Nested Type Declarations

Overview of Nested Type Declarations

- 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.
- A *top-level* class, enum type, or interface is one that is *not nested*. By a nested type we mean either a nested class, a nested enum, or a nested interface.

Overview of Nested Type Declarations

- In addition to the top-level types, there are four categories of nested classes, one of nested enum types, and one of nested interfaces, defined by the context these nested types are declared in:
  - ➢ static member classes, enums, and interfaces
  - ➢ non-static member classes
  - ➢ local classes
  - ➢ anonymous classes

Inner Classes

The last three categories (non-static member classes, local classes, anonymous classes) are collectively known as inner classes.

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.
- An instance of an inner class can access the members of its immediately enclosing instance by their simple names.

Static member: class, enum, interface

- 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.
- An enum or an interface cannot be instantiated with the new operator.

Note that there are no non-static member, local, or anonymous interfaces.

Non-static member classes

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

Local classes

- *Local classes* can be defined in the context of a block as in a method body or a local block, just as local variables can be defined in a method body or a local block

Anonymous classes

- 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, if the local (or anonymous) class is declared in a non-static context.

Example

```
class TLC {                           // (1) Top level class
  static class SMC {/*...*/} // (2) Static member class
  interface SMI {/*...*/}    // (3) Static member interface
  class NSMC {/*...*/}       // (4) Non-static member (inner) class
  void nsm() {
    class NSLC {/*...*/}    // (5) Local(inner) class in non-static context
  }
  static void sm() {
    class SLC {/*...*/}     // (6) Local (inner) class in static context
  }
  SMC nsf = new SMC() { // (7) Anonymous(inner) class in non-static context
    /*...*/
  };
  static SMI sf = new SMI() { // (8) Anonymous(inner) class in static
                             //                                   context
    /*...*/
  };
  enum SME {/*...*/}   // (9) Static member enum
}
```

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 top-level 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 non-static 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

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.

Static Member Types

- 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*.

Static Member Types

- The accessibility modifiers allowed for members in an enclosing type declaration can naturally be used for nested types.
- Static member classes, enum types and interfaces can only be declared in top-level type declarations, or within other nested static members.

```
//Filename: ListPool.java
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!
}
```

Comments on example

- 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, as shown at (5) in previous example. Its accessibility modifier (and that of the types making up its full name) come into play when it is referenced by an external client.
- The declaration at (8) in example will not compile because the member interface ListPool.MyLinkedList.ILink has private accessibility

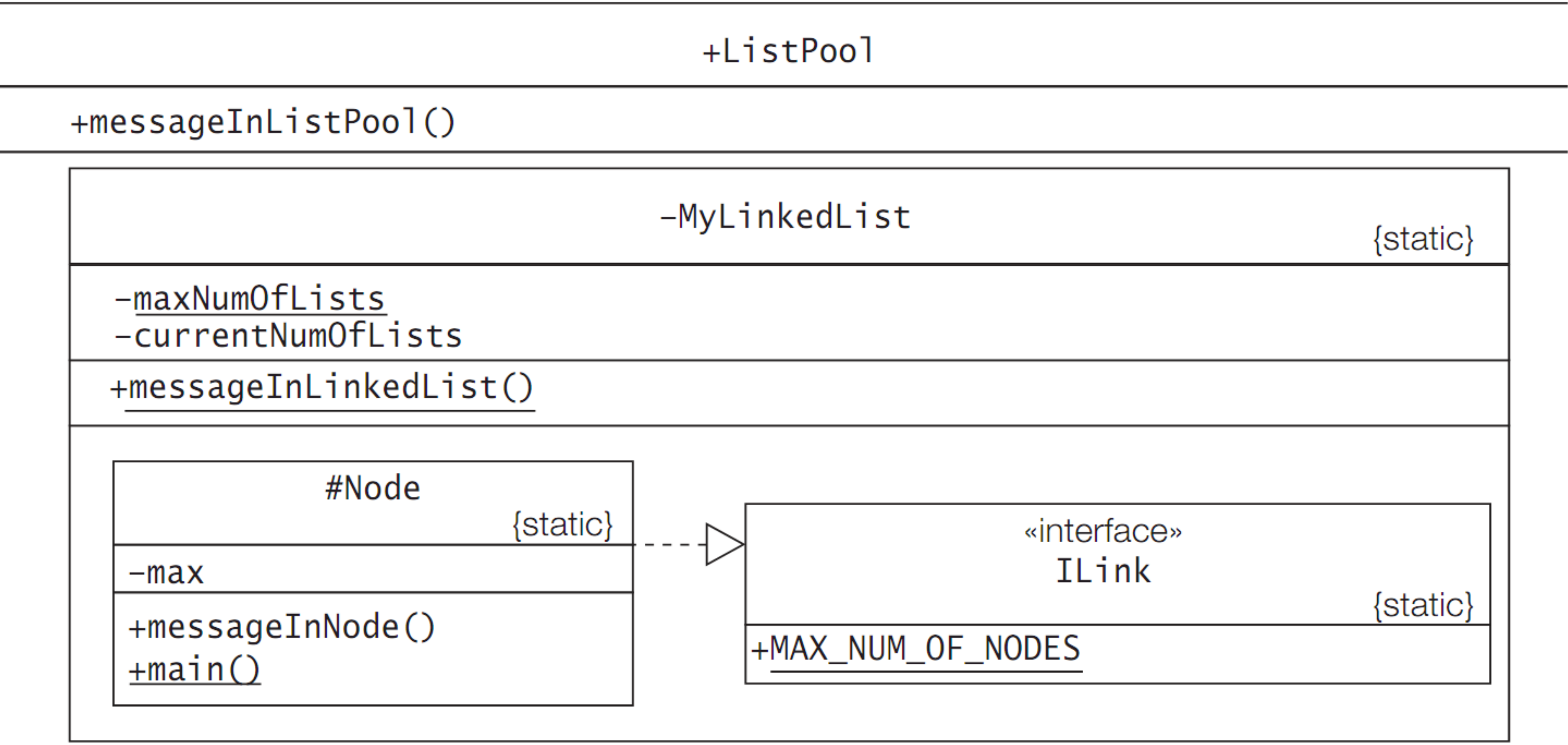
Static Import

- 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.

Import and Static Import

```
//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!
```

Accessing Members in Enclosing Context



```
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
```

Non-Static Member Classes

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, 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.

Non-Static Member Classes

- 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.

Non-Static Member Classes

- 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.

Instantiating Non-Static Member Classes

```
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)
    }
  }
}
```

Instantiating Non-Static Member Classes

```
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)
  }
}
```

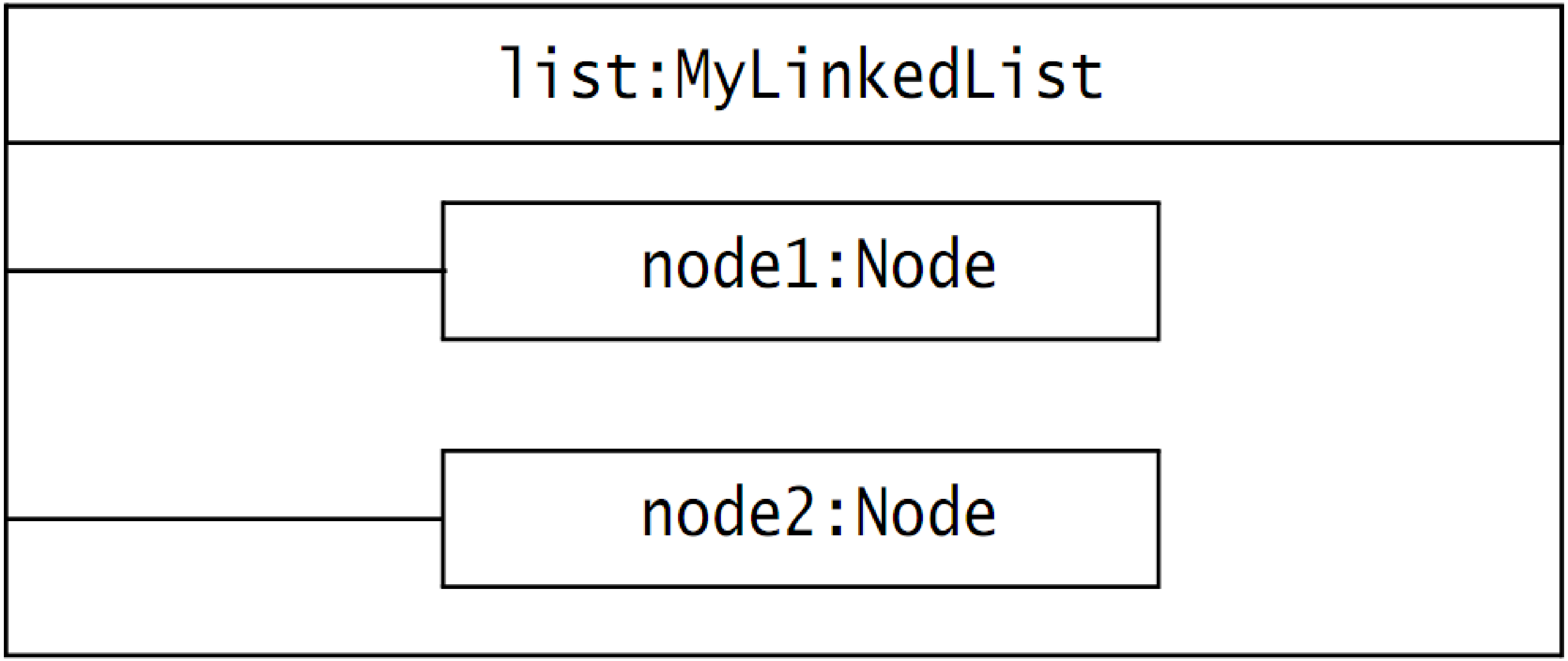
Instantiating Non-Static Member Classes

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>
```

- The <enclosing object reference> in the object creation expression evaluates to an instance of the enclosing class in which the designated non-static member class is defined.
- A new instance of the non-static member class is created and associated with the indicated instance of the enclosing 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>

Outer Object with Associated Inner Objects



Accessing Members in Enclosing Context

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, as explained in the next example.

An example is shown at (10), where the field message from the enclosing class is accessed in the non-static member class.

```
return message + " in " + nodeInfo + " (" + maxNumOfNodes + ")";//(10)
```

It is also possible to explicitly refer to members in the enclosing class, but this requires special usage of the this reference. One might be tempted to write the statement at (10) as follows:

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

Accessing Members in Enclosing Context

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, as explained in the next example.

An example is shown at (10), where the field message from the enclosing class is accessed in the non-static member class.

```
return message + " in " + nodeInfo + " (" + maxNumOfNodes + ")";//(10)
```

incorrect:

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

correct:

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

Accessing Members in Enclosing Context

The expression

`<enclosing class name>.`**`this`**

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

Accessing Hidden Members

- 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 the 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.

```
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)
  }
}
```

Accessing Hidden Members

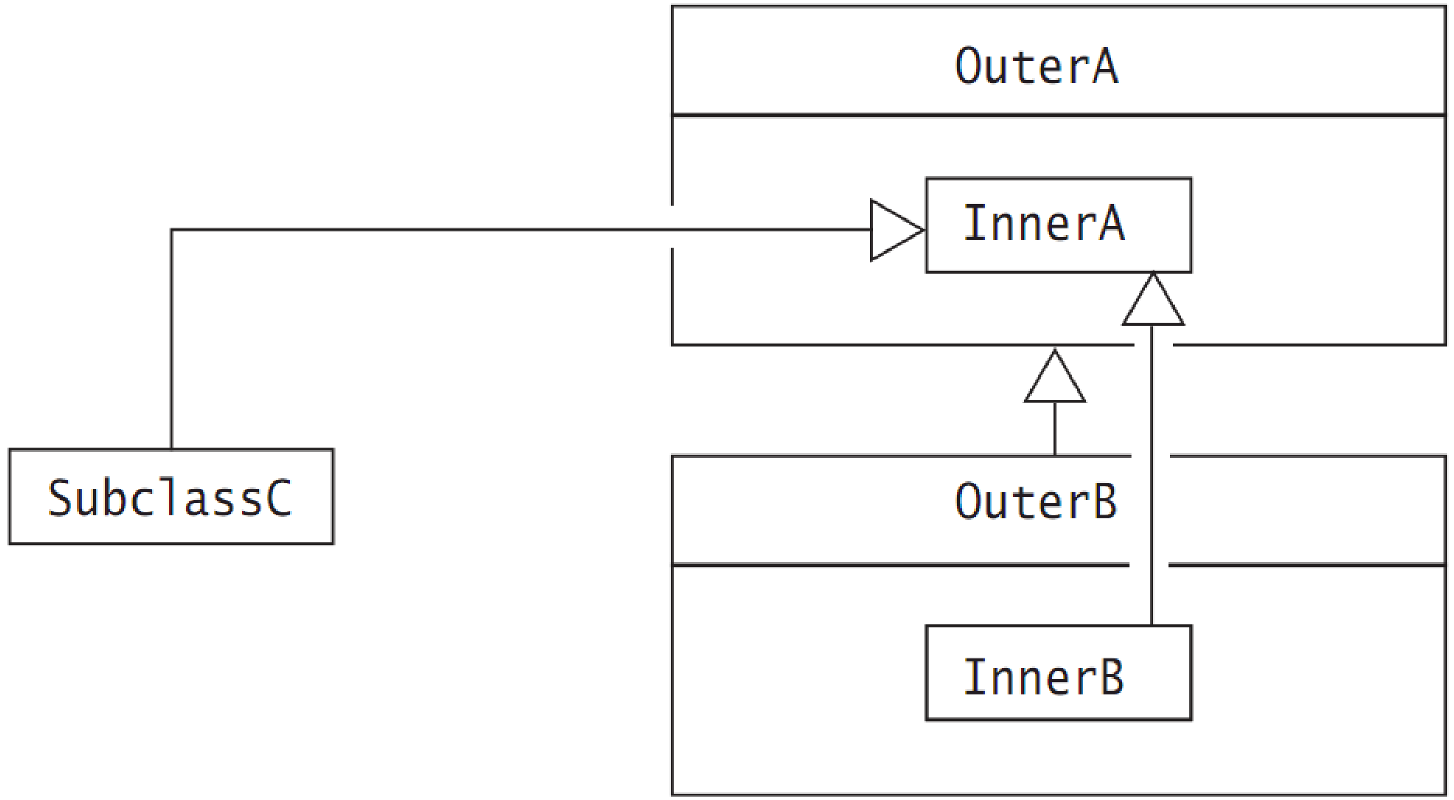
Output from the program:

```
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
```

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.

Inheritance Hierarchy and Enclosing Context



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.

Local Classes

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.
- 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.

Accessing Declarations in Enclosing Context

Accessing Local Declarations in the Enclosing Block

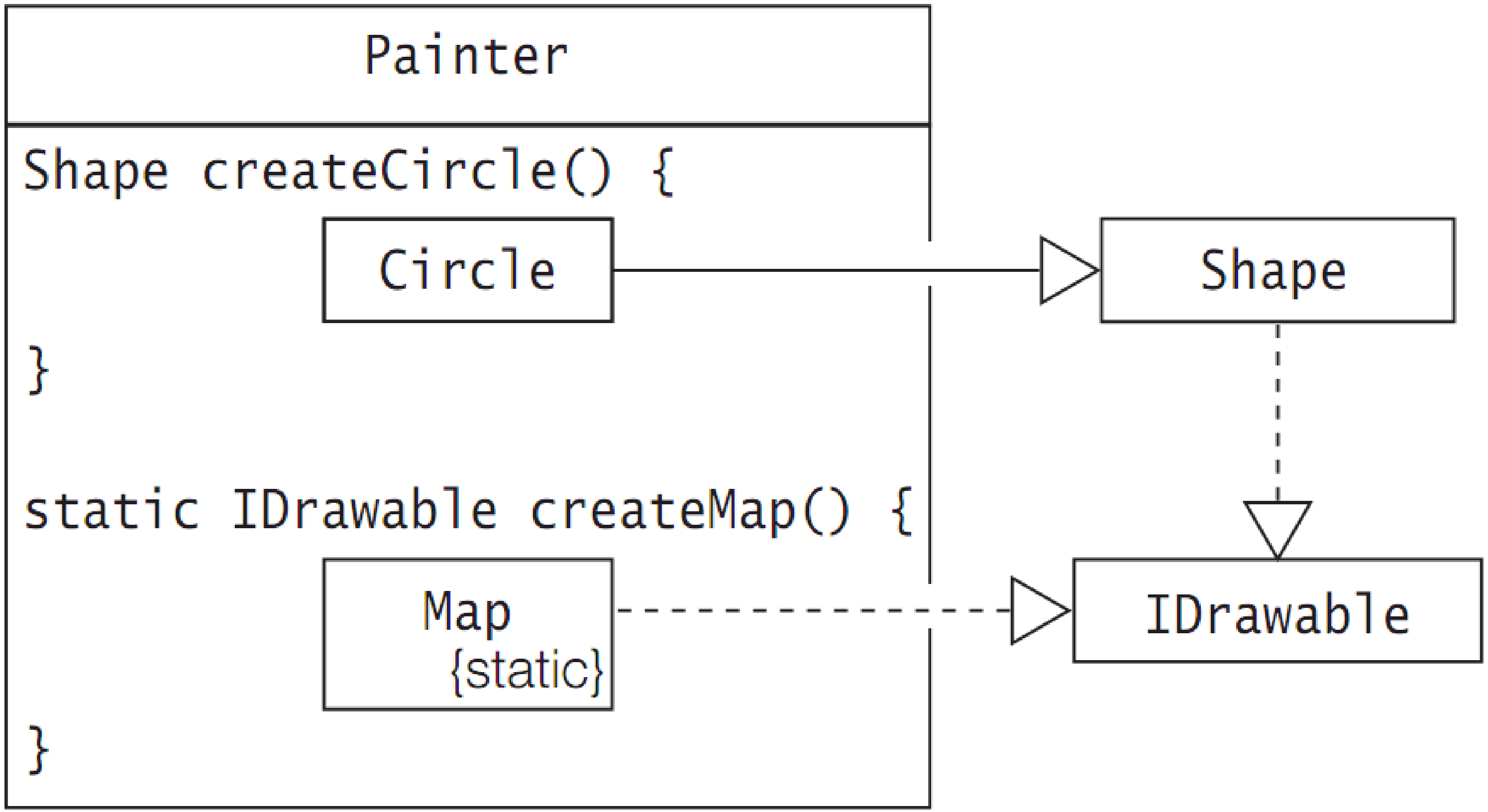
- A local class can access final local variables, final method parameters, and final catch-block parameters in the scope of the local context.
- Such final variables are also read-only in the local class.

Accessing Declarations in Enclosing Context

Accessing Members in the Enclosing Class

- A local class can access members inherited from its superclass in the usual way
- Fields and methods in the enclosing class can be hidden by member declarations in the local class.

Local Classes and Inheritance Hierarchy



Anonymous Classes

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.

Anonymous Classes

Extending an Existing Class

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

Implementing an Interface

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

Any questions?