symbol:   method newMethod(String)
  location: variable flyIns of type FlyingMachine
----
(Alt-Enter shows hints)
```

Anonymous inner classes and polymorphism

- When using anonymous inner classes, polymorphism is actually at work as well.
- Note that we are referencing the object **flyIns** actually by a superclass reference type FlyingMachine that refers to a subclass object.
- Using the anonymous inner class reference variable type (**flyIns** in our example) you can only call methods that are defined inside the type (the class) of the reference variable. Using our example, this means that with **flyIns** we can only call methods that are defined inside the FlyingMachine class.
- In the code above, the call ofly.flyIns.newMethod() is perfectly legal, but it results in a compiler error because the "newMethod()" method is not defined inside the FlyingMachine class, and our object "**flyIns**" is of type FlyingMachine .

Difference between Normal class and Anonymous Inner class:

- A normal class can implement any number of interfaces but anonymous inner class can implement only one interface at a time.
- A normal class can extend a class and implement any number of interface simultaneously. But anonymous Inner class can extend a class or can implement an interface but not both at a time.
- For regular/normal class, we can write any number of constructors but we cannot write any constructor for anonymous Inner class because anonymous class does not have any name and while defining constructor class name and constructor name must be same.

Types of anonymous inner class :

Based on declaration and behavior, there are 3 variations of anonymous Inner classes:

- Anonymous Inner class that extends a class : We can have an anonymous inner class that extends a class. For example, we know that we can create a thread by extending a Thread class. Suppose we need an immediate thread but we don't want to create a class that extend Thread class all the time. By the help of this type of Anonymous Inner class we can define a ready thread as follows:

// Java program to illustrate creating an immediate thread using Anonymous Inner class
that extends a Class

```
class MyThread
{
   public static void main(String[] args)
   {
      //Here we are using Anonymous Inner class that extends a class i.e. Here a Thread class
      Thread t = new Thread()
      {
         public void run()
         {
            System.out.println("Child Thread");
         }
      };
      t.start();
      System.out.println("Main Thread");
   }
}
```

Output:
Main Thread
Child Thread
OR
Child Thread
Main Thread

- Anonymous Inner class that implements a interface : We can also have an anonymous inner class that implements an interface. For example, we also know that by implementing Runnable interface we can create a Thread. Here we use anonymous Inner class that implements an interface.

```
// Java program to illustrate defining a thread Using Anonymous Inner class that
   implements an interface

class MyThread
{
   public static void main(String[] args)
   {
      //  Here we are using Anonymous Inner class that implements an interface
          i.e. Here Runnable interface
      Runnable r = new Runnable()
      {
         public void run()
         {
            System.out.println("Child Thread");
         }
      };
      Thread t = new Thread(r);
      t.start();
      System.out.println("Main Thread");
   }
}
```

Output:
Main Thread
Child Thread
OR
Child Thread
Main Thread

- Anonymous Inner class that defines inside method/constructor argument : Anonymous inner classes in method/constructor arguments are often used in graphical user interface (GUI)

applications. To get you familiar with syntax let's have a look on the following program that creates a thread using this type of Anonymous Inner class :

```
//Java program to illustrate defining a thread Using Anonymous Inner class that define inside argument

class MyThread
{
   public static void main(String[] args)
   {
      //  Here we are using Anonymous Inner class that define inside argument,
          here constructor argument
      Thread t = new Thread(new Runnable()
      {
         public void run()
         {
            System.out.println("Child Thread");
         }
      });
      t.start();
      System.out.println("Main Thread");
   }
}
```

Output:
Main Thread
Child Thread
OR
Child Thread
Main Thread

```
class Parent
{
      public void product()  {   System.out.println("production starts....."); }
}
interface plant  {
      public void grow();
}
class Parentproduct
{
      public static void main(String[] args)
      {
          Parent p1=new Parent() {
               public void product()      {      System.out.println("production is in the process....");     }
          };
          p1.product();
          plant p2=new plant() {
                @Override
                public void grow() {       System.out.println("plant starts growing");           }
          };
          p2.grow();
      }
```

}

Shadowing

Shadowing refers to the practice in Java programming of using two or more types (such as a member variable or a parameter name) with the same name within scopes that overlap. When you do that, the variable with the higher-level scope is hidden because the variable with lower-level scope overrides it. The higher-level variable is then "shadowed."

In this case, the ***this*** keyword refers to the instances of the nested class and the members of the outer class can be referred to using the name of the outer class.

You can access a shadowed class or instance variable by fully qualifying it — that is, by providing the name of the class that contains it.

The following example, ShadowTest, demonstrates this:

Three variables using same name x :
- member variable of the class ShadowTest
- member variable of the inner class FirstLevel
- parameter in methodInFirstLevel.

```
public class ShadowTest {

   public int x = 0;
   class FirstLevel {
      public int x = 1;

      void methodInFirstLevel(int x) {
         System.out.println("x = " + x);
         System.out.println("this.x = " + this.x);
         System.out.println("ShadowTest.this.x = " + ShadowTest.this.x);
      }
   }

   public static void main(String... args) {

      ShadowTest st = new ShadowTest();
      ShadowTest.FirstLevel fl = st.new FirstLevel();
      fl.methodInFirstLevel(23);
   }
}
```

Refers to parameter x

Refers to member of FirstLevel class

Refers member of ShadowTest

First create an instance of outer class

Call the method of inner class, by using inner class instance

Output :

```
x = 23
this.x = 1
ShadowTest.this.x = 0
```

The declaration of the members of an inner class shadow those of the enclosing class if they have the same name. Let's see a quick example:

```
public class NewOuter {

   int a = 1;
   static int b = 2;

   public class InnerClass {
      int a = 3;
      static final int b = 4;

      public void run() {
         System.out.println("a = " + a);
         System.out.println("b = " + b);
         System.out.println("NewOuterTest.this.a = " + NewOuter.this.a);
         System.out.println("NewOuterTest.b = " + NewOuter.b);
         System.out.println("NewOuterTest.this.b = " + NewOuter.this.b);
      }
   }

   @Test
   public void test() {
      NewOuter outer = new NewOuter();
      NewOuter.InnerClass inner = outer.new InnerClass();
      inner.run();

   }
}
```

Another Example :

Here is a java class showing how to define java inner class, static nested class, local inner class and anonymous inner class.

OuterClass.java

```
package com.journaldev.nested;

import java.io.File;    import java.io.FilenameFilter;

public class OuterClass {

   private static String name = "OuterClass";
   private int i;        protected int j;              int k;                public int l;

   //OuterClass constructor
   public OuterClass(int i, int j, int k, int l) {   this.i = i;    this.j = j;   this.k = k;      this.l = l;    }
```

```
public int getI() {   return this.i;    }

//static nested class, can access OuterClass static variables/methods
static class StaticNestedClass {
   private int a;        protected int b;        int c;                  public int d;

   public int getA() {  return this.a;      }

   public String getName() {   return name;      }
}

//inner class, non static and can access all the variables/methods of outer class
class InnerClass {
   private int w;        protected int x;
   int y;                public int z;

   public int getW() {    return this.w;       }

   public void setValues() {    this.w = i;      this.x = j;       this.y = k;       this.z = l;       }

   @Override
   public String toString() {     return "w=" + w + ":x=" + x + ":y=" + y + ":z=" + z;       }

   public String getName() {
      return name;
   }
}
//local inner class ( defined inside the method print )
public void print(String initial) {
   //local inner class inside the method
   class Logger {
      String name;
      public Logger(String name) {     this.name = name;       }

      public void log(String str) {
         System.out.println(this.name + ": " + str);
      }
   }
   Logger logger = new Logger(initial);
   logger.log(name);              logger.log("" + this.i);
   logger.log("" + this.j);       logger.log("" + this.k);       logger.log("" + this.l);
}
//anonymous inner class
```

Declared as static in outer class

Accessing all fields from outer class (k and l are outer class fields)

Accessing even the static field from outer class

Static field of outer class

All other fields of outer class

```
    public String[] getFilesInDir(String dir, final String ext) {
        File file = new File(dir);

        //anonymous inner class implementing  FilenameFilter interface

        String[] filesList = file.list(new FilenameFilter() {

            @Override
            public boolean accept(File dir, String name) {
                return name.endsWith(ext);
            }
        });
        return filesList;
    }
}
```

- This Anonymous inner class extends a class or implement an interface.
- Since an anonymous class has no name, it is not possible to define a constructor.

Here is the test program showing how to instantiate and use inner class in java.

InnerClassTest.java

```
package com.journaldev.nested;

import java.util.Arrays;
//  nested classes can be used in import for easy  instantiation
import com.journaldev.nested.OuterClass.InnerClass;
import com.journaldev.nested.OuterClass.StaticNestedClass;

public class InnerClassTest {

    public static void main(String[] args) {
        OuterClass outer = new OuterClass(1,2,3,4);

        //  static nested classes example
        StaticNestedClass staticNestedClass =
                                    new StaticNestedClass();
        StaticNestedClass staticNestedClass1 =
                                    new StaticNestedClass();

        System.out.println(staticNestedClass.getName());
        staticNestedClass.d=10;
        System.out.println(staticNestedClass.d);
        System.out.println(staticNestedClass1.d);

        //  inner class example
```

Here is the output of java inner class example program.

```
OuterClass
10
0
OuterClass
w=0:x=0:y=0:z=0
w=1:x=2:y=3:z=4
Outer: OuterClass
Outer: 1
Outer: 2
Outer: 3
Outer: 4
[NestedClassTest.java, OuterClass.java]
[NestedClassTest.class,
OuterClass$1.class,
OuterClass$1Logger.class,
OuterClass$InnerClass.class,
OuterClass$StaticNestedClass.class,
OuterClass.class]
```

```
        InnerClass innerClass = outer.new InnerClass();
        System.out.println(innerClass.getName());
        System.out.println(innerClass);
        innerClass.setValues();
        System.out.println(innerClass);

        //calling method using local inner class
        outer.print("Outer");

        //calling method using anonymous inner class
        System.out.println(Arrays.toString(outer.getFilesInDir("src/com/journaldev/nested", ".java")));

        System.out.println(Arrays.toString(outer.getFilesInDir("bin/com/journaldev/nested", ".class")));
    }

}
```

Notice that when OuterClass is compiled, separate class files are created for inner class, local inner class and static nested class.

More Examples

Inner Class

```
public class Parcel1 {
    class Contents {
        private int i = 11;
        public int value()    { return i; }
    }
    class Destination {
        private String label;
        Destination(String whereTo) {     label = whereTo;     }
        String readLabel()   { return label; }
    }
    // Using inner classes looks just like using any other class, within Parcel1:
    public void ship(String dest) {
        Contents c = new Contents();          Destination d = new Destination(dest);
        System.out.println(d.readLabel());
     }
     public static void main(String[] args) {
         Parcel1 p = new Parcel1();
         p.ship("Tanzania");
    }
}
```

The inner classes, when used inside ship( ), look just like the use of any other classes. Here, the only practical difference is that the names are nested within Parcel1.

```
public class Parcel2 {
      class Contents {
            private int i = 11;
            public int value()   { return i;   }
      }
      class Destination {
            private String label;
            Destination(String whereTo) {     label = whereTo;    }
            String readLabel()  { return label; }
      }
      public Destination to(String s) {
            return new Destination(s);
      }
      public Contents cont() {
            return new Contents();
      }
      public void ship(String dest) {
            Contents c = cont();
            Destination d = to(dest);
            System.out.println(d.readLabel());
      }
      public static void main(String[] args) {
            Parcel2 p = new Parcel2();
            p.ship("Tanzania");
            Parcel2 q = new Parcel2();

            // Defining references to inner classes:
            Parcel2.Contents c = q.cont();
            Parcel2.Destination d = q.to("Borneo");
      }
}
```

If you want to make an object of the inner class anywhere except from within a non-static method of the outer class, you must specify the type of that object as **OuterClassName.InnerClassName**, as seen in main( ).

Anonymous inner classes

```
// A method that returns an anonymous inner class.

public class Parcel6 {
   public Contents cont() {    return new Contents() {
         private int i = 11;
         public int value() { return i; }
   }; // Semicolon required in this case
 }
 public static void main(String[] args) {
```

The cont( ) method combines the creation of the return value with the definition of the class that represents that return value! In addition, the class is anonymous; it has no name.

Create an object of an anonymous class that'is inherited from Contents." The reference returned by the new expression is automatically upcast to a Contents reference.

```
    Parcel6 p = new Parcel6();
    Contents c = p.cont();
  }
}
```

The anonymous inner-class syntax is a shorthand for:

```
class MyContents implements Contents {
  private int i = 11;
  public int value() { return i; }
}
return new MyContents();
```

In the anonymous inner class, Contents is created by using a default constructor. The following code shows what to do if your base class needs a constructor with an argument:

```
// An anonymous inner class that calls the base-class constructor.

public class Parcel7 {
  public Wrapping wrap(int x) {
    // Base constructor call:
    return new Wrapping(x) {  // Pass constructor argument.
      public int value() {
        return super.value() * 47;
      }
    }; // Semicolon required
  }
  public static void main(String[] args) {
    Parcel7 p = new Parcel7();
    Wrapping w = p.wrap(10);
  }
}
```

That is, you simply pass the appropriate argument to the base-class constructor, seen here as the **x** passed in **new Wrapping(x)**.

You can also perform initialization when you define fields in an anonymous class:

```
// An anonymous inner class that performs nitialization.

public class Parcel8 {
  // Argument must be final to use inside  anonymous inner class:
  public Destination dest(final String dest) {
    return new Destination() {
          private String label = dest;
          public String readLabel() { return label; }
    };
  }
  public static void main(String[] args) {
```

If you are defining an anonymous inner class and want to use an object that is defined outside the anonymous inner class, the compiler requires that the argument reference be final, like the argument to dest( ). If you forget, you will get a compile-time error message.

```
    Parcel8 p = new Parcel8();
    Destination d = p.dest("Tanzania");
  }
}
```

As long as you are simply assigning a field, the approach in this example is fine. But what if you need to perform some constructor-like activity? You can't have a named constructor in an anonymous class (since there is no name!), but with *instance initialization*, you can, in effect, create a constructor for an anonymous inner class, like this:

```
// Creating a constructor for an anonymous inner class.
import com.bruceeckel.simpletest.*;

abstract class Base {
  public Base(int i) {   System.out.println("Base constructor, i = " + i);    }
  public abstract void f();
}

public class AnonymousConstructor {
  private static Test monitor = new Test();
  public static Base getBase(int i) {
    return new Base(i) {
      {
        System.out.println("Inside instance initializer");
      }
      public void f() {    System.out.println("In anonymous f()");     }
    };
  }
  public static void main(String[] args) {
    Base base = getBase(47);
    base.f();
    monitor.expect(new String[] {
      "Base constructor, i = 47",
      "Inside instance initializer",
      "In anonymous f()"
    });
  }
}
```

In this case, the variable i did not have to be final. While i is passed to the base constructor of the anonymous class, it is never directly used inside the anonymous class. .

Here is an example with instance initialization. Note that the arguments to **dest( )** must be final since they are used within the anonymous class:

```
// Using "instance initialization" to perform construction on an anonymous inner class.
public class Parcel9 {
  private static Test monitor = new Test();
  public Destination
```

```
  dest(final String dest, final float price) {
    return new Destination() {
      private int cost;
      // Instance initialization for each object:
      {
        cost = Math.round(price);
        if(cost > 100)         System.out.println("Over budget!");
      }
      private String label = dest;
      public String readLabel() { return label; }
    };
  }
  public static void main(String[] args) {
    Parcel9 p = new Parcel9();
    Destination d = p.dest("Tanzania", 101.395F);
    monitor.expect(new String[] {    "Over budget!"    });
  }
}
```

Inside the instance initializer you can see code that could not be executed as part of a field initializer (that is, the **if** statement).

So in effect, an instance initializer is the constructor for an anonymous inner class. Of course, it is limited; you cannot overload instance initializers, so you can have only one of these constructors. .