# Inner class

In Java, an inner class is a class that is defined inside another class. It can have access to the members of the outer class, including private members, and can be used to encapsulate and organize related functionality within a single unit.

### There are four types of inner classes in Java:

**Member inner class / Inner class**: This is the most common type of inner class. It is defined inside another class and has access to the members of the outer class. It can be instantiated only with an instance of the outer class.

**Local inner class / method-Local inner class**: This is a class that is defined inside a method or a block of code. It can access the variables and parameters of the enclosing method or block, but only if they are declared final or effectively final.

**Anonymous inner class**: This is a class that is defined inline without a name. It is typically used to define a subclass of an existing class or interface and override its methods.

**Static nested class/ static inner class**: This is a class that is defined as a static member of another class. It does not have access to the members of the outer class, but it can be instantiated without an instance of the outer class.

Inner classes can be useful for implementing callbacks, listeners, and event handlers, as well as for implementing data structures such as iterators, enumerations, and queues. They can also be used to create more readable and maintainable code by encapsulating related functionality within a single unit.

## Member inner class / Inner class

* An inner class instance has access to all members of the outer class, even those marked as private.

|  |  |
| --- | --- |
| class MyOuter{  class MyInner{}  } | javac MyOuter.java  MyOuter.class  MyOuter$MyInner.class |

You can’t run main() method of inner class directly, the only way to access the inner class is through a live instance of the outer class.

|  |  |
| --- | --- |
| public class OuterClass {  private int x;  public void outerMethod() {  InnerClass inner = new InnerClass();  inner.innerMethod();  }  public class InnerClass {  public void innerMethod() {  x = 5; // Inner class has access to the private member of the outer class  System.out.println("Inner class method called");  }  }  } | In this example, OuterClass contains a private member variable x and a public method outerMethod().  OuterClass also contains a member inner class InnerClass, which has access to the private member x of the outer class.  The innerMethod() method of InnerClass modifies the value of x and prints a message to the console.  To create an instance of InnerClass, you first need to create an instance of OuterClass:    OuterClass outer = new OuterClass();  Then, you can create an instance of InnerClass using the outer instance: |

Create an inner class object from outside of outer class instance code

To create an inner class object from outside of the outer class instance code, you will need to have an instance of the outer class first. Once you have an instance of the outer class, you can use it to create an instance of the inner class.

Here's an example:

public class OuterClass {

private int x = 5;

public class InnerClass {

public void innerMethod() {

System.out.println("x = " + x);

}

}

}

In this example, OuterClass contains an inner class InnerClass that has access to the private member variable x.

To create an instance of InnerClass from outside of the OuterClass, you will first need to create an instance of OuterClass:

OuterClass outer = new OuterClass();

Once you have an instance of OuterClass, you can create an instance of InnerClass using the new keyword and the name of the inner class:

OuterClass.InnerClass inner = outer.new InnerClass();

Note that the syntax for creating an instance of an inner class from outside of the outer class code is slightly different from the syntax for creating an instance of a regular class. You need to use the instance of the outer class to create an instance of the inner class, and you need to qualify the inner class name with the outer class name, separated by a dot (.).

|  |  |
| --- | --- |
| public class MyOuter {  private int myVariable = 10;  public void outerMethod(){  MyInner myInnerObject = new MyInner();  myInnerObject.innerMethod();  }  public class MyInner {  private int myVariable = 11;  public void innerMethod() {  System.out.println("myOuterVariable="+ myVariable);//11  System.out.println("myOuterVariable="+ this.myVariable);//11  System.out.println("myOuterVariable="+ MyOuter.this.myVariable);//10  }  }  } | public class Test {   public static void main(String[] args) {   *//how to access InnerClass inner method?   //get the outer object* MyOuter outerObject = new MyOuter();   *//create object* MyOuter.MyInner innerObject = outerObject.new MyInner();   *//access inner class method* innerObject.innerMethod();   *//second way* new MyOuter().new MyInner().innerMethod();  } } |

Can outer class access private method of inner class?

No, an outer class cannot access a private method of an inner class directly. The purpose of making a method private is to restrict its access to the class where it is defined.

However, the inner class can provide a public method that can be accessed by the outer class, and that method can in turn call the private method of the inner class. Here's an example:

public class OuterClass {

private int outerData;

private class InnerClass {

private int innerData;

private void innerMethod() {

innerData = outerData \* 2;

}

public void callInnerMethod() {

innerMethod();

}

}

public void outerMethod() {

InnerClass inner = new InnerClass();

inner.callInnerMethod();

inner. innerMethod ();

}

}

In this example, OuterClass has a private field outerData, and InnerClass has a private method innerMethod that uses outerData. However, OuterClass cannot call innerMethod directly. Instead, it creates an instance of InnerClass in its outerMethod, and calls the public method callInnerMethod, which in turn calls innerMethod. This way, OuterClass can indirectly access the private method innerMethod of InnerClass.

member modifiers applied to inner class

In Java, the member modifiers that can be applied to an inner class are the same as those that can be applied to a regular class. These modifiers include:

**public**: If you declare an inner class as public, it can be accessed from any class in any package.

**protected**: If you declare an inner class as protected, it can be accessed from any class in the same package and from any subclass of the outer class.

**private**: If you declare an inner class as private, it can only be accessed from within the outer class.

**static**: If you declare an inner class as static, it becomes a static nested class. A static nested class does not have a reference to an instance of the outer class and can be instantiated independently of any instance of the outer class.

**final**

**strictfp**

**abstract**

Here's an example:

public class OuterClass {

private int x = 5;

public class InnerClass {

private int y = 10;

public void innerMethod() {

System.out.println("x = " + x);

System.out.println("y = " + y);

}

}

public static class StaticInnerClass {

private static int z = 15;

public void staticInnerMethod() {

System.out.println("z = " + z);

}

}

}

In this example, OuterClass contains two inner classes: InnerClass and StaticInnerClass.

InnerClass is not declared as static, so it has access to the instance variables of the outer class. StaticInnerClass is declared as static, so it does not have access to the instance variables of the outer class.

InnerClass is also not declared with any access modifiers, so it has package-private access by default. This means that it can be accessed from other classes in the same package, but not from classes in other packages.

StaticInnerClass is declared as public, so it can be accessed from any class in any package.

Note that an inner class can also have other modifiers such as final or abstract, depending on the requirements of your program.

referencing the inner or outer instance from within the inner class

In Java, you can reference the inner or outer instance from within the inner class using the this keyword and the OuterClassName.this syntax.

When you use the this keyword inside an inner class, it refers to the instance of the inner class itself. If you want to refer to the instance of the outer class from within the inner class, you can use the OuterClassName.this syntax.

Here's an example:

public class OuterClass {

private int x = 5;

public class InnerClass {

private int y = 10;

public void innerMethod() {

// References the instance of the inner class

System.out.println("this.y = " + this.y);

// References the instance of the outer class

System.out.println("OuterClass.this.x = " + OuterClass.this.x);

}

}

}

In this example, OuterClass contains an inner class InnerClass. InnerClass has its own private member variable y.

In the innerMethod() method of InnerClass, you can use the this keyword to refer to the instance of InnerClass itself and the OuterClassName.this syntax to refer to the instance of the outer class.

For example, this.y refers to the value of y in the current instance of InnerClass, while OuterClass.this.x refers to the value of x in the instance of OuterClass that contains the current instance of InnerClass.

You can also use the this and OuterClassName.this syntax to call methods or access other members of the inner or outer class, as long as they are not private.

## Local inner class

In Java, a local inner class is a class that is defined inside a method or a block of code. It is similar to a member inner class in that it has access to the variables and parameters of the enclosing method or block, but it is only visible within that method or block.

Here's an example of a local inner class:

public class OuterClass {

private int x = 5;

public void outerMethod() {

final int y = 10;

class InnerClass {

public void innerMethod() {

System.out.println("x = " + x); // Accesses the private member of the outer class

System.out.println("y = " + y); // Accesses the final variable of the outer method

}

}

InnerClass inner = new InnerClass();

inner.innerMethod();

}

}

In this example, OuterClass contains a private member variable x and a public method outerMethod(). outerMethod() declares a final variable y and defines a local inner class InnerClass. The innerMethod() method of InnerClass accesses the private member x of the outer class and the final variable y of the outer method.

To create an instance of InnerClass, you simply declare and instantiate it within the outerMethod() method:

InnerClass inner = new InnerClass();

You can then call the innerMethod() method of InnerClass:

inner.innerMethod();

This will output the following message to the console:

x = 5

y = 10

Note that because the local inner class is only visible within the outerMethod() method, you cannot create an instance of InnerClass outside of that method.

Object creation must be after class declaration.

Variable used inside local method class must be final.

|  |
| --- |
| public class MethodInnerClass {    public void test() {   int x=10;   class Helper {  String name;  static int *id*;   public Helper(String name) {  this.name = name;  System.*out*.println(x);  }   public void test2(){   }   }   Helper object1 = new Helper("Ravi");  Helper object2 = new Helper("Ravi-2");    }   } |

## Anonymous inner class

In Java, an anonymous inner class is a local inner class that does not have a name. It is defined and instantiated in a single expression, often used as a way to implement interfaces or extend classes on the fly.

Here's an example of an anonymous inner class that implements an interface:

public class OuterClass {

public void doSomething() {

// Declare and instantiate an anonymous inner class that implements the Runnable interface

Runnable myRunnable = new Runnable() {

@Override

public void run() {

System.out.println("Running in a separate thread!");

}

};

// Create a new thread using the anonymous inner class as the runnable

Thread myThread = new Thread(myRunnable);

myThread.start();

}

}

In this example, OuterClass contains a method doSomething() that declares and instantiates an anonymous inner class that implements the Runnable interface. The run() method of the anonymous inner class simply prints a message to the console. The doSomething() method then creates a new thread using the anonymous inner class as the runnable and starts the thread.

Note that the anonymous inner class is defined and instantiated in a single expression, using the new keyword followed by the interface name or class name being implemented or extended. The curly braces contain the class definition and the semicolon ends the expression.

Anonymous inner classes are often used in event handling and GUI programming, where they can be used to define and instantiate event listeners and handlers. They can also be used to provide custom implementations of abstract classes or interfaces without needing to create a separate named class.

Anonymous class can implement only one interface.

Anonymous class can’t extends a class and implement an interface at the same time.

**For below example:**

Employees can be Interface

interface Employees{  
 void getName();

}

Employees can be class

class Employees{  
 void getName() {  
 System.*out*.println("Saurabh");  
 }  
  
 }

Employees can be a abstract class.

abstract class Employees{  
 void getName() {  
 System.*out*.println("Saurabh");  
 }  
  
 public abstract void getSalary();  
}

|  |
| --- |
| abstract class Employees{  void getName() {  System.*out*.println("Saurabh");  }   public abstract void getSalary(); }  public class AnonymousClass {   public static void main(String[] args) {   Employees e = new Employees(){  @Override  public void getName(){  System.*out*.println("Nikit");  }   public void getSalary(){  System.*out*.println("10000");  }   };    e.getName();  e.getSalary();    } } |

## Static nested class

In Java, a static nested class is a class that is defined as a static member of another class. It is similar to a top-level class, but it is enclosed within the namespace of the outer class and has access to its static members.

Here's an example of a static nested class:

public class OuterClass {

private static int x = 5;

public static class InnerClass {

public void innerMethod() {

System.out.println("x = " + x); // Accesses the static member of the outer class

}

}

}

In this example, OuterClass contains a private static member variable x and a public static nested class InnerClass. The innerMethod() method of InnerClass accesses the static member x of the outer class.

To create an instance of InnerClass, you simply declare and instantiate it as you would with any other class:

OuterClass.InnerClass inner = new OuterClass.InnerClass();

You can then call the innerMethod() method of InnerClass:

inner.innerMethod();

This will output the following message to the console:

x = 5

Note that because the static nested class is a static member of the outer class, you can access it using the dot notation OuterClass.InnerClass. You can also create an instance of InnerClass without needing an instance of the outer class, since it is a static member and does not depend on any instance of the outer class.

Static class must be called in static reference.

|  |
| --- |
| class Outer {   static class Inner {   void method(){  System.*out*.println("Inner");  }  }  }  public class StaticInnerClass {  public static void main(String[] args) {   *//non static inner class, inner class  //Outer o = new Outer();  //Outer.Inner innerObject = o.new Inner();  //innerObject.method();   //static inner class  //Outer.Inner object = new Outer.Inner();  //object.method();* } } |

## Builder Design Pattern

The Builder Design Pattern is a creational pattern used in object-oriented programming to create complex objects with many optional parameters. It allows the construction of objects step by step, with finer control over the initialization process, while also improving code readability and reducing the number of constructors and constructor parameters.

The key idea behind the Builder Design Pattern is to separate the construction of a complex object from its representation, so that the same construction process can create different representations of the same object. The pattern involves creating a separate Builder class that accepts inputs and builds the object in a step-by-step manner, with a final build() method that returns the fully constructed object.

Here's an example of implementing the Builder Design Pattern for a Car class:

public class Car {

private String make;

private String model;

private int year;

private int mileage;

private boolean hasSunroof;

private boolean hasLeatherSeats;

private Car(CarBuilder builder) {

this.make = builder.make;

this.model = builder.model;

this.year = builder.year;

this.mileage = builder.mileage;

this.hasSunroof = builder.hasSunroof;

this.hasLeatherSeats = builder.hasLeatherSeats;

}

// getters for all properties

public static class CarBuilder {

private String make;

private String model;

private int year;

private int mileage;

private boolean hasSunroof;

private boolean hasLeatherSeats;

public CarBuilder(String make, String model, int year) {

this.make = make;

this.model = model;

this.year = year;

}

public CarBuilder setMileage(int mileage) {

this.mileage = mileage;

return this;

}

public CarBuilder setHasSunroof(boolean hasSunroof) {

this.hasSunroof = hasSunroof;

return this;

}

public CarBuilder setHasLeatherSeats(boolean hasLeatherSeats) {

this.hasLeatherSeats = hasLeatherSeats;

return this;

}

public Car build() {

return new Car(this);

}

}

}

In this example, the Car class has private properties for make, model, year, mileage, hasSunroof, and hasLeatherSeats. It also has a private constructor that accepts a CarBuilder object, which is responsible for setting the properties of the Car object.

The Car class also has a static inner class called CarBuilder, which has setters for each of the optional properties of the Car object. The CarBuilder class also has a build() method that creates a new instance of the Car class with the given properties.

Using this design pattern, you can create a Car object like this:

Car car = new Car.CarBuilder("Honda", "Civic", 2022)

.setMileage(10000)

.setHasSunroof(true)

.setHasLeatherSeats(false)

.build();

This creates a new Car object with the given properties using the CarBuilder class. You can add or remove properties as needed, and the Car object will be constructed with only the properties that are set.

## Singleton class using Bill Pugh design pattern

The Bill Pugh Singleton Design Pattern is a way to implement a singleton class in Java. This pattern is also known as the "Initialization on Demand Holder (IODH)" pattern.

The Bill Pugh Singleton Design Pattern creates a singleton instance of the class lazily, only when it is needed, while also ensuring thread safety. It does this by using a private inner static class to hold the singleton instance. The singleton instance is created only when the inner class is loaded, which is guaranteed to be thread-safe by the Java Language Specification.

Here's an example of implementing a Singleton class using the Bill Pugh Singleton Design Pattern:

public class Singleton {

private Singleton() {

// private constructor to prevent instantiation from outside

}

private static class SingletonHolder {

private static final Singleton INSTANCE = new Singleton();

}

public static Singleton getInstance() {

return SingletonHolder.INSTANCE;

}

}

In this example, the Singleton class has a private constructor to prevent instantiation from outside the class. It also has a private static inner class called SingletonHolder, which holds the instance of the Singleton class. The static initializer of the inner class initializes the instance variable only once when the inner class is loaded by the JVM.

The getInstance() method of the Singleton class returns the singleton instance held by the SingletonHolder inner class. The SingletonHolder class is only loaded on demand, when the getInstance() method is called for the first time. This way, the Singleton instance is created lazily and only when it is needed.

Using this design pattern, you can obtain a Singleton instance like this:

Singleton singleton = Singleton.getInstance();

This creates a new Singleton instance if it does not already exist, or returns the existing instance if it does. The Singleton instance is created only when it is first needed and is thread-safe, thanks to the use of the private static inner SingletonHolder class.

## Summary

Here are some of the key points from this chapter.

**Inner Classes**

❑ A "regular" inner class is declared inside the curly braces of another class, but

outside any method or other code block.

❑ An inner class is a full-fledged member of the enclosing (outer) class, so it

can be marked with an access modifier as well as the abstract or final

modifiers. (Never both abstract and final together— remember that

abstract must be subclassed, whereas final cannot be subclassed).

❑ An inner class instance shares a special relationship with an instance of the

enclosing class. This relationship gives the inner class access to all of the

outer class's members, including those marked private.

❑ To instantiate an inner class, you must have a reference to an instance of the

outer class.

❑ From code within the enclosing class, you can instantiate the inner class

using only the name of the inner class, as follows:

MyInner mi = new MyInner();

❑ From code outside the enclosing class's instance methods, you can

instantiate the inner class only by using both the inner and outer class names,

and a reference to the outer class as follows:

MyOuter mo = new MyOuter();

MyOuter.MyInner inner = mo.new MyInner();

❑ From code within the inner class, the keyword this holds a reference to

the inner class instance. To reference the outer this (in other words, the

instance of the outer class that this inner instance is tied to) precede the

keyword this with the outer class name as follows: MyOuter.this;

Method-Local Inner Classes

❑ A method-local inner class is defined within a method of the enclosing class.

❑ For the inner class to be used, you must instantiate it, and that instantiation

must happen within the same method, but after the class definition code.

❑ A method-local inner class cannot use variables declared within the method

(including parameters) unless those variables are marked final.

❑ The only modifiers you can apply to a method-local inner class are abstract

and final. (Never both at the same time, though.)

**Anonymous Inner Classes**

❑ Anonymous inner classes have no name, and their type must be either a

subclass of the named type or an implementer of the named interface.

❑ An anonymous inner class is always created as part of a statement; don't

forget to close the statement after the class definition with a curly brace. This

is a rare case in Java, a curly brace followed by a semicolon.

❑ Because of polymorphism, the only methods you can call on an anonymous

inner class reference are those defined in the reference variable class (or

interface), even though the anonymous class is really a subclass or implementer of the reference variable type.

❑ An anonymous inner class can extend one subclass or implement one

interface. Unlike non-anonymous classes (inner or otherwise), an anonymous

inner class cannot do both. In other words, it cannot both extend a class and

implement an interface, nor can it implement more than one interface.

❑ An argument-defined inner class is declared, defined, and automatically

instantiated as part of a method invocation. The key to remember is that the

class is being defined within a method argument, so the syntax will end the

class definition with a curly brace, followed by a closing parenthesis to end

the method call, followed by a semicolon to end the statement: });

**Static Nested Classes**

❑ Static nested classes are inner classes marked with the static modifier.

❑ A static nested class is not an inner class, it's a top-level nested class.

❑ Because the nested class is static, it does not share any special relationship

with an instance of the outer class. In fact, you don't need an instance of the

outer class to instantiate a static nested class.

❑ Instantiating a static nested class requires using both the outer and nested

class names as follows:

BigOuter.Nested n = new BigOuter.Nested();

❑ A static nested class cannot access non-static members of the outer class,

since it does not have an implicit reference to any outer instance (in other

words, the nested class instance does not get an outer this reference).

## Self-Test

The following questions will help you measure your understanding of the dynamic and life-altering

material presented in this chapter. Read all of the choices carefully. Take your time. Breathe.

1. Which are true about a static nested class? (Choose all that apply.)

A. You must have a reference to an instance of the enclosing class in order to instantiate it

B. It does not have access to non-static members of the enclosing class

C. Its variables and methods must be static

D. If the outer class is named MyOuter, and the nested class is named MyInner, it can be

instantiated using new MyOuter.MyInner();

E. It must extend the enclosing class

2. Given:

class Boo {

Boo(String s) { }

Boo() { }

}

class Bar extends Boo {

Bar() { }

Bar(String s) {super(s);}

void zoo() {

// insert code here

}

}

Which create an anonymous inner class from within class Bar? (Choose all that apply.)

A. Boo f = new Boo(24) { };

B. Boo f = new Bar() { };

C. Boo f = new Boo() {String s; };

D. Bar f = new Boo(String s) { };

E. Boo f = new Boo.Bar(String s) { };

3. Which are true about a method-local inner class? (Choose all that apply.)

A. It must be marked final

B. It can be marked abstract

C. It can be marked public

D. It can be marked static

E. It can access private members of the enclosing class

4. Given:

1.public class TestObj {

2. public static void main(String[] args) {

3. Object o = new Object() {

4. public boolean equals(Object obj) {

5. return true;

6. }

7. }

8. System.out.println(o.equals("Fred"));

9. }

10. }

What is the result?

A. An exception occurs at runtime

B. true

C. Fred

D. Compilation fails because of an error on line 3

E. Compilation fails because of an error on line 4

F. Compilation fails because of an error on line 8

G. Compilation fails because of an error on a line other than 3, 4, or 8

5. Given:

1. public class HorseTest {

2. public static void main(String[] args) {

3. class Horse {

4. public String name;

5. public Horse(String s) {

6. name = s;

7. }

8. }

9. Object obj = new Horse("Zippo");

10. System.out.println(obj.name);

11. }

12. }

What is the result?

A. An exception occurs at runtime at line 10

B. Zippo

C. Compilation fails because of an error on line 3

D. Compilation fails because of an error on line 9

E. Compilation fails because of an error on line 10

6. Given:

public abstract class AbstractTest {

public int getNum() {

return 45;

}

public abstract class Bar {

public int getNum() {

return 38;

}

}

public static void main(String[] args) {

AbstractTest t = new AbstractTest() {

public int getNum() {

return 22;

}

};

AbstractTest.Bar f = t.new Bar() {

public int getNum() {

return 57;

}

};

System.out.println(f.getNum() + " " + t.getNum());

}

}

What is the result?

A. 57 22

B. 45 38

C. 45 57

D. An exception occurs at runtime

E. Compilation fails

7. Given:

1. public class Tour {

2. public static void main(String[] args) {

3. Cathedral c = new Cathedral();

4. // insert code here

5. s.go();

6. }

7. }

8. class Cathedral {

9. class Sanctum {

10. void go() { System.out.println("spooky"); }

11. }

12. }

Which, inserted independently at line 6, compile and produce the output "spooky"? (Choose all

that apply.)

A. Sanctum s = c.new Sanctum();

B. c.Sanctum s = c.new Sanctum();

C. c.Sanctum s = Cathedral.new Sanctum();

D. Cathedral.Sanctum s = c.new Sanctum();

E. Cathedral.Sanctum s = Cathedral.new Sanctum();

8. Given:

class A { void m() { System.out.println("outer"); } }

public class TestInners {

public static void main(String[] args) {

new TestInners().go();

}

void go() {

new A().m();

class A { void m() { System.out.println("inner"); } }

}

class A { void m() { System.out.println("middle"); } }

}

What is the result?

A. inner

B. outer

C. middle

D. Compilation fails

E. An exception is thrown at runtime

9. Given:

public class Car {

class Engine {

// insert code here

}

public static void main(String[] args) {

new Car().go();

}

void go() {

new Engine();

}

void drive() { System.out.println("hi"); }

}

Which, inserted independently at line 5, produce the output "hi"? (Choose all that apply.)

A. { Car.drive(); }

B. { this.drive(); }

C. { Car.this.drive(); }

D. { this.Car.this.drive(); }

E. Engine() { Car.drive(); }

F. Engine() { this.drive(); }

G. Engine() { Car.this.drive(); }

10. Given:

public class City {

class Manhattan {

void doStuff() throws Exception { System.out.print("x "); }

}

class TimesSquare extends Manhattan {

void doStuff() throws Exception { }

}

public static void main(String[] args) throws Exception {

new City().go();

}

void go() throws Exception { new TimesSquare().doStuff(); }

}

What is the result?

A. x

B. x x

C. No output is produced

D. Compilation fails due to multiple errors

E. Compilation fails due only to an error on line 4

F. Compilation fails due only to an error on line 7

G. Compilation fails due only to an error on line 10

H. Compilation fails due only to an error on line 13

11. Given:

public class Navel {

private int size = 7;

private static int length = 3;

public static void main(String[] args) {

new Navel().go();

}

void go() {

int size = 5;

System.out.println(new Gazer().adder());

}

class Gazer {

int adder() { return size \* length; }

}

}

What is the result?

A. 15

B. 21

C. An exception is thrown at runtime

D. Compilation fails due to multiple errors

E. Compilation fails due only to an error on line 4

F. Compilation fails due only to an error on line 5

12. Given:

import java.util.\*;

public class Pockets {

public static void main(String[] args) {

String[] sa = {"nickel", "button", "key", "lint"};

Sorter s = new Sorter();

for(String s2: sa) System.out.print(s2 + " ");

Arrays.sort(sa,s);

System.out.println();

for(String s2: sa) System.out.print(s2 + " ");

}

class Sorter implements Comparator<String> {

public int compare(String a, String b) {

return b.compareTo(a);

}

}

}

What is the result?

A. Compilation fails

B. button key lint nickel

nickel lint key button

C. nickel button key lint

button key lint nickel

D. nickel button key lint

nickel button key lint

E. nickel button key lint

nickel lint key button

F. An exception is thrown at runtime

## Answers:

1.

Answer:

✓ B and D. B is correct because a static nested class is not tied to an instance of the

enclosing class, and thus can't access the non-static members of the class (just as a

static method can't access non-static members of a class). D uses the correct syntax

for instantiating a static nested class.

2

B and C. B is correct because anonymous inner classes are no different from any other

class when it comes to polymorphism. That means you are always allowed to declare a

reference variable of the superclass type and have that reference variable refer to an

instance of a subclass type, which in this case is an anonymous subclass of Bar. Since Bar

is a subclass of Boo, it all works. C uses correct syntax for creating an instance of Boo

3

B and E. B is correct because a method-local inner class can be abstract, although it

means a subclass of the inner class must be created if the abstract class is to be used (so

an abstract method-local inner class is probably not useful). E is correct because a

method-local inner class works like any other inner class—it has a special relationship to

an instance of the enclosing class, thus it can access all members of the enclosing class

4

G. This code would be legal if line 7 ended with a semicolon. Remember that line 3 is a

statement that doesn't end until line 7, and a statement needs a closing semicolon!

5

E. If you use a reference variable of type Object, you can access only those members

defined in class Object.

6

A. You can define an inner class as abstract, which means you can instantiate only

concrete subclasses of the abstract inner class. The object referenced by the variable t

is an instance of an anonymous subclass of AbstractTest, and the anonymous class

overrides the getNum() method to return 22. The variable referenced by f is an instance

of an anonymous subclass of Bar, and the anonymous Bar subclass also overrides the

getNum() method (to return 57). Remember that to create a Bar instance, we need an

instance of the enclosing AbstractTest class to tie to the new Bar inner class instance.

AbstractTest can't be instantiated because it's abstract, so we created an anonymous

subclass (non-abstract) and then used the instance of that anonymous subclass to tie

to the new Bar subclass instance

7

D is correct. It is the only code that uses the correct inner class instantiation syntax.

8

C is correct. The "inner" version of class A isn't used because its declaration comes

after the instance of class A is created in the go() method

9

C and G are correct. C is the correct syntax to access an inner class’s outer instance

method from an initialization block, and G is the correct syntax to access it from

a constructor

10

C is correct. The inner classes are valid, and all the methods (including main()), correctly

throw an Exception, given that doStuff() throws an Exception. The doStuff() in class

TimesSquare overrides class Manhattan's doStuff() and produces no output.

11

B is correct. The inner class Gazer has access to Navel's private static and private

instance variables

12

A is correct, the inner class Sorter must be declared static to be called from the static

method main(). If Sorter had been static, answer E would be correct.