Table of Contents

1 Introduction

A class that is declared within another type declaration is called a nested class. Similarly,an interface or an enum type that is declared within another type declaration is called a nested interface or a nested enum type, respectively

Below are four categories of nested classes, defined by the context these nested types are declared in:

- static member classes
- non-static member classes
- local classes
- anonymous classes

Below is nested enum types, defined by the context these nested

- static member enums

Below is nested interfaces defined by the context these nested

- static member interfaces

They differ from non-inner classes in one important aspect: that an instance of an inner class may be associated with an instance of the enclosing class

The instance of the enclosing class is called the immediately enclosing instance.

A static member class, enum, or interface

> can be declared either at the top-level, or in a nested static type.
> A static class can be instantiated like any ordinary top-level class, using its full name. No enclosing instance is required to instantiate a static member class

Non-static member classes

> are defined as instance members of other classes, just as
> fields and instance methods are defined in a class, An instance of a non-static memberclass always has an enclosing instance associated with it

Local classes can be defined in the

> context of a block as in a method body
> or a local block

anonymous classes can be defined as expressions and instantiated on the fly. An instance of a local (or an anonymous) class has an enclosing instance associated with it

A nested type cannot have the same name as any of its enclosing types.

Below table columns explain this -

- The Declaration Context column lists the lexical context in which a type can be declared.
- The Accessibility Modifiers column indicates what accessibility can be specified for the type.
- The Enclosing Instance column specifies whether an enclosing instance is associated with an instance of the type.
- The Direct Access to Enclosing Context column lists what is directly accessible in the enclosing context from within the type.
- The Declarations in Type Body column refers to what can be declared in the body of the type.

```
class TLC { // (1) Top level class

        // static inner class/interface/enum

        static class SMC {/*...*/} // (2) Static member class

        interface SMI {/*...*/} // (3) Static member interface

        enum SME {/*...*/} // (9) Static member enum


        // Non-static inner class

        class NSMC {/*...*/} // (4) Non-static member (inner) class


        // Local classes

        void nsm() {
                class NSLC {/*...*/} // (5) Local (inner) class in non-static context
        }
        static void sm() {
                class SLC {/*...*/} // (6) Local (inner) class in static context
        }

        // Anonymous classes

        SMC nsf = new SMC() { // (7) Anonymous (inner) class in non-static context
                /*...*/
        };
        static SMI sf = new SMI() { // (8) Anonymous (inner) class in static context
                /*...*/
        };
}
```

Type	**Declaration Context**	**Accessibility Modifiers**	**Enclosing Instance**	**Direct Access to Enclosing Context**	**Declarations in Type Body**
Top-level Class, Enum, or Interface	Package	public or default	No	N/A	All that are valid in a class, enum, or interface body, respectively
Static Member Class, Enum, or Interface	As member of a toplevel type or a nested static type	All	No	Static members in enclosing context	All that are valid in a class, enum, or interface body, respectively
Non-static Member Class	As non-static member of enclosing type	All	Yes	All members in enclosing context	Only non-static declarations + final static fields
Local Class	In block with non-static context	None	Yes	All members in enclosing context + final local variables	Only non-static declarations + final static fields
	In block with static context	None	No	Static members in enclosing context + final local variables	Only non-static declarations + final static fields
Anonymous Class	As expression in nonstatic context	None	Yes	All members in enclosing context + final local variables	Only non-static declarations + final static fields
	As expression in static context	None	No	Static members in enclosing context + final local variables	Only non-static declarations + final static fields

2 Static Member Types

A static member class, enum type, or interface comprises the same declarations as those allowed in an ordinary top-level class, enum type, or interface, respectively.

A static member class must be declared explicitly with the keyword static, as a static member of an enclosing type.

Nested interfaces are considered implicitly static, the keyword static can, therefore, be omitted.

Nested enum types are treated analogously to nested interface in this regard: they are static members.

The accessibility modifiers allowed for members in an enclosing type declaration can naturally be used for nested types.

```
package smc;

public class ListPool { // (1) Top-level class

        public static class MyLinkedList { // (2) Static member class

                private interface ILink { } // (3) Static member interface

                public static class BiNode implements IBiLink { } // (4) Static member class

        }

        interface IBiLink extends MyLinkedList.ILink { } // (5) Static member interface

}

//Filename: MyBiLinkedList.java

package smc;

public class MyBiLinkedList implements ListPool.IBiLink { // (6)

        ListPool.MyLinkedList.BiNode objRef1 = new ListPool.MyLinkedList.BiNode(); // (7)

        //ListPool.MyLinkedList.ILink ref; // (8) Compile-time error!

}
```

- Static variables and methods belong to a class, and not to instances of the class. The same is true for static member classes and interfaces.
- Within the scope of its top-level class or interface, a member class or interface can be referenced regardless of its accessibility modifier and lexical nesting,
- Its accessibility modifier (and that of the types making up its full name) comes into play when it is referenced by an external client.
- Above Example will not compile because the member interface ListPool.MyLinkedList.ILink **has private** accessibility.
- A static member class can be instantiated without any reference to any instance of the enclosing context, as is the case for instantiating top-level classes

- There is seldom any reason to import nested types from packages. It would undermine the encapsulation achieved by such types.
- However, a compilation unit can use the import facility to provide a shortcut for the names of member classes and interfaces. Note that type import and static import of nested static types is equivalent: in both cases, a type name is imported.

```
//Filename: Client1.java

import smc.ListPool.MyLinkedList; // (1) Type import

public class Client1 {

    MyLinkedList.BiNode objRef1 = new MyLinkedList.BiNode();// (2)

}

//Filename: Client2.java

import static smc.ListPool.MyLinkedList.BiNode; // (3) Static import

public class Client2 {

    BiNode objRef2 = new BiNode(); // (4)

}

class BiListPool implements smc.ListPool.IBiLink { } // (5) Not accessible because of package level acceisbility!
```

2.1 Accessing Members in Enclosing Context

Static code does not have a *"this"* reference and can, therefore, only directly access other members that are declared static within the same class. **But not instance members**, in its enclosing context.

Note that a static member class **can define both static and instance members**, like any other top-level class. **However, its code can only directly access static members in its enclosing context.**

A static member class, being a member of the enclosing class or interface, can have any accessibility (public, protected, package/default, private), like any other member of a class.

```
//Filename:  ListPool.java
public class ListPool { // Top-level class

        public void messageInListPool() { // Instance method

                System.out.println("This is a ListPool object.");

        }

        private static class MyLinkedList { // (1) Static class

                private static int maxNumOfLists = 100; // Static variable
                private int currentNumOfLists; // Instance variable
                public static void messageInLinkedList() { // Static method
                    System.out.println("This is MyLinkedList class.");
                }
                interface ILink { int MAX_NUM_OF_NODES = 2000; } // (2) Static interface
                protected static class Node implements ILink { // (3) Static class
                    private int max = MAX_NUM_OF_NODES; // (4) Instance variable
                    public void messageInNode() { // Instance method
                        // int currentLists = currentNumOfLists; // (5) Not OK.
                        int maxLists = maxNumOfLists;
                        int maxNodes = max;
                        // messageInListPool(); // (6) Not OK.
                        messageInLinkedList(); // (7) Call static method
                    }
                    public static void main (String[] args) {
                        int maxLists = maxNumOfLists; // (8)
                        // int maxNodes = max; // (9) Not OK.
                        messageInLinkedList(); // (10) Call static method
                    }
                } // Node
        } // MyLinkedList
} // ListPool
```

3 Non-Static Member Classes

Non-static member classes are inner classes that are defined without the keyword static as members of an enclosing class or interface. Non-static member classes are on par with other non-static members defined in a class. The following aspects about non-static member classes should be noted:

- An instance of a non-static member class can **only exist with an instance of its enclosing class**.
  - This means that an instance of a non-static member class **must be created in the context of an instance** of the enclosing class.
  - This also means that a **non-static member class cannot have static members**. In other words,
  - Directly the non-static member **class does not provide any services, only instances of the class do**
  - However, final static variables are allowed, as these are constants.
- Code in a non-static member class can directly refer to any member (including nested) of any enclosing class or interface, including private members. No fully qualified reference is required.
- Since a non-static member class is a member of an enclosing class, it can have any accessibility: public, package/default, protected, or private

A typical application of non-static member classes **is implementing data structures.** For example, a class **for linked lists could define the nodes** in the list with the help of a non-static member class which could be declared private so that it was not accessible outside of the top-level class. **Nesting promotes encapsulation, and the close proximity** allows classes to exploit each other's capabilities.

3.1 Instantiating Non-Static Member Classes

In the below Example, the declaration of a static variable at (6) in class Node is flagged as a compile-time error, but defining a final static variable at (7) is allowed.

A special form of the new operator is used to instantiate a non-static member class:

*<enclosing object reference>.**new** <non-static member class constructor call>*

A new instance of the non-static member class is created and associated with the indicated instance of the enclosing class.

Note that the **expression returns a reference value** that denotes a new instance of the non-static member class.

It is **illegal to specify the full name of the non-static member class in the constructor call**, as the enclosing context is already given by the <enclosing object reference>.

The non-static method makeInstance() at (3) in the class MyLinkedList creates an instance of the Node using the new operator, as shown at (4):

```
return new Node(info, next);
```

This inner object is denoted by the reference node1. This reference can then be used in the normal way to access members of the inner object.

An attempt to create an instance of the non-static member class without an outer instance, using the new operator with the full name of the inner class, as shown at (16), results in a compile-time error.

The special form of the new operator is also used in the object creation expression at (17).

```
MyLinkedList.Node node2 = list.new Node("node2", node1);

class MyLinkedList { // (1)
	private String message = "Shine the light"; // (2)
	public Node makeInstance(String info, Node next) { // (3)
		return new Node(info, next); // (4)
	}
	public class Node { // (5) NSMC
		// static int maxNumOfNodes = 100; // (6) Not OK.
		final static int maxNumOfNodes = 100; // (7) OK.
		private String nodeInfo; // (8)
		private Node next;
		public Node(String nodeInfo, Node next) { // (9)
			this.nodeInfo = nodeInfo;
			this.next = next;
		}
		public String toString() {
			return message + " in " + nodeInfo + " (" + maxNumOfNodes + ")"; //
		(10)
		}
	}
}
public class ListClient { // (11)
	public static void main(String[] args) { // (12)
		MyLinkedList list = new MyLinkedList(); // (13)
		MyLinkedList.Node node1 = list.makeInstance("node1", null); // (14)
		System.out.println(node1); // (15)
		// MyLinkedList.Node nodeX
		// = new MyLinkedList.Node("nodeX", node1); // (16) Not OK.
		MyLinkedList.Node node2 = list.new Node("node2", node1); // (17)
		System.out.println(node2); // (18)
	}
}
```

3.2 Instantiating Non-Static Member Classes

An implicit reference to the enclosing object is always available in every method and constructor of a non-static member class.

A method can explicitly use this reference with a special form of the this construct. From within a non-static member class, it is possible to refer to all members in the enclosing class directly.

return **MyLinkedList.*this*.message** + " in " + this.nodeInfo + " (" + this.maxNumOfNodes + ")";

The below expression evaluates to a reference that denotes the enclosing object (of the class <enclosing class name>) of the current instance of a non-static member class.

<enclosing class name>.this

3.2.1 *Accessing Hidden Members*

```
//Filename: Client2.java
class TLClass { // (1) TLC
        private String id = "TLClass "; // (2)
        public TLClass(String objId) { id = id + objId; } // (3)
        public void printId() { // (4)
                System.out.println(id);
        }
        class InnerB { // (5) NSMC
                private String id = "InnerB "; // (6)
                public InnerB(String objId) { id = id + objId; } // (7)
                public void printId() { // (8)
                        System.out.print(TLClass.this.id + " : "); // (9) Refers to (2)
                        System.out.println(id); // (10) Refers to (6)
                }
                class InnerC { // (11) NSMC
                        private String id = "InnerC "; // (12)
                        public InnerC(String objId) { id = id + objId; } // (13)
                                public void printId() { // (14)
                                System.out.print(TLClass.this.id + " : "); // (15) Refers to (2)
                                System.out.print(InnerB.this.id + " : "); // (16) Refers to (6)
                                System.out.println(id); // (17) Refers to (12)
                        }
                        public void printIndividualIds() { // (18)
                                TLClass.this.printId(); // (19) Calls (4)
                                InnerB.this.printId(); // (20) Calls (8)
                                printId(); // (21) Calls (14)
                        }
                } // InnerC
        } // InnerB
} // TLClass
```

```
public class OuterInstances { // (22)
    public static void main(String[] args) { // (23)
        TLClass a = new TLClass("a"); // (24)
        TLClass.InnerB b = a.new InnerB("b"); // (25)
        TLClass.InnerB.InnerC c1 = b.new InnerC("c1"); // (26)
        TLClass.InnerB.InnerC c2 = b.new InnerC("c2"); // (27)
        b.printId(); // (28)
        c1.printId(); // (29)
        c2.printId(); // (30)
        TLClass.InnerB bb = new TLClass("aa").new InnerB("bb"); // (31)
        TLClass.InnerB.InnerC cc = bb.new InnerC("cc"); // (32)
        bb.printId(); // (33)
        cc.printId(); // (34)
        TLClass.InnerB.InnerC ccc = new TLClass("aaa").new InnerB("bbb").new
        InnerC("ccc");// (35)
        ccc.printId(); // (36)
        System.out.println("------------");
        ccc.printIndividualIds(); // (37)
    }
}
```

Output from the program:

```
TLClass a : InnerB b
TLClass a : InnerB b : InnerC c1
TLClass a : InnerB b : InnerC c2
TLClass aa : InnerB bb
TLClass aa : InnerB bb : InnerC cc
TLClass aaa : InnerB bbb : InnerC ccc
------------
TLClass aaa
TLClass aaa : InnerB bbb
TLClass aaa : InnerB bbb : InnerC ccc
```

Fields and methods in the enclosing context can be **hidden by fields and methods with the same names in the non-static member** class. The special form of ***this*** syntax can be used to access members in the enclosing context, somewhat analogous to using the keyword super in subclasses to access hidden superclass members.

All three classes have a private non-static String field named id and a non-static method named printId. The member name in the nested class hides the name in the enclosing context. **These members are not overridden in the nested classes because no inheritance is involved.** In order to refer to the hidden members, the nested class can use the special this construct

3.2.2 Inheritance Hierarchy and Enclosing Context

Inner classes can extend other classes, and vice versa.
An inherited field (or method) in an inner subclass can hide a field (or method) with the same name in the enclosing context.
Using the simple name to access this member will access the inherited member, not the one in the enclosing context.

```
class Superclass {
        protected double x = 3.0e+8;
}
//______________________________________________________________________________
class TopLevelClass { // (1) Top-level Class
        private double x = 3.14;
        class Inner extends Superclass { // (2) Non-static member Class
                public void printHidden() { // (3)
                        // (4) x from superclass:
                        System.out.println("this.x: " + this.x);
                        // (5) x from enclosing context:
                        System.out.println("TopLevelClass.this.x: " + TopLevelClass.this.x);
                }
        } // Inner
} // TopLevelClass
______________________________________________________________________________
public class HiddenAndInheritedAccess {
        public static void main(String[] args) {
                TopLevelClass.Inner ref = new TopLevelClass().new Inner();
                ref.printHidden();
        }
}
```

3.2.2.1 Extending an inner class

Note that *SubclassC and the class OuterA are not related in any way*, and that the subclass OuterB inherits the class InnerA from its superclass OuterA.

An instance of SubclassC is created at (8).

An instance of the class OuterA is explicitly passed as argument in the constructor call to SubclassC.

The constructor at (4) for SubclassC has a special super() call in its body at (5).
This call ensures that the constructor of the superclass InnerA has an outer object (denoted by the reference outerRef) to bind to.

The non-default constructor at (4) **and the outerRef.super() expression at (5) are mandatory** to set up the proper relationships between the objects involved.

```
class OuterA { // (1)
	class InnerA { } // (2)
}
________________________________________________________________________________
class SubclassC extends OuterA.InnerA { // (3) Extends NSMC at (2)
	// (4) Mandatory non-default constructor:
	SubclassC(OuterA outerRef) {
		outerRef.super(); // (5) Explicit super() call
	}
}
________________________________________________________________________________
class OuterB extends OuterA { // (6) Extends class at (1)
	class InnerB extends OuterB.InnerA { } // (7) Extends NSMC at (2)
}
//______________________________________________________________________________
public class Extending {
	public static void main(String[] args) {
		// (8) Outer instance passed explicitly in constructor call:
		new SubclassC(new OuterA());
		// (9) No outer instance passed explicitly in constructor call to InnerB:
		new OuterB().new InnerB();
	}
}
```

4 Local Classes

A local class is an inner class that is defined in a block. This could be a method body, a constructor body, a local block, a static initializer, or an instance initializer.

Blocks in a non-static context have a ***this*** reference available, which refers to an instance of the class containing the block. An instance of a local class, which is declared in such a non-static block, has an instance of the enclosing class associated with it.

This gives such a non-static local class much of the same capability as a non-static member class.

If the block containing a local class declaration is defined in a **static context** (that is, a static method or a static initializer), the local class is implicitly static in the sense that its instantiation does not require any outer object.

This aspect of local classes is reminiscent of static member classes. However, note that a local class cannot be specified with the keyword static.

Some restrictions that apply to local classes are

- Local classes cannot have static members, as they cannot provide class-specific services. However, final static fields are allowed, as these are constants. This is illustrated in Example 8.9 at (1) and (2) in the NonStaticLocal class, and also by the StaticLocal class at (11) and (12).

- Local classes cannot have any accessibility modifier. The declaration of the class is only accessible in the context of the block in which it is defined, subject to the same scope rules as for local variable declarations.

4.1 Accessing Declarations in Enclosing Context

Declaring a local class in a static or a non-static block influences what the class can access in the enclosing context

4.1.1 Accessing Local Declarations in the Enclosing Block

A local class can access final local variables, final method parameters, and final catch-block parameters in the scope of the local context. Such final variables are also read-only in the local class.
This also applies to static local classes, as shown at (13) and (14) in the StaticLocal class.

Access to non-final local variables is not permitted from local classes

Declarations in the enclosing block of a local class can be hidden by declarations in
the local class. the field hides the local variable by the same name in the enclosing method.
There is no way for the local class to refer to such hidden declarations

```
class Base{
          protected  int nsf1;
}
class TLCWithLocalClasses { // Top level Class
        private double nsf1; // Non-static field
        private int nsf2; // Non-static field
        private static int sf; // Static field
        void nonStaticMethod(final int fp) { // Non-static Method
                final int flv = 10; // final local variable
                final int hlv = 30; // final (hidden) local variable
                int nflv = 20; // non-final local variable
                class NonStaticLocal extends Base { // Non-static local class
                        //static int f1; // (1) Not OK. Static members  not allowed.
                        final static int f2 = 10;// (2) final static members allowed.
                        int f3 = fp; // (3) final param from enclosing method.
                        int f4 = flv; // (4) final local var from enclosing method.
                        //double f5 = nflv; // (5) Not OK. Only finals from enclosing method.
                        double f6 = nsf1; // (6) Inherited from superclass.
                        double f6a = this.nsf1; // (6a) Inherited from superclass.
                        double f6b = super.nsf1; // (6b) Inherited from superclass.
                        double f7 = TLCWithLocalClasses.this.nsf1;// (7) In enclosing object.
                        int f8 = nsf2; // (8) In enclosing object.
                        int f9 = sf; // (9) static from enclosing class.
                        int hlv; // (10) Hides local variable.
                }
        }
        static void staticMethod(final int fp) { // Static Method
                final int flv = 10; // final local variable
                final int hlv = 30; // final (hidden) local variable
                int nflv = 20; // non-final local variable
                class StaticLocal extends Base { // Static local class
                        //static int f1; // (11) Not OK. Static members  not allowed.
                        final static int f2 = 10;// (12) final static members allowed.
                        int f3 = fp; // (13) final param from enclosing method.
                        int f4 = flv; // (14) final local var from enclosing method.
                        //double f5 = nflv; // (15) Not OK. Only finals from enclosing method.
                        double f6 = nsf1; // (16) Inherited from superclass.
                        double f6a = this.nsf1; // (16a) Inherited from superclass.
                        double f6b = super.nsf1; // (16b) Inherited from superclass.
                        //double f7 = TLCWithLocalClasses.this.nsf1; //(17) No enclosing object.
                        //int f8 = nsf2; // (18) No enclosing object.
                        int f9 = sf; // (19) static from enclosing class.
                        int hlv; // (20) Hides local variable.
                }
        }
}
```

4.1.2 Accessing Members in the Enclosing Class

A local class can access members inherited from its superclass in the usual way. by using the field's simple name, the standard this reference, and the super keyword, respectively. This also applies for static local classes.

Fields and methods in the enclosing class can be hidden by member declarations in the local class.

The non-static field nsf1, inherited by the local classes, hides the field by the same name in the class TLCWithLocalClasses. A non-static local class can access both static and non-static members defined in
the enclosing class.

The special form of the this construct can be used in non-static local classes for explicit referencing of members in the enclosing class, regardless of whether these members are hidden or not.

However, the special form of the this construct cannot be used in a static local class, as shown at (17), since it does not have any notion of an outer object. The static local class cannot refer to such hidden declarations

However, a static local class can only directly access members defined in the enclosing class that are static.

4.2 Instantiating Local Classes

Clients outside the scope of a local class cannot instantiate the class directly because such classes are, after all, local.

A local class can be instantiated in the block in which it is defined. Like a local variable.

A local class must be declared before being used in the block.

A method can return instances of any local class it declares. The local class type must then be assignable to the return type of the method.

The return type cannot be the same as the local class type, since this type is not accessible outside of the method. A supertype of the local class must be specified as the return type.

This also means that, in order for the objects of the local class to be useful outside the method, a local class should implement an interface or override the behavior of its supertypes.

As references to a local class cannot be declared outside of the local context, the functionality of the class is only available through supertype references. The method draw() is invoked on objects in the array at (14). The program output indicates which objects were created.

```
interface IDrawable { // (1)
```

```
    void draw();
}
//____________________________________________________________________________
class Shape implements IDrawable { // (2)
    public void draw() { System.out.println("Drawing a Shape."); }
}
//____________________________________________________________________________
class Painter { // (3) Top-level Class
    public Shape createCircle(final double radius) { // (4) Non-static Method
        class Circle extends Shape { // (5) Non-static local class
            public void draw() {
                System.out.println("Drawing a Circle of radius: " + radius);
            }
        }
        return new Circle(); // (6) Object of non-static local class
    }
    public static IDrawable createMap() { // (7) Static Method
        class Map implements IDrawable { // (8) Static local class
            public void draw() { System.out.println("Drawing a Map."); }
        }
        return new Map(); // (9) Object of static local class
    }
}
//____________________________________________________________________________
public class LocalClassClient {
    public static void main(String[] args) {
        IDrawable[] drawables = { // (10)
            new Painter().createCircle(5), // (11) Object of non-static local class
            Painter.createMap(), // (12) Object of static local class
            new Painter().createMap() // (13) Object of static local class
            };
        for (int i = 0; i < drawables.length; i++) // (14)
            drawables[i].draw();
        System.out.println("Local Class Names:");
        System.out.println(drawables[0].getClass()); // (15)
        System.out.println(drawables[1].getClass()); // (16)
    }
}
```

Output from the program:

```
Drawing a Circle of radius: 5.0
Drawing a Map.
Drawing a Map.
Local Class Names:
class Painter$1$Circle
class Painter$1$Map
```

The code shows how local classes can be used, together with assertions, to implement certain kinds of postconditions. The basic idea is that a computation wants to save or cache some data that is later required when checking a postconditon.

```
class Account {
    int balance;
    /** (1) Method makes a deposit into an account. */
    void deposit(final int amount) {
        /** (2) Local class to save the necessary data and to check
            that the transaction was valid. */
        class Auditor {
            /** (3) Stores the old balance. */
            private int balanceAtStartOfTransaction = balance;
            /** (4) Checks the postcondition. */
            void check() {
                assert balance - balanceAtStartOfTransaction == amount;
            }
        }

        Auditor auditor = new Auditor(); // (5) Save the data.
        balance += amount; // (6) Do the transaction.
        auditor.check(); // (7) Check the postcondition.
    }
    public static void main(String[] args) {
        Account ac = new Account();
        ac.deposit(250);
    }
}
```

5 Anonymous Classes

Classes are usually first defined and then instantiated using the **new** operator.

Anonymous classes combine the process of **definition and instantiation into a single step.** Anonymous classes are defined at the location they are instantiated, using **additional syntax** with the **new** operator. As these classes do not have a name, an instance of the class can only be created together with the definition

An anonymous class can be defined and instantiated in contexts where a reference value can be used. Anonymous classes are typically used for creating objects on the fly in contexts such as

- the value in a return statement
- an argument in a method call
- in initialization of variables

Typical uses of anonymous classes are to implement

- event listeners in GUI-based applications,
- threads for simple tasks

5.1 Extending an Existing Class

The following syntax can be used for defining and instantiating an anonymous class that extends an existing class specified by <superclass name>:

```
new <superclass name> (<optional argument list>) {
        <member declarations>
}
```

Optional arguments can be specified, **which are passed to the superclass constructor**. Thus, the superclass must provide a constructor corresponding to the arguments passed.

No extends clause is used in the construct. Since **an anonymous class cannot define constructors** (as it does not have a name), an instance initializer can be used to achieve the same effect as a constructor.

Only non-static members and final static fields can be declared in the class body

createShape() at (4) defines a non-static anonymous class at (5), which extends the superclass Shape. The anonymous class **overrides the inherited method draw().**

As we cannot declare references of an anonymous class, the functionality of the class is only available through superclass references, any non-overridden methods will not be accessed in external client

```
interface IDrawable{ // (1)

        void draw();
}
________________________________________________________________________________
class Shape implements IDrawable { // (2)
        public void draw() { System.out.println("Drawing a Shape."); }
}
________________________________________________________________________________
class Painter { // (3) Top-level Class
        public Shape createShape() { // (4) Non-static Method
                return new Shape(){ // (5) Extends superclass at (2)
                        public void draw() { System.out.println("Drawing a new Shape."); }
                };
        }
        public static IDrawable createIDrawable() { // (7) Static Method
                return new IDrawable(){ // (8) Implements interface at (1)
                        public void draw() {
                                System.out.println("Drawing a new IDrawable.");
                        }
                };
        }
}
________________________________________________________________________________
public class AnonClassClient {
        public static void main(String[] args) { // (9)
                IDrawable[] drawables = { // (10)
                                new Painter().createShape(), // (11) non-static anonymous class
                                Painter.createIDrawable(), // (12) static anonymous class
                                new Painter().createIDrawable() // (13) static anonymous class
                        };
                for (int i = 0; i < drawables.length; i++) // (14)
                        drawables[i].draw();
                System.out.println("Anonymous Class Names:");
                System.out.println(drawables[0].getClass());// (15)
                System.out.println(drawables[1].getClass());// (16)
        }
}
```

Output from the program:

```
        Drawing a new Shape.
        Drawing a new IDrawable.
        Drawing a new IDrawable.
        Anonymous Class Names:
        class Painter$1
        class Painter$2
```

5.2 Implementing an Interface

The following syntax can be used for defining and instantiating an anonymous class that implements an interface specified by <interface name>:

new <interface name>() { <member declarations> }

An anonymous class provides a single interface implementation, and no arguments are passed.

The anonymous class implicitly extends the Object class. Note that **no implements clause is used in the construct**

The functionality of objects of an anonymous class that implements an interface is available through references of the interface type and the Object type (i.e., the supertypes).

The anonymous class below implements the ActionListener interface that has the method actionPerformed().

When the addActionListener() method is called on the GUI-button denoted by the reference quitButton, the anonymous class is instantiated and the reference value of the object is passed as a parameter to the method.

The method addActionListener() of the GUI-button can use the reference value to invoke the method actionPerformed() in the ActionListener object.

```
quitButton.addActionListener (
        new ActionListener() { // (1) Anonymous class implements an interface.
                // Invoked when the user clicks the quit button.
                public void actionPerformed(ActionEvent evt) {
                        System.exit(0); // (2) Terminates the program.
                }
        }
);
```

5.3 Instantiating Anonymous Classes

The discussion on instantiating local classes (see Example 8.10) is also valid for instantiating anonymous classes.

The class AnonClassClient in the above Example creates one instance at (11) of the non-static anonymous class defined at (5), and two instances at (12) and (13) of the static anonymous class defined at (8).
The program output shows the polymorphic behavior and the runtime types of the objects.

Similar to a non-static local class, an instance of a non-static anonymous class has an instance of its enclosing class at (11).

An enclosing instance is not mandatory for creating objects of a static anonymous class, as shown at (12).

5.4 Accessing Declarations in Enclosing Context

Access rules for local classes also apply to anonymous classes. Below illustrates the access rules for anonymous classes

Anonymous classes can access **final** variables only in the **enclosing context**.
Inside the definition of a non-static anonymous class, members of the enclosing context can be referenced using the <enclosing class name>.this construct.

Non-static anonymous classes can also access any non-hidden members in the enclosing context by their simple names, whereas static anonymous classes can only access non hidden static members.

```
class Base {
        protected int nsf1;
}
//______________________________________________________________________
class TLCWithAnonClasses { // Top level Class
        private double nsf1; // Non-static field
        private int nsf2; // Non-static field
        private static int sf; // Static field
        Base nonStaticMethod(final int fp) { // Non-static Method
                final int flv = 10; // final local variable
                final int hlv = 30; // final (hidden) local variable
                int nflv = 20; // non-final local variable
                return new Base() { // Non-static anonymous class
                        //static int f1; // (1) Not OK. Static members not allowed.
                        final static int f2 = 10; // (2) final static members allowed.
                        int f3 = fp; // (3) final param from enclosing method.
                        int f4 = flv; // (4) final local var from enclosing method.
                        //double f5 = nflv; // (5) Not OK. Only finals from enclosing method.
                        double f6 = nsf1; // (6) Inherited from superclass.
                        double f6a = this.nsf1; // (6a) Inherited from superclass.
                        double f6b = super.nsf1; // (6b) Inherited from superclass.
                        double f7 = TLCWithAnonClasses.this.nsf1; // (7) In enclosing object.
                        int f8 = nsf2; // (8) In enclosing object.
                        int f9 = sf; // (9) static from enclosing class.
                        int hlv; // (10) Hides local variable.
                };
        }
        static Base staticMethod(final int fp) { // Static Method
                final int flv = 10; // final local variable
                final int hlv = 30; // final (hidden) local variable
                int nflv = 20; // non-final local variable
                return new Base() { // Static anonymous class
                        //static int f1; // (11) Not OK. Static members not allowed.
                        final static int f2 = 10; // (12) final static members allowed.
                        int f3 = fp; // (13) final param from enclosing method.
                        int f4 = flv; // (14) final local var from enclosing method.
                        //double f5 = nflv; // (15) Not OK. Only finals from enclosing method.
                        double f6 = nsf1; // (16 ) Inherited from superclass.
                        double f6a = this.nsf1; // (16a) Inherited from superclass.
                        double f6b = super.nsf1; // (16b) Inherited from superclass.
                        //double f7 = TLCWithAnonClasses.this.nsf1; //(17) No enclosing object.
                        //int f8 = nsf2; // (18) No enclosing object.
                        int f9 = sf; // (19) static from enclosing class.
                        int hlv; // (20) Hides local variable.
                };
        }
    }
```