Nested Classes

Nested Classes and Interfaces

- Classes and interfaces can be declared inside other classes and interfaces, either as members or within blocks of code.

Some Definitions

- Nested = a class or interface definition is somewhere inside another one
- Top-level class or interface = an instance of a type **does not** need to be instantiated with a reference to any enclosing instance
- Inner class = an instance of a class **does** need to be instantiated with an enclosing instance
- Inner Member class = defined inside another class, but not inside any methods.
- Inner Local class = defined inside a method
- Named Inner Local class = has a class name
- Anonymous Inner Local class = does not have a class name

Nested Classes

```
class TheEnclosingClass{
         ...
       class ANestedClass  {
            ...
        }
}
```

- **Definition**: A **nested class** is a class that is a member of another class.
- **Reason for making nested classes**:
  - the nested class **makes sense only in the context of its enclosing class**
  - the nested class **needs the enclosing class to have the right functionality**.

Nested Static Classes

- A **nested class** that is **declared static** **is attached to the enclosing class** and not to objects of the enclosing class. Instance fields and methods can not be directly accessed.

Example: *Linked List*

```java
public class LinkedList {
   private Node first;
   .....
   public static class Node{
      public Node next;
      public Object data;
   }
   .....
}
```

Static Nested Classes/Interfaces — Overview

- A nested class/interface which is declared as `static` acts just like any non-nested class/interface, except that its name and accessibility are defined by its enclosing type.
- Static nested types are members of their enclosing type
  - They can access all other members of the enclosing type including the private ones.
  - Inside a class, the static nested classes/interfaces can have private, package, protected or public access; while inside an interface, all the static nested classes/interfaces are implicitly public.
  - They serve as a structuring and scoping mechanism for logically related types

Static Inner Classes

- Since a static inner class has no connection to an object of the outer class, within an inner class method
  - Instance variables of the outer class cannot be referenced
  - Nonstatic methods of the outer class cannot be invoked
- To invoke a static method or to name a static variable of a static inner class within the outer class, preface each with the name of the inner class and a dot

Static Nested Classes/Interfaces (cont.)

- Static nested classes
  - If a class is nested in an interface, it’s always static (omitted by convention)
  - It can extend any other class, implement any interface and itself be extended by any other class to which it’s accessible
  - Static nested classes serve as a mechanism for defining logically related types within a context where that type makes sense.

Static Nested Classes/Interfaces (cont.)

- Nested interfaces
  - Nested interfaces are always static (omitted by convention) since they don’t provide implementation

Static Nested Classes/Interfaces (cont.)

```
public class BankAccount {
     private long number;     //account number
     private long balance;    //current balance
     public static class Permissions {
          public boolean canDeposit, canWithdraw,
  canClose;
     }
//  . . .
}
```

- Code outside the `BankAccount` class must use `BankAccount.Permissions` to refer to this class

```
BankAccount.Permissions perm =
  acct.permissionsFor(owner);
```

Inner Classes

- **A nested class that is not static is called an inner class.**
- An **inner class is associated with an object** of its enclosing class and it has **direct and unlimited access to that object's instance variables and methods**.
- A **nested class** can be **declared at** the **top level** inside a class **or inside any block of code**.

Inner classes

```
class Outer {
    int n;

    class Inner {
        int ten = 10;
        void setNToTen( ) { n = ten; }
    }

    void setN ( ) {
        new Inner( ).setNToTen( );
    }
}
```

Inner Class

- Name Reference
  - OuterClass inside : Use InnerClass Simple name
  - OuterClass outside : OuterClass.InnerClass

```
public static void main(String[] args) {
        OuterClass  outObj = new  OuterClass();
       OuterClass.InnerClass   inObj = outObj.new  InnerClass();
}
```

- Access Modifier
  - public, private, protected

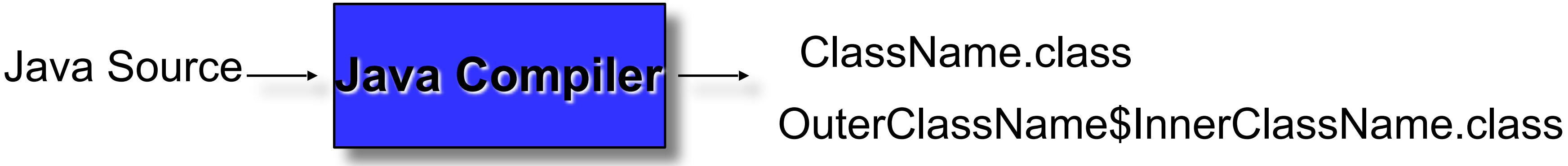
Inner class cannot have static variable

Nesting Inner Classes

- It is legal to nest inner classes within inner classes
  - The rules are the same as before, but the names get longer
  - Given class `A`, which has public inner class `B`, which has public inner class `C`, then the following is valid:

```
A aObject = new A();
A.B bObject = aObject.new B();
A.B.C cObject = bObject.new C();
```

Name of Inner Class



```
class Outer {
        class Inner1 {
            class Inner2 {   // …        }
            // ...
        }
        // ...
}
```

Outer.class
Outer$Inner1.class
Outer$Inner1$Inner2.class

Simple inner class example

```
class Outer{
  private int x1;

 Outer(int x1){
     this.x1 = x1;
  }

 public void foo(){ System.out.println(“fooing”);}

  public class Inner{
     private int x1 = 0;
     void foo(){
       System.out.println(“Outer value of x1: “ + Outer.this.x1);
       System.out.println(“Inner value of x1: “ + this.x1);
     }
  }
```

Simple example, cont -- driver

- Rules for instantiation

```
public class TestDrive{

  public static void main(String[] args){
     Outer outer = new Outer();
     Inner inner = outer.new Inner(); //must call new through
                                                     //outer object handle
      inner.foo();

      // note that this can only be done if inner is visible
      // according to the regular scoping rules
  }
}
```

Non-static Classes — Inner classes

- *Inner classes* are associated with instances of its enclosing class.

```
public class BankAccount {
    private long number;     //  account number
    private long balance;    // current balance
    private Action lastAct; //last action performed

    public class Action {
        private String act;
        private long amount;

        Action(String act, long amount) {
            this.act = act;
            this.amount = amount;
        }
    }
```

Non-static Classes — Inner classes

```
   public void deposit(long amount) {
      balance += amount;
      lastAct = new Action(“deposit”, amount);
   }

   public void withdraw(long amount) {
      balance -= amount;
      lastAct = new Action(“withdraw”, amount);
   }
   // . . .
}
```

Inner classes (cont.)

- When an inner class object is created, it MUST be associated with an object of its enclosing class.
- Usually, inner class objects are created inside instance methods of the enclosing class. When this occurs, the current enclosing object `this` is associated with the inner object by default.

```
lastAct = this.new Action(“deposit”, amount);
```

  - When `deposit` creates an `Action` object, a reference to the enclosing `BankAccount` object is automatically stored in the `Action` object.

Inner class

```
class BankAccount
{     ...
   private double balance;

   // Inner Class
   private class InterestAdder implements ActionListener
   {  private double rate;
      ...
      public void actionPerformed(ActionEvent event)
      {
         // update interest
         double interest = balance * rate / 100;
         balance += interest;
         ...
      }
   }
}
```

Inner classes

- Using the saved reference, the inner-class object can refer to the enclosing object's fields directly by their names. The full name will be the enclosing object `this` preceded by the enclosing class name

- To refer to the field number in the BankAccount part

```
BankAccount.this.balance
```

- With inner classes, Java defines a outer class reference with each inner class which can be accessed with **BankAccount.this**

```
public void actionPerformed(ActionEvent e)
{
   double interest = BankAccount.this.balance *
this.rate/100;
   BankAccount.this.balance += interest;
}
```

- The balance field is a private member of the outer class.

Inner classes (cont.)

- method for `transfer` is added

```
public void transfer(BankAccount other, long
  amount) {

   other.withdraw(amount);
   deposit(amount);

  lastAct = this.new Action(“transfer”, amount);
  other.lastAct = other.new Action(“transfer”,
    amount);
  }
```

Inner classes (cont.)

- The enclosing class can also access the private members of its inner class, but only via explicit reference to an inner class object.
- An object of the enclosing class need not have any inner class objects associated with it, or it could have many.
- An inner class acts as a top-level class except that it can’t have static members (except for final static fields).
- Inner classes can also be extended.

Inheritance, Scoping and Hiding

- All members declared within the enclosing class are said to be in *scope* inside the inner class.
- An inner class's own fields and methods can hide those of the enclosing object. Two possible ways:

1). A member with the same name is declared in the inner class

  - Any direct use of the name refers to the version inside the inner class

```
   class Host {
      int x;
      class Helper {
         void increment() {int x=0; x+
+;}
      } }
```

  - Access to the enclosing object's members needs be preceded by `this` explicitly

Inheritance, Scoping and Hiding

2). A member with the same name is inherited by the inner class

- The direct use of the name is not allowed

```
        class Host {
            int x;
            class Helper extends Unknown {      //
     Unknown class has a field x
                void increment~~()~~ {x++;}
            }
        }
```

- Use `enclosingClassName.this.name` to refer to the version in the outer class
- or `super.name` to refer to the version in the super class
- Use `this.name` in the inner class

Specifying which scope you want:

```
public class ScopeConflict {
  String s = "outer";

  class Inner extends SuperClass {

    String s = "inner";

    void foo(){
      System.out.println(this.s);
      System.out.println(super.s);

     System.out.println(ScopeConflict.this.s);
    }
  }
}

class SuperClass {
  String s = "super";
}
```

output:

```
inner (from this.s)
super (from super.s)
outer (from ScopeConflict.this.s)
```

Inheritance, Scoping and Hiding (cont.)

- A method within an inner class which has the same name as an enclosing **method hides all overloaded forms** of the enclosing method, even if the inner class itself does not declare those overloaded forms.

```
class Outer {
   void print() {}
   void print(int value) {}
   class Inner {
      void print() {}
      void show() {
         print();
         Outer.this.print();
         print(1); // no Inner.print(int)
      }
   }
}
```

Local Inner Classes

- You can define inner classes in code blocks. They are called *local inner classes.*
- You can actually declare an inner class inside of a method, just like you could declare a local variable.
- Local classes do not get an access modifier – they are automatically restricted to the method they are defined in
- Can only refer to final members of the enclosing class
  - They are NOT members of the class which contains the code but are local to that block, as a local variable.
  - They are completely inaccessible outside of the block.
  - Only one modifier is allowed—final—which makes them unextendable

Anonymous Inner classes

- When using a local inner class, if you only want to make one instance of it, you don’t even need to give it a name
- This is known as an anonymous inner class
- These are convenient for event programming
- However, the syntax is extremely cryptic.

Anonymous Inner classes

- You have to look very carefully to see a difference between construction of a new object, and construction of a new inner class extending a class.

```
//A person object from  Person class
Person queen=new Person("Mary");  //Person Object
```

```
//An object of an inner class extending Person interface
Person count = new Person() {
                        //class code here
                       };
```

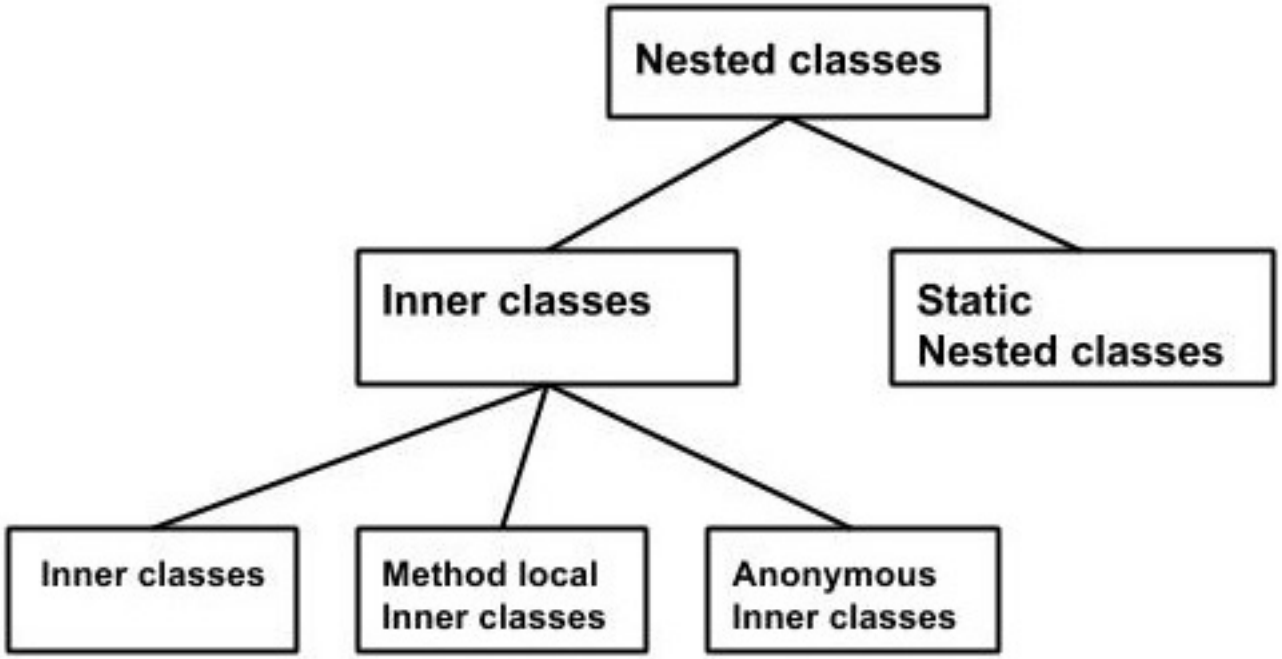
Anonymous Inner classes

- Anonymous Inner classes cannot have constructors, since constructors have to have the same name as the class, and these classes have no names.
- As you can see, the syntax for these is confusing – both for people writing and reading the code. Use this with care, if at all.

Anonymous Inner classes

```
public void start(final double rate)
{
    ActionListener adder = new ActionListener()
    {
        public void actionPerformed(ActionEvent evt)
        {
            double interest = balance * rate / 100;
            balance += interest;
        }
    };
```

- This is saying, construct a new object of a class that implements the ActionListener interface, where the one required method (actionPerformed) is defined inside the brackets.


Nested classes
Inner classes
Static
Nested classes
Inner classes
Method local
Inner classes
Anonymous
Inner classes