2 Advanced class design

Exam objectives covered in this chapter	What you need to know
[2.1] Identify when and how to apply abstract classes	The design requirements and implications of using abstract classes in your application.
[2.2] Construct abstract Java classes and subclasses	Construction and inheritance with abstract Java classes.
[2.3] Use the static and final keywords	The need for defining static and final members (classes, methods, initializer blocks, and variables). The implications of defining nonstatic/nonfinal members as static/final members, and vice versa.
[2.4] Create top-level and nested classes	The flavors of nested classes—inner, static nested, method local, and anonymous. The design benefits, advantages, and disadvantages of creating inner classes. How each type of nested class is related to its outer class. The access and nonaccess modifiers that can be used with the definition of these classes and their members.
[2.5] Use enumerated types	How to compare enumerated types with regular classes. How to define enums with constructors, variables, and methods. How to define enums within classes, interfaces, and methods. How to override methods of a particular enum constant. Use of variables of enum types—when to use the enum name and when to leave it. Use of enumerated types in `switch` constructs. The default methods available to all enums.

While designing your application, you might need to answer questions like these:

- How do I ensure that a derived class implements an inherited behavior in its own specific manner?
- When do I prevent my class from being extended or methods from being overridden?
- When do I make objects share the same copy of a variable and when do I provide them with their own separate individual copy?
- When do I create an inner class to perform a set of related tasks and when do I let the top-level class handle it?
- How do I define constants by using enums?

Design decisions require insight into the benefits and pitfalls of multiple approaches. When armed with adequate information, you can select the best practices and approaches to designing your classes and application.

The topics covered in this chapter will help you answer the aforementioned questions. I'll take you through examples and give you multiple choices to help you determine the best option for designing your classes. This chapter covers

- Abstract classes
- Keywords `static` and `final`
- Enumerated types
- Nested and inner classes

EXAM TIP Take note of the relationship between an exam objective heading and its subobjectives. For example, the topics of using the `static` and `final` keywords, using enumerated types, and creating top-level and nested classes are included within the main objective *advanced class design*. So, apart from using the correct syntax of all of these, the exam will query you on the impact of their use on the design of a class and an application.

Let's start with the first exam objective in this chapter, identifying when and how to apply abstract classes.

2.1 Abstract classes and their application

[2.1] Identify when and how to apply abstract classes

[2.2] Construct abstract Java classes and subclasses

Imagine you're asked to bring a bouquet of flowers. Because no particular flower is specified, you can choose any flower. How is the term *flower* used here? It communicates a group of properties and behavior, which are applicable to multiple types of flowers. Flowers like tulip, rose, hibiscus, and lotus, though *similar,* are also *unique*. In this example, you can compare the term *flower* to an abstract class: the term captures basic properties and behavior, yet enforces individual flower types to implement *some* of that behavior in a unique manner—all flowers *must* have petals, though of different size, color, or shape.

In this section, we'll focus on how the exam will test you on identifying abstract classes, and understanding their need, construction, use, and application. We'll also cover the dos and don'ts of creating abstract classes. For the exam, it's important to compare the similarities and differences of abstract classes and concrete classes. These differences affect the creation and use of these classes. Let's start by identifying abstract classes.

2.1.1 Identify abstract classes

An abstract class *is* an incomplete class or *is considered to be* incomplete. You define it by using the keyword `abstract`. You can't instantiate an abstract class, but you can subclass it to create abstract or concrete derived classes. The choice of defining an abstract class depends on the application context in which the classes are created; it all depends on the details that you need for a class in an application. For example, a class `Animal` might be defined as an abstract class in one application but not in another.

Imagine that you need to create a simple application, GeoAnimals, which helps young children identify a predefined set of common wild animals, while including basic information like the food the animals eat and their habitat. Figure 2.1 shows (a

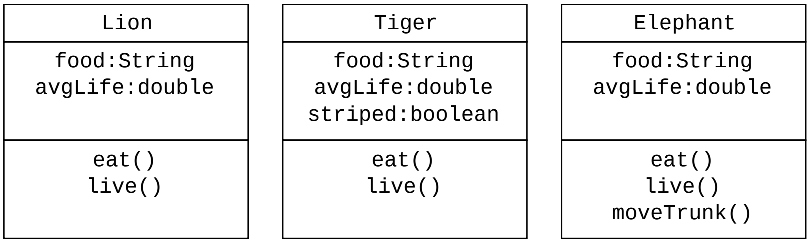


Figure 2.1 Classes `Lion`, `Tiger`, and `Elephant` identified for creating the application GeoAnimals

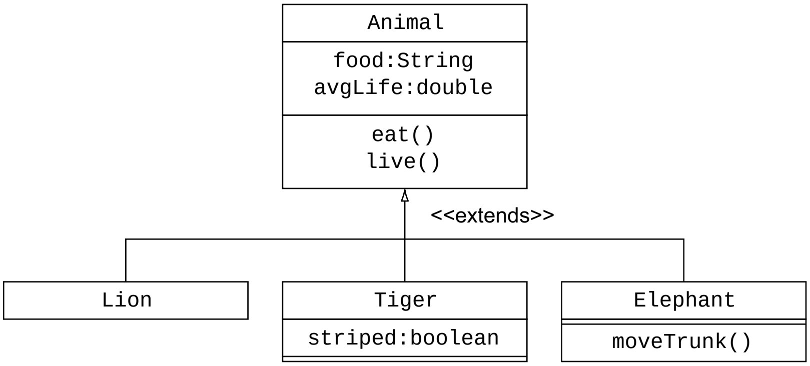


Figure 2.2 Classes `Lion`, `Tiger`, and `Elephant` inherit class `Animal`.

few) classes—`Lion`, `Elephant`, and `Tiger`—that you might identify for this application. I deliberately limited the number of classes to keep the example simple.

As you can see, classes `Lion`, `Tiger`, and `Elephant` have common attributes and behavior. Let's pull out another generic class—say, `Animal`—and make the rest of the classes extend it. Figure 2.2 shows the new arrangement.

Now the big question: Do you need to define the base class `Animal` as an abstract class? How can you determine this? You can ask yourself simple questions to answer the big question:

- Should my application be allowed to create instances of the generic class `Animal`? If no, define class `Animal` as an abstract class.
- Does class `Animal` include behavior that's common to all its derived classes, but can't be generalized (*must* it be implemented by the derived classes in their own specific manner)? If yes, define the relevant method as an abstract method and class `Animal` as an abstract class.

In this sample application, you'd *never* need objects of class `Animal` because they would always refer to a *specific* type of animal. So class `Animal` qualifies to be defined as an abstract class. Also, *eating* behavior, though common to all the animals, is unique to every specific animal. So method `eat()` is a perfect candidate to be defined as an abstract method. Figure 2.3 shows the new arrangement, where class `Animal` is defined as an abstract class, and method `eat()` is defined as an abstract method. Now all the derived classes must implement method `eat()`.

Because an abstract class is meant to be extended by other classes, and its abstract methods are meant to be implemented, it's recommended that you document its expected behavior in your real-life projects. This documentation will enable your class to be inherited and used appropriately.

EXAM TIP An abstract method doesn't define an implementation. It enforces all the concrete derived classes to implement it.

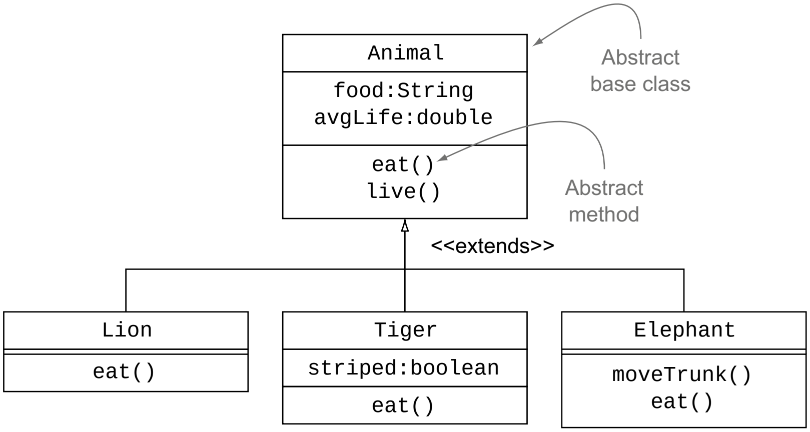


Figure 2.3 Abstract class `Animal` defines an abstract method `eat()` and is inherited by classes `Lion`, `Tiger`, and `Elephant`.

Until this point, you've looked at how to identify an abstract class. Does that imply that if you were to define class `Animal` in another application, you'd define it as an abstract class? Not always. For example, an application that counts all living beings, categorized as humans, animals, and plants, might not need to define class `Animal` as an abstract class because the application might not need to create its specific types; it needs only a count of the total animals. If you need to store the *type* of an animal, class `Animal` can define an attribute—say, `species`. This arrangement is shown in figure 2.4.

Another frequently asked question by new programmers or designers is, when is an abstract base class fit to be defined as an interface? Interfaces can be defined only when no implementation of any method is provided. Also, an interface can define only constants, which can't be reassigned another value by the implementing classes. The base class `Animal` discussed previously can't be defined as an interface; it can't define its attributes `food` and `avgLife` as constants. As an example of an interface, the Java package `java.util` contains multiple interfaces, such as `List`. The interface `List` defines multiple methods, which must be implemented by all the implementing classes, such as `ArrayList`.

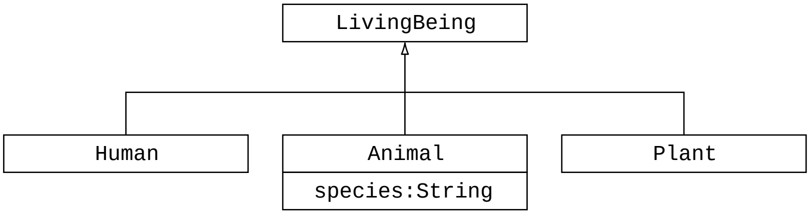


Figure 2.4 Class `Animal` need not be always defined as an abstract class in *all* applications.

NOTE Starting with Java 8, an interface can define a default implementation of its methods, so an implementing class might *not* necessarily override these methods. But this exam is based on Java 7 and I'll continue to refer to an interface as the one that can't define method implementation.

Now that you understand how to identify abstract classes, let's look at how to construct abstract classes and their subclasses, and how to apply them. On the exam, you'll be questioned on correct construction of abstract classes, their subclasses, and their dos and don'ts.

2.1.2 Construct abstract classes and subclasses

To keep the code small, let's code the abstract class `Animal` and only two of its derived classes, `Lion` and `Elephant`:

```
abstract class Animal {
    protected String food;
    protected double avgLife;

    Animal(String food, double avgLife) {
        this.food = food;
        this.avgLife = avgLife;
    }

    abstract void eat();

    void live() {
        System.out.println("Natural habitat : forest");
    }
}
```

❶ Abstract base class Animal

❷ Properties of class Animal

❸ Constructor

❹ Abstract method eat()

❺ Nonabstract method live()

At ❶, abstract class `Animal` is defined by prefixing the class definition with the keyword `abstract`. The code at ❷ defines attributes to store values for food and average life span: `food` and `avgLife`. The code at ❸ defines a constructor for class `Animal`. You can't instantiate an abstract class, but you can create its constructors, including overloaded constructors. At least one of the `Animal` constructors must be called by instances of its derived classes. The code at ❹ defines abstract method `eat()`, which delegates the responsibility of implementing it to the derived classes. You can define class `Animal` as an abstract class, even if doesn't define any abstract methods. The code at ❺ defines method `live()`, with an implementation. If required, it can be overridden by the derived classes.

EXAM TIP It isn't obligatory for abstract classes to define abstract methods. *Abstract methods* must not define a body.

Following is the definition of derived class `Lion`:

```
class Lion extends Animal{
    Lion(String food, double avgLife) {
        super(food, avgLife);
    }
    void eat() {
        System.out.println("Lion-hunt " + food);
    }
}
```

❶ **Constructor**

❷ **Implement method eat()**

Class `Lion` extends the base class `Animal`. It defines a constructor at ❶, which accepts a `double` value for average life and a `String` value for food and passes it on to its base class's constructor. At ❷, class `Lion` implements method `eat()`. Let's now define class `Elephant`:

```
class Elephant extends Animal{
    Elephant(String food, double avgLife) {
        super(food, avgLife);
    }

    void eat() {
        System.out.println("Elephant-method eat");
    }

    void moveTrunk() {
        System.out.println("Elephant-method moveTrunk");
    }
}
```

❶ **Constructor**

❷ **Implement method eat()**

❸ **New method moveTrunk()**

At ❶, class `Elephant` defines a constructor that calls its base class constructor. At ❷, it implements abstract method `eat()` from the base class. At ❸, it defines a new method, `moveTrunk()`.

EXAM TIP Notice the power of base class constructors to ensure that all derived class constructors pass them a value. The base class `Animal` defines only one constructor that accepts a value for its instance variables `food` and `avgLife`. Because a derived class constructor must call its base class's constructor, classes `Lion` and `Elephant` define a constructor that calls `Animal`'s constructor.

Let's put all these classes to work, in class `GeoAnimals`, as follows:

```
class GeoAnimals{
    Animal[] animals = new Animal[2];

    GeoAnimals() {
        animals[0] = new Lion("Antelope", 20);
        animals[1] = new Elephant("Bananas", 60);
    }
```

❶ **An array of type Animal—base class of Lion and Elephant**

❷ **Initialize array animals with separate instances of Lion and Elephant.**

```
    void flashcards() {
        for (Animal anAnimal : animals) {
            anAnimal.eat();
            anAnimal.live();
        }
    }

    public static void main(String args[]) {
        GeoAnimals myAnimals = new GeoAnimals();
        myAnimals.flashcards();
    }
}
```

❸ **Iterate through objects of array animals, calling methods eat() and live().**

❹ **Create an instance of GeoAnimals and call method flashcards().**

Here's the output of the preceding code (blank lines were added to improve readability):

```
Lion-hunt Antelope
Natural habitat : forest

Elephant-method eat
Natural habitat : forest
```

Let's walk through this code. The code at ❶ declares an array of type `Animal`. Though you can't create instances of abstract class `Animal`, an array of `Animal` can be used to store objects of its derived classes, `Lion` and `Elephant`. The code at ❷ initializes animals with instances of classes `Lion` and `Elephant`. The code at ❸ iterates through the array `animals`, calling methods `eat()` and `live()` on all its elements. The code at ❹ defines method `main()` that creates an object of class `GeoAnimals`, calling method `flashcards()`.

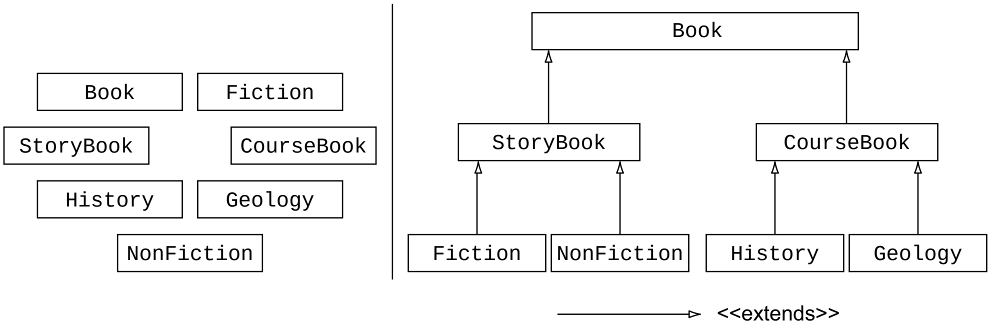
The code in this section walked you through how to create abstract classes and their subclasses, and how to use them. The efficient use of abstract classes lies in their identification in an application. Let's see how well you score on identifying all abstract and concrete classes in an application in the following "Twist in the Tale" exercise.

Twist in the Tale 2.1

The following are names of multiple classes. Your task is to arrange these in an inheritance hierarchy, connecting all base and derived classes. At end of the exercise, all these classes should be connected, with the base class at the top and derived classes below it.

```
      Omnivore    Lion
  Dog       Animal      Tiger
      Cat         Herbivore
  Carnivore Deer        Elephant
```

Here's another example that will help you attempt the preceding exercise.



EXAM TIP An abstract method can't be defined in a concrete class. It can be defined in an abstract class only.

2.1.3 Understand the need for abstract classes

An abstract class represents partial implementation of an object or concept. But why do you need partial implementation? Do abstract classes exist only so other classes can inherit them? These questions are frequently asked by new Java application designers. You might also have to answer these questions on the exam.

You need an abstract class to pull out and group together the common properties and behavior of multiple classes—the same reason you need a nonabstract base class. You define a *base class* as an *abstract class* to prevent creation of its instances. As the creator, you think that it doesn't include enough details to create its own objects. When you define abstract methods in a base class, it *forces* all its nonabstract derived classes to implement the incomplete functionality (abstract methods) in their *own* unique manner.

Because you can't create instances of an abstract class, there's not much sense in creating an abstract base class, which isn't extended by other classes.

EXAM TIP Abstract classes make a point loud and clear: they *force* the concrete derived classes to implement a base class's abstract methods, in their own unique manner.

Note that I haven't discussed the need for, advantages of, or disadvantages of creating nonabstract base classes in this section. This section specifically covers base classes that are abstract.

2.1.4 Follow the dos and don'ts of creating and using abstract classes

Apart from the points covered in the previous section, the exam will likely include other theoretical and coding questions on the dos and don'ts of creating and implementing abstract classes.

DON'T CREATE AN ABSTRACT CLASS ONLY TO PREVENT CREATION OF ITS OBJECTS

To prevent instantiation of a class by using the operator `new`, define all its class constructors as private. For example, class `java.lang.Math` in the Java API doesn't allow creation of its objects by defining its constructor as a private member:

```
package java.lang;
public final class Math{
    private Math() { /*code */}
}
```

DON'T MAKE AN ABSTRACT CLASS IMPLEMENT INTERFACES THAT RESULT IN INVALID METHOD IMPLEMENTATION

When a class implements an interface, the class must implement its methods (unless the class is abstract) to meet the contract. But if the class defines methods with the same name as the one defined in the interface, they should either comply with correct method overriding or overloading rules or else the class won't compile. In the following example, class `Animal` can't implement interface `Live`:

```
interface Live{
    boolean eat();
}

abstract class Animal implements Live{
    public abstract void eat();
}
```

Method eat() that returns boolean value

Won't compile; method eat() from Live and Animal can't coexist.

Method eat() doesn't return any value.

Class `Animal` won't compile because method `eat()` from interface `Live` and method `eat()` defined in class `Animal` exist as invalid overloaded methods.

DON'T CREATE OBJECTS OF AN ABSTRACT CLASS

Code that creates objects of an abstract class won't compile:

```
abstract class Animal{}
class Forest {
    Animal animal = new Animal();
}
```

Won't compile; can't instantiate abstract classes.

DON'T DEFINE AN ABSTRACT CLASS AS A FINAL CLASS

A final class can't be extended. On the other hand, abstract classes are created so they can be extended by other classes. Hence, abstract classes can't be defined as final classes.

```
abstract final class Animal {}
```

Won't compile

Don't force an abstract class to implement all methods from the interface(s) it implements

An abstract class can implement multiple interfaces. It might not implement *all* the abstract methods from the implemented interface(s), leaving them to be implemented by all its nonabstract derived classes:

```
interface Live{
    void eat();
}
abstract class Animal implements Live{}
```

Abstract class Animal doesn't implement eat() from interface Live.

Do use an object of an abstract class to refer to objects of its nonabstract derived classes

An abstract class can't be instantiated. But this doesn't stop you from using a reference variable of an abstract base class to refer to an instance of its nonabstract derived class:

```
abstract class Animal{}
class Deer extends Animal{}
class Forest{
    Animal animal = new Deer();
}
```

Abstract class variable can refer to instance of its derived class

Comparing an abstract class with a concrete class is obvious, as covered in the next section. This comparison will help you with multiple exam objectives: identifying abstract classes, their construction, and their application.

2.1.5 Compare abstract classes and concrete classes

Do you think the constructor of an abstract base class is called in the same manner as that of a concrete base class? Yes, indeed. Table 2.1 answers many more questions like this by comparing abstract and concrete classes.

Table 2.1 Comparing an abstract class with a concrete class

Comparison Category	Abstract class	Concrete class
Create a new type	✓	✓
Use as base class	✓	✓
Extend another class	✓	✓
Implement interfaces	✓	✓
Define attributes and concrete methods	✓	✓
Require at least one constructor to be called by its derived classes	✓	✓
Define abstract methods	✓	✗
Allow object creation	✗	✓

Before moving on to the next section, let's quickly list the points to remember about abstract classes for the exam.

Rules to remember for creating abstract classes

- An abstract class must be defined by using the keyword `abstract`.
- An abstract class can extend any other abstract or concrete class and implement other interfaces.
- An abstract class can define multiple constructors.
- An abstract class can define instance and static variables.
- An abstract class can define instance and static methods.
- An abstract class might not necessarily define an abstract method and can exist without any abstract method.
- A class can't define an abstract static method.
- Don't create an abstract class just to prevent creation of its instances.
- Don't make an abstract class implement interfaces that result in incorrect overloaded or overridden methods.

Rules to remember for subclassing an abstract class

- A concrete subclass must implement all the abstract methods in its abstract superclass(es).
- An abstract subclass might not implement all the abstract methods in its abstract superclass(es).
- A subclass must call at least one constructor from its superclass.

Identification of abstract classes is an important design decision. It changes how other classes might use the abstract class. Similarly, creating classes that can't be extended, creating methods that can't be overridden, or creating class (or static) members are other important design decisions. In the next section, you'll see how you can do so by using the `static` and `final` nonaccess modifiers.

2.2 Static and final keywords

[2.3] Use the static and final keywords

Defining a class as a final class prevents it from being extended. Similarly, a static variable or method can be accessed without instances of its class; the variable or method is available after its class is loaded into memory. These are a few examples of how the nonaccess modifiers `static` and `final` change the default behavior of a Java entity. For the exam, you need to understand the need for defining static and final members (classes, methods, initializer blocks, and variables) together with their correct definition and use. You also need to know the implications of defining nonstatic/nonfinal members as static/final members, and vice versa.

2.2.1 Static modifier

You can define variables, methods, nested classes, and nested interfaces as static members. They belong to a class and not to instances. They can be accessed soon after their class is loaded into memory. Top-level classes, interfaces, and enums can't be defined as static entities. Watch out for code that declares top-level classes, interfaces, and enums as static members. Such code won't compile. Let's get started with static class variables.

Static variables

Static variables belong to a class and are shared by all its instances. Their value is the same for all instances of their class. A static class variable is created when its class is loaded into memory by the JVM. It can exist and is accessible even if no instances of the class exist. So you can use it to perform operations that span multiple instances of a class. Class `Book` defines a static class variable `bookCount`, to count the instances of class `Book` that are created while your program is running:

```
class Book {
    static int bookCount;
    public Book() {
        ++bookCount;
    }
}
class Publisher{
    public static void main(String args[]){
        System.out.println(Book.bookCount);
        Book b1 = new Book();
        Book b2 = new Book();
        System.out.println(Book.bookCount);
    }
}
```

Static variable bookCount

bookCount is incremented in constructor

❶ **Prints "0"**

❷ **Prints "2"**

Assuming that no instances of `Book` were created earlier, the code at ❶ prints `0`. The code at ❷ prints 2 due to creation of two instances of class `Book`, created on the preceding lines. Each invocation of the construction increments the value of the static class variable `bookCount` by 1.

> **EXAM TIP** Unlike instance variables, which are initialized for each instance, static class variables are initialized only once, when they are loaded into memory. The default variable values are `false` for `boolean`; `'\u0000'` for `char`; `0` for `byte`; `short`, `int`, `0L` for `long`; `0.0F` for `float`; `0.0D` for `double`; and `null` for objects.

Because the same value of a static class variable is shared across all the instances of a class, if modified, the same modified value is reflected across all instances. In the following code, the value of the static class variable bookCount is accessed and modified using the class name Book and instances b1 and b2 (modifications in bold):

```
class Book {
    static int bookCount;
    public Book() {
        ++bookCount;
    }
}
class Publisher{
    public static void main(String args[]){
        System.out.println(Book.bookCount);
        Book b1 = new Book();
        Book b2 = new Book();
        System.out.println(Book.bookCount);
        b1.bookCount = 10;
        System.out.println(b2.bookCount);
    }
}
```

Prints "0" (access bookCount using class name Book)

Set value of bookCount to 10, using reference variable b1.

Prints "10" (access bookCount using reference variable b2)

NOTE For simplicity, I've defined the variable bookCount with default access, which is directly accessed and manipulated outside the class Book. This isn't a recommended approach in real-life projects. Encapsulate your data by defining the class and instance variables as private and make them accessible outside their class through accessor and mutator methods.

On the exam, you're likely to see code that accesses a static class variable by using the name of its class and its instances. Although a static class variable is allowed to be accessed by using instances of a class, it's not a preferred approach; it makes the static class variable seem to belong to an instance, which is incorrect. Always refer to a static class member by using its class name.

EXAM TIP You can access a static member by using the name of its class or any of its instances. All these approaches refer to the same static member. The preferred approach is to use a class name; otherwise, a static member *seems* to be tied to an instance, which is incorrect.

A combination of the static and final nonaccess modifiers is used to define *constants* (variables whose value can't change). In the following code, the class Emp defines the constants MIN_AGE and MAX_AGE:

```
class Emp {
    public static final int MIN_AGE = 20;
    static final int MAX_AGE = 70;
}
```

Constant MIN_AGE

Constant MAX_AGE

Static methods

Static methods don't need instances of a class. They can be called even if no instance of the class exists. You define static methods to access or manipulate static class variables. The static methods can't access nonstatic fields or nonstatic methods. Referring to the example of class `Book`, which used the static variable `bookCount` to count all instances of class `Book`, static methods `getBookCount()` and `incrementBookCount()` can be created to access `bookCount` and manipulate it:

```
class Book {
    private static int bookCount;
    public static int getBookCount(){
        return bookCount;
    }

    public void incrementBookCount() {
        ++bookCount;
    }
}
```

Static method to retrieve value of static variable bookCount

Static method to increment value of static variable bookCount

You also use static methods to define *utility methods*—methods that *usually* manipulate the method parameters to compute and return an appropriate value:

```
static double average(double num1, double num2, double num3) {
    return(num1+num2+num3)/3;
}
```

A static method might not always define method parameters. For example, the method `random` in class `java.lang.Math` doesn't accept any parameters. It returns a pseudorandom number, greater than or equal to `0.0` and less than `1.0`.

EXAM TIP A static method is used to manipulate static class variables or to define utility methods. A utility method may or may not accept method parameters.

What can a static method access?

Neither static class methods nor static class variables can access the nonstatic instance variables and instance methods of a class. But the reverse is true: nonstatic variables and methods can access static variables and methods because the static members of a class exist even if no instances of the class exist. Static members are forbidden from accessing instance methods and variables, which can exist only if an instance of the class is created.

Examine the following code:

```
class MyClass {
    static int x = count();
    int count() { return 10; }
}
```

Compilation error

This is the compilation error thrown by the previous class:

```
MyClass.java:3: nonstatic method count() cannot be
                            referenced from a static context
    static int x = count();
                   ^
1 error
```

The following code is valid:

```
class MyClass {
    static int x = result();
    static int result() { return 20; }
    int nonStaticResult() { return result(); }
}
```

Static variable referencing a static method

Nonstatic method using a static method

You can use constructors or instance initializer blocks to initialize the instance variables. But how can you initialize the static variables, after they're loaded into memory? *Static initializer blocks* are the answer.

STATIC INITIALIZER BLOCKS

A *static initializer* block is a code block defined using braces and prefixed by the keyword `static`:

```
static {
    //code to initialize static variables
}
```

Because static variables can't be initialized using the constructors of a class, a static initializer block is used to initialize static variables. This initializer block executes when a class is loaded by the JVM into memory. You can define multiple static initializer blocks in your code, which execute in the order of their appearance. All types of statements are allowed in this block, including declaration, initialization, assignment, and calling of other static variables and methods. In the following example, class `AffiliateProgram` defines a static variable `accountOpenBonus`. The variable `accountOpenBonus` is initialized using a static initializer block:

```
class AffiliateProgram {
    private static int accountOpenBonus;
    static {
        accountOpenBonus = 5;
    }
}
```

Declare static variable

Initialize static variable using a static initializer block

You might argue that you could initialize the static variable `accountOpenBonus` as follows:

```
class AffiliateProgram {
    private static int accountOpenBonus = 5;
}
```

Declare and initialize a static variable

But what happens if you need to use a calculated value or initialize the value of accountOpenBonus based on the outcome of a condition:

```
class AffiliateProgram {
    private static int accountOpenBonus;
    static {
        if (/* file XYZ exists */)
            accountOpenBonus = 5;
        else
            accountOpenBonus = 15;
    }
}
```

Conditional assignment of variable accountOpenBonus

Again, you might argue that you can move the preceding conditional execution to a static method and use it to initialize variable accountOpenBonus, without using the static initializer block:

```
class AffiliateProgram {
    private static int accountOpenBonus = initAccountOpenBonus();

    private static int initAccountOpenBonus() {
        if (/* file XYZ exists */)
            return 5;
        else
            return 15;
    }
}
```

Conditional assignment of variable accountOpenBonus using static method

What happens if method initAccountOpenBonus() throws a checked exception, say, FileNotFoundException? In this case, you *must* use a static initializer block to assign the returned value from initAccountOpenBonus() to variable accountOpenBonus. As you know, if a method throws a checked exception, its use should either be enclosed within a try block or the method that uses it should declare the exception to be thrown. In this case, neither is possible; the declaration of the variable accountOpenBonus can't be enclosed within a try block because this statement doesn't exist within a method or code block. Here's the relevant code:

Won't compile; can't declare and initialize variable using method that throws checked exception

```
import java.io.*;
class AffiliateProgram {
    private static int accountOpenBonus = initAccountOpenBonus();
    private static int initAccountOpenBonus() throws FileNotFoundException
    {
        //relevant code
    }
}
```

And here's the way out:

```
class AffiliateProgram {
    private static int accountOpenBonus;
    private static int initAccountOpenBonus()
                                      throws FileNotFoundException{
        //..relevant code
    }
    static {
        try {
            accountOpenBonus = initAccountOpenBonus();
        }
        catch (FileNotFoundException e) {
            //.. relevant code
        }
    }
}
```

Static variable

Static method that throws checked exception

static initializer block

try block to catch FileNotFoundException thrown by initAccountOpenBonus()

Another reason for the existence of a static initializer block is to add values (static or dynamic) to collection objects that have already been initialized. Here's an example:

```
class AddValuesToStaticVariables {
    static private String[] dataStores = new String[5];
    static {
        dataStores[0] = "us.ny";
        dataStores[1] = "jp.tk";
        dataStores[2] = "gr.br";
        //..code that assigns dynamic value to dataStores[]
    }
}
```

String array dataStores is initialized.

Add explicit values to dataStores.

Pull values from the database and add to String array dataStores.

The static initializer blocks can be tricky and cumbersome to work with when it comes to debugging them. On the exam, beware of code that defines multiple initializer blocks. If a class defines multiple initializer blocks, they execute in the order of their appearance in a class. Let's examine output of code that defines multiple initializer blocks:

```
class StaticInitBlocks {
    static int staticVar = 10;
    static {
        System.out.println("First");
        ++staticVar;
    }
    static {
        System.out.println("Second");
        ++staticVar;
    }

    static void modifyStaticVar() {
        ++staticVar;
    }
    public StaticInitBlocks() {
        System.out.println("Constructor:" + staticVar);
    }

    public static void main(String args[]) {
        new StaticInitBlocks();
        modifyStaticVar();
        new StaticInitBlocks();
    }
}
```

Code in a static initializer block executes when a class is loaded in memory by JVM—before creation of its instances. The output of the preceding code is as follows:

```
First
Second
Constructor:12
Constructor:13
```

Can the static and instance initializer blocks access the static or instance variables of a class, like other methods? Let me modify the preceding code and use it for the next "Twist in the Tale" exercise.

Twist in the Tale 2.2

Following is modified code for class `DemoMultipleStaticBlocks`. Answer the question before you execute it on your system. Which answer correctly shows its output?

```
class DemoMultipleStaticBlocks {
    static {
        ++staticVar;
    }
    static int staticVar ;
    static {
        ++staticVar;
    }
    public DemoMultipleStaticBlocks() {
        System.out.println("Constructor:" + staticVar);
    }
    public static void main(String args[]) {
        new DemoMultipleStaticBlocks();
    }
}
```

- **a** `Constructor: 2`
- **b** `Constructor: 1`
- **c** `Constructor: 0`
- **d** Compilation error
- **e** Runtime exception

EXAM TIP On the exam, beware of code that defines multiple initializer blocks. If a class defines multiple initializer blocks, they execute in the order of their appearance in a class.

Watch out for another combination on the exam: initialization of a static class variable and its manipulation in a static block. What do you think is the order of execution in the following code? Will the following example code print 1 or 11?

```
public class AssignManipulateStaticVariable {
    static {
        rate = 10;
    }
    static int rate = 0;

    static {
        ++rate;
    }
    public AssignManipulateStaticVariable() {
        System.out.println(rate);
    }
    public static void main(String args[]) {
        new AssignManipulateStaticVariable();
    }
}
```

First static initializer block to assign 10 to rate

Declare static variable rate and assign 0 to it.

Second static initializer block to increment rate by 1

Prints "1"

For the preceding code, the compiler rearranges the code to execute. It declares the static variable `age` and then picks up the code from all the static initializer blocks and assignment of `age` and combines them in a single static initializer block, in the order of their occurrence, as follows:

```
static int rate;
static
{
    rate = 10;
    rate = 0;
    ++rate;
}
```

The preceding code explains why `AssignManipulateStaticVariable` prints `1` and not `11`.

STATIC CLASSES AND INTERFACES

Let's look at other types of static entities: static classes and interfaces. These are also referred to as *nested classes, static nested classes, static interfaces,* and *static nested interfaces.* You can't prefix the definition of a top-level class or an interface with the keyword `static`. A top-level class or interface is one that isn't defined within another class or interface. The following code fails to compile:

```
static class Person {}
static interface MyInterface {}
```

But you can define a class and an interface as a static member of another class. The following code is valid:

```
class Person {
    static class Address {}
    static interface MyInterface {}
}
```

Static nested class

Static nested interface

As you know, the static variables and methods of a class are accessible without the existence of any of its objects. Similarly, you can access a static class without an object of its outer class. You'll learn all about the other details of the static classes in section 2.3.2 .

2.2.2 Nonaccess modifier—final

The decision to apply the nonaccess modifier `final` to a class, interface, variable, or method is an important class design decision. To start, should you define your class as a final class? Yes, if you don't want it to be subclassed. Should you define your method as a final method? Yes, if you don't want any of its subclasses to override it. Do you want to define a variable as a final variable? Yes, if after the variable is initialized, you don't want it to be reassigned another value. Knowing these details will enable you to make the right decisions—when, why, where, and how to apply the nonaccess modifier `final` and when not to. Apart from testing you on how to use the modifier `final` in code, the exam will also query you on the implications of its use on the design or behavior of code. Let's start with final variables.

FINAL VARIABLES

The final variables can be initialized only *once*. You can tag all types of variables—static variables, instance variables, local variables, and method parameters—with the nonaccess modifier `final`. Because of the differences in how these variable types are initialized, they exhibit different behavior. Let's start with defining a static variable as a final variable:

```
class TestFinal {
    static final int staticFinal = 10;
}
```

A final static class variable can be initialized with its declaration, or by using a static initializer block, which is *guaranteed* to execute only once for a class. Because a static method can be called multiple times, it can't define code to initialize a final (static) variable:

```
class TestFinal {
    static final int staticFinal2 = 12345;

    static final int staticFinal;

    static {
        staticFinal = 1234;
    }
```

Static final variable initialized with its declaration

Static final variable not initialized

Static initializer block to initialize static variable

```
    static void setStaticFinal(int value) {
        staticFinal = value;
    }
}
```

Won't compile; static method can execute multiple times and so it can't include initialization of final variable.

Because the constructor of a class executes on creation of every instance of the class, you can't initialize a final static variable in the constructor. The following code won't compile:

```
class FinalStatic {
    static final int finalVar;

    FinalStatic() {
        finalVar = 10;
    }
}
```

Final static variable can't be initialized in constructor of a class

Similarly, though you can initialize a final *instance variable* in the class's constructor or its instance initializer block, you can't initialize it in an instance *method*. Instance methods can execute more than once:

```
class InstanceFinalVariables {
    final int finalVar2 = 710;

    final int finalVar;

    InstanceFinalVariables() {
        finalVar = 10;
    }

    void setValue(int a) {
        finalVar = a;
    }
}
```

Instance final variable initialized with its declaration

Instance final variable not initialized

Class's constructor initializes instance final variable

Won't compile; a method may execute multiple times and so can't include code to initialize a final variable.

EXAM TIP If a static or instance variable is marked `final`, it must be initialized, or the code won't compile.

Interestingly, you can survive code with an uninitialized final local variable, if you don't use it:

```
class MyClass {
    void setValue(int a) {
        final int finalLocalVar1;
        finalLocalVar1 = 20;

        final int finalLocalVar2;
    }
}
```

Final local variable declared and initialized on separate lines

Uninitialized final local variable; compiles successfully

In the preceding code, if you try to use the uninitialized final local variable `finalLocalVar2`, your code won't compile. Modified code is as follows:

```
class MyClass {
    void setValue(int a) {
        final int finalLocalVar1;
        finalLocalVar1 = 20;

        final int finalLocalVar2;

        System.out.println(finalLocalVar2);
    }
}
```

Won't compile when you try to use an uninitialized local variable.

Method parameters are initialized when the method is invoked. If a method marks its method parameter(s) as `final`, the method body can't reassign a value to it, as follows:

```
class MyClass {
    void setValue(final int finalMethodParam) {
        finalMethodParam = 10;
    }
}
```

Final method parameter

Won't compile; value can't be assigned to final method parameter.

There's a difference between final primitive variables and final object reference variables. The final primitive variables can't change, but the object referred to by final object reference variables can be changed. Only the final reference itself can't be changed:

```
class MyClass {
    void addCondition(final StringBuilder query) {
        query.append("WHERE id > 500");
        query = new StringBuilder("SELECT name FROM emp");
    }
}
```

Final method parameter

Can modify object referred to by final reference variable query.

Won't compile; can't reassign another object to final reference variable query.

CONDITIONAL ASSIGNMENT OF FINAL VARIABLES

What happens when you initialize a final variable within an `if-else` construct, `switch` statement, or `for`, `do-while`, or `while` loop? In this case, code that assigns a value to the final variable *might* not execute. If the Java compiler is doubtful about the initialization of your final variable, the code won't compile. For example, the constructor of `MyClass` assigns a value to its final instance variable `finalVar` by using an `if` statement, as shown in the following code listing.

Listing 2.1 Conditional assignment of final instance variable in class's constructor

```
class MyClass {
    final int finalVar;
    MyClass(double a, double b) {
            if (a > b)
                   finalVar = 20;
            else if (b >= a)
                   finalVar = 30;
    }
}
```

❶ Assigns 20 to finalVar if a > b

❷ Assigns 30 to finalVar if b >= a

Class `MyClass` fails compilation with the following compilation error message:

```
variable finalVar might not have been initialized
```

The code at ❶ assigns a value to `finalVar` if the condition `a > b` evaluates to `true`. The code at ❷ assigns a value to `finalVar` if condition `b >= a` evaluates to `true`. The compiler has its doubts about being able to execute (and thus initialize) in all conditions. So, the Java compiler will consider initialization of a final variable complete only if the initialization code will execute in *all* conditions. Adding an `else` branch results in successful code compilation:

```
class MyClass {
    final int finalVar;
    MyClass(double a, double b) {
        if (a>b)
            finalVar = 20;
        else
            finalVar = 30;
    }
}
```

❶ Assigns 20 to finalVar if a > b

❷ Assigns 30 to finalVar otherwise

In the preceding code, ❶ initializes `finalVar` to `30` if the condition `a > b` evaluates to `true`. Otherwise, it initializes `finalVar` to `30` at ❷. Let's modify the code in listing 2.1 so it uses constant literal values instead of variables, in `if` conditions:

```
class MyClass {
    final int finalVar;
    MyClass(double a, double b) {
        if (1>2)
            finalVar = 10;
        else if (100>10)
            finalVar = 20;
    }
}
```

Assigns 10 to finalVar if 1 > 2

❶ Assigns 20 to finalVar if 100 > 10

The preceding code compiles successfully, because with the constant values, the compiler can determine that code at ❶ will execute, initializing a value to the final variable for sure.

Let's modify the code again, so it continues to use the variables in the `if` conditions. In listing 2.1, the Java compiler complained that variable `finalVar` might not have been initialized. So let's explicitly assign a value to variable `finalVar`, before the start of the `if` statement, as follows:

```
class MyClass {
    final int finalVar;
    MyClass(double a, double b) {
        finalVar = 100;
        if (a>b)
            finalVar = 20;
        else if (b>=a)
            finalVar = 30;
    }
}
```

❶ Explicit assignment to final variable finalVar

❷ Conditional assignment to final variable finalVar

The code at ❶ initializes `finalVar`, and the code at ❷ tries to assign a value to it, conditionally. Because a final variable can't be reassigned a value, the preceding code fails compilation with the following compilation error message:

```
variable finalVar might already have been assigned
                   finalVar = 20;
                   ^
```

EXAM TIP On the exam, look out for multiple initializations of a final variable. Code snippets that try to reinitialize a final variable won't compile.

The simplest way to initialize a final variable is to do so with its declaration. If not initialized with its declaration, a static final variable can be initialized in the class's static initializer block. An instance final variable can be initialized in its constructor or the instance initializer block. A local final variable can be assigned a value in the method in which it's defined. A final method parameter can't be reassigned a value within the method.

It's time for you to attempt the next "Twist in the Tale" exercise, which tests you on understanding assignment of a base class's final instance variable from the derived class.

Twist in the Tale 2.3

Let's modify the code used in the preceding examples so class `MyClass` does not initialize its final instance variable `finalVar`. This variable is initialized in its derived class, `MyDerivedClass`, as follows:

```
abstract class MyClass {
    public final int finalVar;
}
class MyDerivedClass extends MyClass {
    MyDerivedClass() {
        super();
        finalVar = 1000;
    }
}
```

Your task is to first think about the possible output of the following code before you compile it on your system:

```
class Test {
    {System.out.println(new MyDerivedClass().finalVar);}
    public static void main(String args[]) {
        new Test();
    }
}
```

FINAL METHODS

The final methods defined in a base class can't be overridden by its derived classes. The final methods are used to prevent a derived class from overriding the implementation of a base class's method. Can you think of any scenario where you'd need this? Picture this: the base class of all the Java classes, `java.lang.Object`, defines multiple final methods—`wait`, `notify`, `getClass`. Methods `wait()` and `notify()` are used in threading and synchronization. If the derived classes were allowed to override these methods, how do you think Java would implement threading and synchronization?

If a derived class tries to override a final method from its base class, it won't compile, as follows:

```
class Base {
    final void finalMethod() {}
}
class Derived extends Base {
void finalMethod() {}
}
```

Final method in base class

Won't compile; final method in base class can't be overridden.

The preceding code fails to compile, with the following compilation error message:

```
finalMethod() in Derived cannot override finalMethod() in Base
    final void finalMethod() {}
               ^
  overridden method is final
1 error
```

As you know, you can override only what is inherited by a derived class. The following code compiles successfully, even though the derived class *seems* to override a final method from its base class:

```
class Base {
    private final void finalMethod() {}
}
class Derived extends Base {
    final void finalMethod() {}
}
```

Private methods aren't inherited by derived classes.

Compiles successfully

The base class's private methods aren't inherited by a derived class. In the previous code, method `finalMethod()` is defined as a private method in class `Base`. So it doesn't matter whether it's marked as a final method. Method `finalMethod()` defined in class `Derived` doesn't override the base class's method `finalMethod()`. Class `Derived` defines a new method, `finalMethod()`.

EXAM TIP The private methods of a base class aren't inherited by its derived classes. A method using the same signature in the derived class isn't an overridding method, but a new method.

Final classes

You can prevent a class from being extended by marking it as a `final` class. But why would you do so? A class marked as a final class can't be derived by any other class. For example, class `String`, which defines an immutable sequence of characters, is defined as a final class. It's a core class, which is used in a lot of Java API classes and user-defined classes. What happens, say, if a developer extends class `String` and modifies its `equals()` method to return a value `true` for all method parameter values passed to it? Because we can't extend the final class `String`, let's create a class `MyString` and override its `equals()` method to return `true` without comparing any values:

```
class MyString {
    String name;
    MyString (String name) {this.name = name;}
    public boolean equals(Object o) {
             return true;
    }
}
```

Many classes from the Java API, like `HashMap` and `ArrayList`, rely heavily on the correct implementation of `equals()` for searching, deleting, and retrieving objects. Imagine the effect it would have if you try to use objects of class `MyString` in an `ArrayList` and retrieve a matching value:

```
class UseMyStringInCollectionClasses {
    public static void main(String args[]) {
        ArrayList<MyString> list = new ArrayList<>();
        MyString myStrEast = new MyString("East");
        MyString myStrWest = new MyString("West");
        list.add(myStrEast);
        System.out.println(list.contains(myStrWest));
    }
}
```

Creates ArrayList—list

Add only one element to list—myStrEast

Prints "true"—list can find myStrWest, which was never added to it.

Surprisingly, the preceding code prints `true` even though `list` doesn't contain a matching `MyString` object. This is because method `contains90` in `ArrayList` uses the `equals()` (overridden) method of the objects it holds. If you can't get all of this explanation, don't worry. Collection classes are covered in detail in chapter 4. Imagine the havoc that objects of an extended `String` class can cause, if class `String` wasn't defined as a final class, was allowed to be extended, and its methods overridden.

You can mark a class as a final class by prefixing its definition with the keyword `final`:

```
final class FinalClass {
    //.. this need not be detailed here
}
```

Class is marked final by adding final to its definition.

You can't reverse the position of the keywords `final` and `class`:

```
class final ClassBeforeFinalWontCompile  {}
```

Won't compile; position of keywords class and final can't be interchanged.

The original intent of defining an abstract class was to extend it to create more meaningful and concrete classes. Because a final class can't be extended, you can't define a class both as final and abstract:

```
abstract final class FinalAbstractClassDontExist {}
```

Won't compile; a class can't be defined both as final and abstract.

If you try to extend a final class, your class won't compile:

```
final class Base {}
class Derived extends Base {}
```

Final class

Won't compile; can't extend a final base class.

EXAM TIP Look out for trick questions on the exam that extend final classes from the Java API, like class `String` and the wrapper classes `Byte`, `Short`, `Integer`, `Long`, `Float`, `Double`, `Boolean`, and `Character`. When you don't look at the source code of the base class and see that it's marked `final`, it's easy to overlook that the classes that extend it won't compile. Though the authors of this exam claim not to include trick questions, they also state that they expect the candidates to know "their stuff."

The enumerated types share some characteristics of the `final` keyword. Enumerated types enable you to define a new type, but with a predefined set of objects. Let's see how in the next section.

2.3 Enumerated types

[2.5] Use enumerated types

Think of the courses offered by a university or maybe even the roles within an organization: each defines a finite and predefined set of objects. These finite and predefined sets of objects can be defined as enumerated types, or *enums*. An enum defines a new custom data type (like interfaces and classes). Users are allowed to use only *existing* enum objects; they can't create new enum objects. *Type safety* was the main reason for introducing enumerated types in Java version 5.0, discussed further in the following section.

2.3.1 Understanding the need for and creating an enum

Let's assume that you have been assigned the task of creating a gaming application that can be played at exactly three levels: beginner, intermediate, and expert. How would you restrict your variable to be assigned only these three values?

You can accomplish this by creating an enum. An enum enables you to create a *type*, which has a *fixed* set of *constants*. Following is an example of the enum `Level`, which defines three programming levels:

```
enum Level { BEGINNER, INTERMEDIATE, EXPERT }
```

The enum values are constant values.

An enum lets you define a new type, the way classes and interfaces enable you to define your own types. The preceding line of code creates a new type, `Level`, which defines the constants `BEGINNER`, `INTERMEDIATE`, and `EXPERT` of type `Level`. (Syntactically, you can use any case for defining these constant values, but following Oracle's recommendation of using uppercase letters for constant values will save you a lot of headaches.) These constants are also static members and are accessible by using the name of the enum in which they're defined.

You can assign a gaming level, defined by the enum `Level` for a game. Let's define a class, `Game`, which defines an instance variable, `gameLevel`, of type `Level`, as follows:

```
class Game {
    Level gameLevel;
}
```

Variable of type Level

Class `GameApp` defines a field `game` of type `Game` and initializes it as follows:

```
class GameApp {
    Game game = null;

    public void startGame () {
        game = new Game();
        game.gameLevel = Level.BEGINNER;
    }
}
```

Assigns constant BEGINNER

The class `GameApp` demonstrates the real benefit of all this enum business. Because the variable `gameLevel` (defined in class `Game`) is of type `Level`, you can assign *only one* of the constants defined in the enum `Level`—that is, `Level.BEGINNER`, `Level.INTERMEDIATE`, or `Level.EXPERT`.

Let's look into the finer details of enums, as discussed in the next section.

2.3.2 Adding implicit code to an enum

When you create an enum, Java adds implicit code and modifiers to its members. These details will help you explain the behavior of enum constants, together with how to access and use them. Let's work with the enum `Level` created in the previous section (2.3.1) and decompile its `Level.class` file, using a decompiler (like JD):

```
enum Level { BEGINNER, INTERMEDIATE, EXPERT }
```

A decompiler converts Java byte code (.class) to a Java source file (.java). The newly created Java source file will include any implicit code that was added during the compilation process. Listing 2.2 shows decompiled enum `Level` (to make the code easier to understand, I've added some comments):

Listing 2.2 Decompiled enum Level

```
final class Level extends Enum
{
    public static final Level BEGINNER;
    public static final Level INTERMEDIATE;
    public static final Level EXPERT;

    private static final Level $VALUES[];

    static
    {
        BEGINNER = new Level("BEGINNER", 0);
        INTERMEDIATE = new Level("INTERMEDIATE", 1);
        EXPERT = new Level("EXPERT", 2);
        $VALUES = (new Level[] {
            BEGINNER, INTERMEDIATE, EXPERT
        });
    }

    public static Level[] values()
    {
        return (Level[])$VALUES.clone();
    }

    public static Level valueOf(String s)
    {
        return (Level)Enum.valueOf(Level, s);
    }

    private Level(String s, int i)
    {
        super(s, i);
    }
}
```

❶ **enum is implicitly declared final.**

❷ **enum constants are implicitly public, static, and final.**

❸ **Array to store reference to all enum constants**

❹ **Creation of enum constants occurs in static initializer block**

❺ **Method values return an array of all enum constants.**

❻ **Method valueOf() parses a String value and returns corresponding enum constant**

❼ **Private constructor**

In the decompiled code, at ❶ you can notice that an enum is implicitly defined as a `final` entity. At ❷ you can notice that all enum constants are implicitly declared as `public`, `final`, and `static` variables. The code at ❸ defines an array to store a reference to all enum constants. The variables are declared at ❷ and ❸. They are initialized in a static initializer block at ❹. Method `values()` returns an array of all enum constants at ❺ and `valueOf()` returns an enum constant for a corresponding `String` value at ❻. The code at ❼ defines a private constructor.

If the enum constants are themselves created in a static initializer block, when does a static initializer block in an enum execute? See for yourself in the next "Twist in the Tale" exercise.

Twist in the Tale 2.4

Let's add some code to the enum `Level` so it defines a constructor and a static initializer block. Examine the code and determine the correct options that follow.

```
enum Level {
    BEGINNER;
    static{ System.out.println("static init block"); }
    Level(){
        System.out.println("constructor");
    }
    public static void main(String... args){
        System.out.println(Level.BEGINNER);
    }
}
```

a
```
constructor
static init block
BEGINNER
```

b
```
static init block
constructor
BEGINNER
```

c
```
constructor
static init block
beginner
```

d
```
static init block
constructor
beginner
```

In listing 2.2 you'll notice that all enum constants of `BEGINNER`, `INTERMEDIATE`, and `EXPERT` are created in the order they were defined and assigned an ordinal: `0`, `1`, and `2`. Are enum constants created in this manner? Let's check it out in the next section.

2.3.3 *Extending java.lang.Enum*

All enums in Java extend the abstract class `java.lang.Enum`, defined in the Java API. As always, it's interesting to peek at the source code from the Java API, to understand why some pieces of code behave in a particular way. Let's look at the (partial) code of class `java.lang.Enum`, which will help you get the hang of how the enum constants are created, their order, and their default names. Please note that the comments aren‘t part of the code from class `java.lang.Enum`. These comments have been added to clarify the code for you.

Listing 2.3 Partial code listing of class `java.lang.Enum`

```
public abstract class Enum<E extends Enum<E>>
implements Comparable<E>, Serializable {
    private final String name;
    private final int ordinal;

    protected Enum(String name, int ordinal) {
        this.name = name;
        this.ordinal = ordinal;
    }

    public String toString() {
        return name;
    }

    public final String name() {
        return name;
    }
    //.. rest of the code
}
```

Name of the enum constant

Position of enum constant

Name and position of enum constant is saved on its creation

Default implementation of toString() returns name of enum constant

Method name() is marked final and can't be overridden; returns enum constant's name.

The class `Enum` defines only one constructor with `String` and `int` parameters to specify its name and ordinal (order). Every enum constant is implicitly assigned an order on its creation. Let's refer back to the example of enum `Level` as defined in listing 2.2. The enum constant values `BEGINNER`, `INTERMEDIATE`, and `EXPERT` are created *within* the enum `Level`, in its static initializer block (refer to code listing 2.1), as follows:

```
public static final level BEGINNER = new Level ("BEGINNER", 0);
public static final level INTERMEDIATE = new Level ("INTERMEDIATE", 1);
public static final level EXPERT = new Level ("EXPERT", 2);
```

EXAM TIP Watch out for exam questions that use methods like `Collections.sort()` from the Collections API to sort enum constants. The default order of enum constants is their order of definition. The enum constants aren't sorted alphabetically.

Now, examine the following code:

```
public class TestEnum {
    public static void main(String args[]) {
        System.out.println(Level.BEGINNER.name());
        System.out.println(Level.BEGINNER);
    }
}
```

Prints "BEGINNER"

Also prints "BEGINNER" by calling method toString()

Note both methods—`toString()` and `name()` defined in `java.lang.Enum`—return the value of the instance variable `name` (revisit code listing 2.3—class `java.lang.Enum` defines an instance variable name). Because method `name()` is a final method, you can't override it. But you can override method `toString()` to return any description that you want.

EXAM TIP For an enum constant `BEGINNER` in enum `Level`, calling `System.out.println(Level.BEGINNER)` returns the name of the enum constant—that is, `BEGINNER`. You can override `toString()` in an enum to modify this default return value.

Because a class can extend from only one base class, an attempt to make your enum extend any other class will fail. The following code won't compile:

```
class Person {}
enum Level extends Person { BEGINNER, INTERMEDIATE, EXPERT }
```

Won't compile

But you can make your enum implement any number of interfaces. A class can extend only one base class but can implement multiple interfaces. The following code compiles successfully:

```
interface MyInterface {}
enum Level implements MyInterface { BEGINNER, INTERMEDIATE, EXPERT }
```

Will compile

```
You can't explicitly make a class extend java.lang.Enum:
class MyClass extends java.lang.Enum {}
```

Won't compile

EXAM TIP An enum implicitly extends `java.lang.Enum`, so it can't extend any other class. But a class can't explicitly extend `java.lang.Enum`.

2.3.4 Adding variables, constructors, and methods to your enum

You can add variables, constructors, and methods to an enum. You can also override the nonfinal methods from the `java.lang.Enum` class. Following is an example:

```
enum IceCream {
    VANILLA, STRAWBERRY, WALNUT, CHOCOLATE;

    private String color;

    public String getColor() {
        return color;
    }

    public void setColor(String val) {
        color = val;
    }
```

❶ enum constants

❷ Instance variable in enum IceCream

❸ Method getColor()

❹ Method setColor()

```
    public String toString() {              ❺ Override method
        return "MyColor:"+ color;             toString()
    }
}
```

The code at ❶ defines a list of enum constants: `VANILLA`, `STRAWBERRY`, `WALNUT`, and `CHOCOLATE` in enum `IceCream`. Note that this constant list must be the first in the enum definition and should be followed by a semicolon. A semicolon is optional if you don't add methods and variables to your enum. The code at ❷ defines an instance variable `color` in enum `IceCream`. The code at ❸ and ❹ adds methods `getColor()` and `setColor()` to enum `IceCream`. The code at ❺ overrides the `public toString()` method inherited from class `java.lang.Enum`.

EXAM TIP The enum constant list must be the first in the enum definition and should be followed by a semicolon. A semicolon is optional if you don't add methods and variables to your enum.

You can call the methods defined in the preceding example, as follows:

```
public class UseIceCream {
    public static void main(String[] args) {
        IceCream.VANILLA.setColor("white");                  ❶
        System.out.println(IceCream.VANILLA.getColor());     ❷
        System.out.println(IceCream.VANILLA);                ❸
    }
}
```

The output of this code is as follows:

```
white
MyColor:white
```

Are you thinking that the code at ❶ is invalid because enum values are constant values? Note that this code is absolutely valid. `VANILLA` is a type of enum `IceCream`, and you can call methods that are available to it. The code at ❷ calls the method `getColor()`, and the code at ❸ calls method `toString()`, which was overridden in enum `IceCream`.

You can also define constructors in your enum and override methods that apply only to particular enum constants, as follows in listing 2.4 (modifications in bold):

Listing 2.4 enum `IceCream` with custom constructor and constant specific class body

```
enum IceCream {
    VANILLA("white"),
    STRAWBERRY("pink"),
    WALNUT("brown") {
        public String toString() {
            return "WALNUT is Brown in color";
        }
        public String flavor() {
            return "great!";
        }
    },
    CHOCOLATE("dark brown");

    private String color;
    IceCream(String color) {
        this.color = color;
    }
    public String toString() {
        return "MyColor:" + color;
    }
}
```

❶ **Methods defined between { and } are available only to enum constant WALNUT.**

❷ **This method can't be executed.**

❸ **Constructor that accepts string**

The code at ❶, known as a *constant specific class body*, defines overridding methods for a particular enum constant, WALNUT. The code at ❸ defines a constructor for enum `IceCream`, but it can be used only within an enum. A constructor in an enum can be defined only with *default* or `private` access; `public` and `protected` access levels aren't allowed.

EXAM TIP An enum can't define a constructor with `public` or `protected` access level.

In the preceding code, it might be strange to note that though you can define a method `flavor()` at ❷, you can't call it, as follows:

```
public class UseIceCream {
    public static void main(String[] args) {
        System.out.println(IceCream.VANILLA);
        System.out.println(IceCream.WALNUT);
        //System.out.println(IceCream.WALNUT.flavor());
    }
}
```

Prints "MyColor:white"

Prints "WALNUT is Brown in color"

Won't compile

This behavior can be attributed to `WALNUT` creating an anonymous class and overriding the methods of enum `IceCream`. But it's still referenced by a variable of type `IceCream`, which doesn't define the method `flavor`. If this leaves you guessing about what all this stuff with anonymous classes is, you can go grab a glass of anonymous inner classes in section 2.4.4.

Let's see how class `IceCream`'s constant `WALNUT` returns a custom value for its `toString()` method, that it overrides in its constant specific class body. In the following example, class `IceCreamParlor` uses method `values()` to access all enum constants and outputs their values:

```
class IceCreamParlor {
    public static void main(String args[]) {
        for (IceCream ic : IceCream.values())
            System.out.println(ic);
    }
}
```

Method values() returns an array of all enum constants.

The output of the preceding code is

```
MyColor:white
MyColor:pink
WALNUT is Brown in color
MyColor:dark brown
```

In the preceding output, notice how the `String` representation of `WALNUT` differs from the other enum constants.

EXAM TIP An enum constant can define a constant specific class body and use it to override existing methods or define new variables and methods.

2.3.5 Where can you define an enum?

You can define an enum as a top-level enum, or as a member of a class or an interface. Until now, you worked with a top-level enum. The following code shows you how to define an enum as a member of another class or interface:

```
class MyClass {
    enum Level { BEGINNER, INTERMEDIATE, EXPERT }
}
interface MyInterface {
    enum Level { BEGINNER, INTERMEDIATE, EXPERT }
}
```

enum as a member of other class

enum as a member of an interface

But you can't define an enum local as a method. For example, the following code won't compile:

```
class MyClass {
    void aMethod() {
        enum Level { BEGINNER, INTERMEDIATE, EXPERT }
    }
}
```

At the end of this section on enums, let's revisit the important rules that you should remember for the exam.

Rules to remember about enums

- An enum can define a main method. This means that you can define an enum as an executable Java application.
- The enum constant list must be defined as the first item in an enum, before the declaration or definition of methods and variables.

(continued)

- The enum constant list might not be followed by a semicolon, if the enum doesn’t define any methods or variables.
- When an enum constant overrides an enum method, the enum constant creates an anonymous class, which extends the enum.
- An enum constant can define a constant specific class body and use it to override existing methods or define new variables and methods.
- An enum implicitly extends `java.lang.Enum`, so it can’t extend any other class. But a class can’t explicitly extend `java.lang.Enum`. An enum can implement interface(s).
- An enum can never be instantiated using the keyword `new`.
- You can define multiple constructors in your enums.
- An enum can’t define a constructor with `public` or `protected` access level.
- An enum can define an abstract method. Just ensure to override it for all your enum constants.
- The enum method `values()` returns a list of all the enum constants.
- An enum can be defined as a top-level enum, or as a member or another class or interface. It can’t be defined local to a method.

When you’re consuming a lot of information, missing the small and simple details is easy. Let’s check whether you remember and can spot some basic information about enums in the next “Twist in the Tale” exercise.

Twist in the Tale 2.5

Let’s modify the code used in enum `IceCream` in this section. Examine the code and determine the correct options that follow.

```
public enum IceCreamTwist {
    VANILLA("white"),
    STRAWBERRY("pink"),
    WALNUT("brown"),
    CHOCOLATE("dark brown");

    String color;

    IceCreamTwist(String color) {
        this.color = color;
    }

    public static void main(String[] args) {
        System.out.println(VANILLA);            //line1
        System.out.println(CHOCOLATE);          //line2
    }
}
```

a Compilation error: Can’t run an enum as a standalone application.

b Compilation error at (#1) and (#2): Can’t access `VANILLA` and `CHOCOLATE` in a static main method.

c No errors. Output is

```
VANILLA
CHOCOLATE
```

d No errors. Output is

```
white
dark brown
```

An enum defines a new type with limitations. Similarly, nested and inner classes define a new type with constraints on their use. Defined within another class, nested and inner classes are characterized with a different set of behavior that sets them apart from the top-level classes. Let's uncover these details in the next section.

2.4 Static nested and inner classes

[2.4] Create top-level and nested classes

A *nested class* is a class defined within another class. Nested classes that are declared as static are referred to as *static nested classes*. Nested classes that aren't declared as static are referred to as *inner classes*. Like a regular top-level class, an inner or static nested class can define variables and methods.

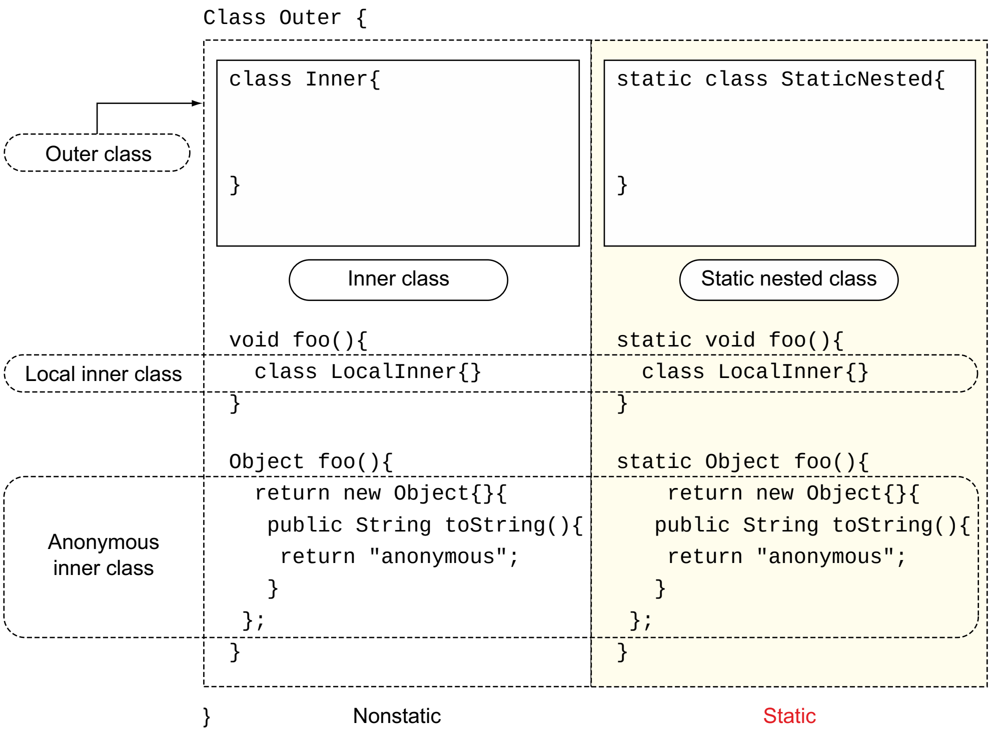
You can also define inner classes within methods and without a name. Figure 2.5 shows the types of inner classes, distinguished by their placement within the top-level class and whether they are defined as and with static members.

Before we dive into a detailed discussion of all the flavors of the inner classes, table 2.2 provides a quick definition of the types of inner classes.

Table 2.2 Flavors of inner classes and their definitions

Type of inner class	Description
Static or static nested class	Is a static member of its enclosing class and can access all the static variables and members of its outer class
Inner or member class	Is an instance member of its enclosing class. It can access all the instance and static members of its outer class, including private members.
Method local inner class	Is defined within a method. Local inner classes are local to a method. They can access all the members of a class, including its private members, but they can be accessed only within the method in which they're defined.
Anonymous inner class	Is a local class without a name

For the exam, you'll need to know why inner classes and static nested classes are important in the design of an application, their advantages and disadvantages, and how to create and use them. Let's start with a discussion of the advantages of inner



```
Class Outer {
  class Inner{

  }

  void foo(){
    class LocalInner{}
  }

  Object foo(){
    return new Object{}{
     public String toString(){
      return "anonymous";
     }
   };
  }
}
```

```
  static class StaticNested{

  }

  static void foo(){
    class LocalInner{}
  }

  static Object foo(){
      return new Object{}{
     public String toString(){
      return "anonymous";
     }
   };
  }
```

Figure 2.5 An outer class showing all types of inner classes that it can define: inner class, static nested class, local inner class, and anonymous inner class

classes, followed by a detailed discussion of all these classes. At the end of this section, we'll discuss their disadvantages.

2.4.1 Advantages of inner classes

Inner classes offer multiple advantages. To start, they help you objectify the functionality of a class, within it. For example, you might define a class `Tree`, which defines operations to add objects, remove objects, and sort them based on a condition. Instead of defining methods and variables to sort them within the class `Tree`, you could encapsulate sorting functionality within another class `TreeSort`. Because the class `TreeSort` would always work with `Tree` and might not be needed outside the class `Tree`, `TreeSort` can be defined as an inner class within class `Tree`. Another example for using inner classes is as parameter containers. Instead of using long method signatures, inner classes are often used to keep method signatures compact by passing reference parameters of inner classes instead of a long list of individual parameters.

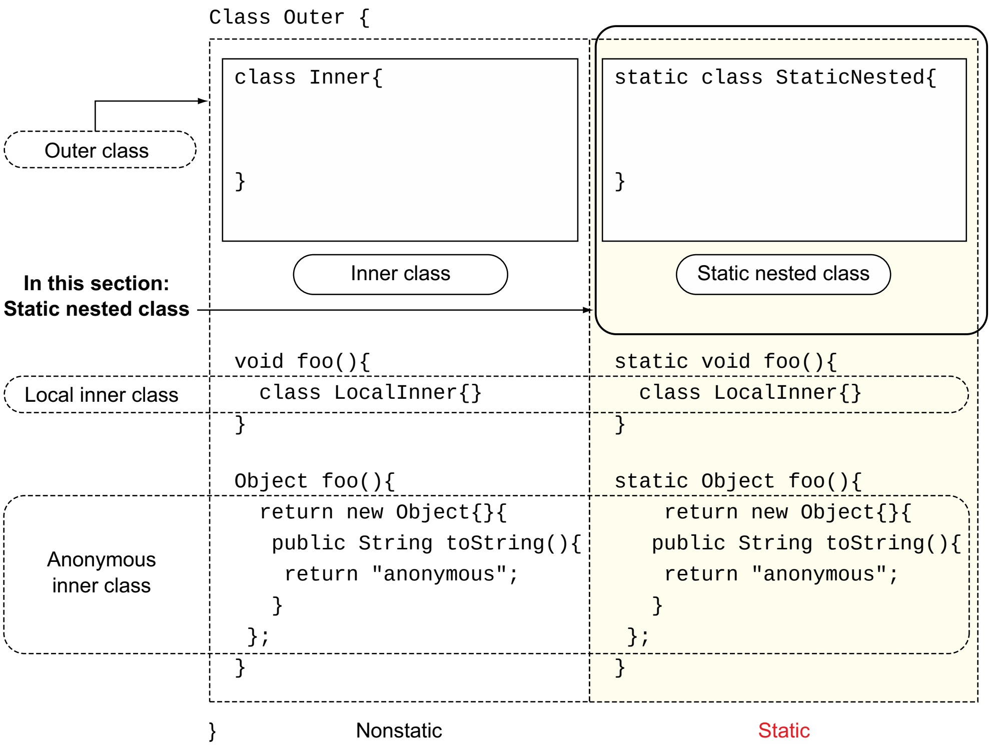
Just as you can organize your top-level classes by using packages, you can further organize your classes by using inner classes. Inner classes might not be accessible to all other classes and packages.

Inner classes also offer a neat way to define callback methods. For example, consider a user-interface-intensive GUI application, which defines multiple controls (buttons, keys, and screen) to accept a user's input. These user controls should register listeners, which are classes that define methods that are called back, when a user control receives an input. Instead of defining a single class to handle all callback methods for multiple user controls, you can use inner classes to define callback methods for individual user controls.

Let's start with the simplest type of inner class: a static nested class.

2.4.2 Static nested class (also called static inner class)

A *static nested class* is a static class that's defined (nested) within another class. It's referred to as a nested class and not an inner class because it isn't associated with any instance of its outer class. You'd usually create a static nested class to encapsulate partial functionality of your main class, whose instance can exist without the instance of its outer class. It can be accessed like any other static member of a class, by using the class name of the outer class. A static nested class is initialized when it's loaded with its outer class in memory. Figure 2.6 shows a static nested class.



```
Class Outer {
    class Inner{

    }

    void foo(){
      class LocalInner{}
    }

    Object foo(){
      return new Object(){
       public String toString(){
        return "anonymous";
       }
      };
    }

    static class StaticNested{

    }

    static void foo(){
      class LocalInner{}
    }

    static Object foo(){
        return new Object(){
       public String toString(){
        return "anonymous";
       }
      };
    }
}
```

Figure 2.6 A static nested class within an outer class

In the following (simplified) example, class DBConnection defines a static nested class, DBConnectionCache, which creates and stores database connections with default connection values. When requested a database connection, class DBConnection checks if a default connection for the specified database exists. If yes, it returns the default connection; otherwise it creates and returns a new connection.

```
class DBConnection {
    public DBConnection (String username, String pwd, String URL) {
        // code to establish a Database connection
    }
    public DBConnection OracleConnection
                  (String username, String pwd, String URL) {
        DBConnection conn = DBConnectionCache.getDefaultOracleConnection();
        if (conn != null) {
            return conn;
        }
        else {
            //establish and return new DBconnection using method parameters
        }
    }
    /*
     * Oversimplified version of a static nested class which uses default
     * values to establish DB connections and store them in a static array
     */
    static class DBConnectionCache {
        static DBConnection connections[];
        static {
            connections = new DBConnection[3];
            connections[0] = new DBConnection
                (/*arguments to establish a connection to an ORACLE DB*/);
            connections[1] = new DBConnection
                (/*arguments to establish a connection to a MySQL DB*/);
        }
        static DBConnection getDefaultOracleConnection() {
            return connections[0];
        }
        static DBConnection getDefaultMySQLConnection() {
            return connections[1];
        }
    }
}
```

EXAM TIP In the preceding example, access to nested class DBConnectionCache can be restricted by using an appropriate access modifier with its definition.

In the next section, you'll work with one of the most important points to be tested on the exam—how to instantiate static nested classes.

Let's code class StaticNested as shown in figure 2.6:

```
class Outer {
    static int outerStatic = 10;
    int outerInstance = 20;

    static class StaticNested {
        static int innerStatic = 10;
        int innerInstance = 20;
    }
}
```

NOTE A static nested class isn't usually referred to as an inner class, because it isn't associated with an object of the outer class.

When you create a static nested class, it's compiled as a separate class file. The .class file for a static nested file includes the name of its outer class. On compiling the code shown in the preceding example, the compiler generates two .class files, Outer.class and Outer$StaticNested.class.

As with a regular top-level class, a static nested class is a type and you can instantiate it. Multiple separate instances of a static nested class can be created. Each instance of the static nested class can have a different value for its instance variables. Let's instantiate the StaticNested class from the preceding example code:

```
class Outer {
    static int outerStatic = 10;
    int outerInstance = 20;

    static class StaticNested {
        static int innerStatic = 10;
        int innerInstance = 20;
    }
    public static void main(String args[]) {
        StaticNested nested1 = new StaticNested();
        Outer.StaticNested nested2 = new Outer.StaticNested();

        nested1.innerStatic = 99;
        nested1.innerInstance = 999;

        System.out.println(nested1.innerStatic + ":" +
                                       nested1.innerInstance);
        System.out.println(nested2.innerStatic + ":" +
                                       nested2.innerInstance);
    }
}
```

When static nested class is instantiated within its outer class, it doesn't need to be prefixed with its outer class name (though it can).

Modify only the value of innerInstance for nested1.

Modify the value of innerStatic for all instances of StaticNested.

Prints "99:999"

Prints "99:20"

When a static nested class is instantiated outside its outer class, you *must* prefix it with its outer class name:

```
class AnotherClass {
    Outer.StaticNested nested1 = new Outer.StaticNested();
    StaticNested nested2 = new StaticNested();
}
```

When static nested class is instantiated outside its outer class, it must be prefixed with its outer class name

Won't compile

```
StaticNested one = new StaticNested();                 ✓
Outer.StaticNested two = new Outer.StaticNested();     ✓

StaticNested three = new Outer.new StaticNested();     ✗
StaticNested four = new Outer().new StaticNested();    ✗
StaticNested five = Outer.new StaticNested();          ✗
```

Figure 2.7 Correct and incorrect instantiation of a static nested class

For the exam, it's important to remember the syntax of instantiating a static nested class: the count of operator `new` and its placement. It uses the `new` operator once, just before the name of the static nested class. Figure 2.7 highlights correct and incorrect instantiation code snippets.

Another point that you must remember when instantiating a static nested class is when to prefix the name of a static nested class with its outer class. Figure 2.8 shows

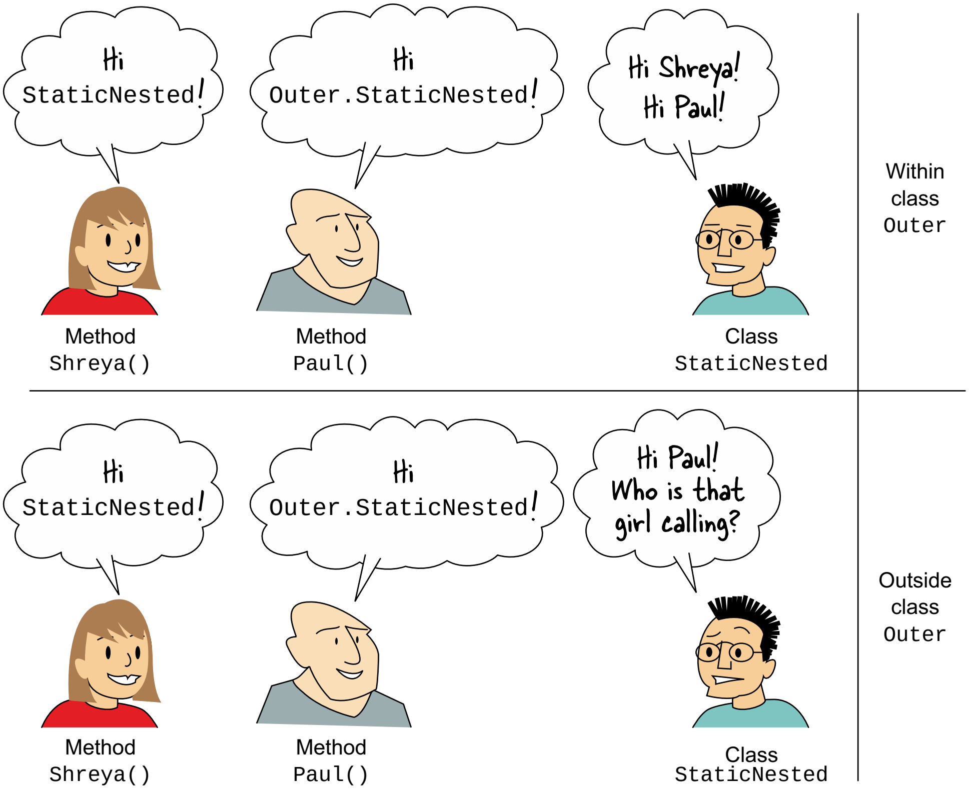


Figure 2.8 An interesting way to remember that you *must* prefix the name of the static inner class with its outer class when referring to it outside its outer class.

an interesting way to remember that you *must* prefix the name of the static inner class with its outer class when you're referring to it outside its outer class. For the rest of the cases you might, but it isn't mandatory, prefix the name of the static nested class with its outer class. In figure 2.8 when method `Shreya` doesn't prefix `StaticNested` with its outer class, outside the class `Outer`, `StaticNested` doesn't seem to recognize the call.

Accessing members of a static nested class

To access the static members of a static nested class, you need not create an object of this class. You need an object of a static nested class to access its instance members. Here's an example:

```
class Outer1 {
    public static void main(String args[]) {
        System.out.println(new Outer.StaticNested().innerInstance);
        System.out.println(Outer.StaticNested.innerStatic);
    }
}
```

Object of StaticNested class required to access its instance members

Object of StaticNested class not required to access its static members

NOTE On the exam, you might be asked whether you can instantiate a static nested class, how to instantiate it, and whether it can define instance or static members, or both.

Access levels of a static nested class

A static nested class can be defined using all access levels: `private`, *default* access, `protected`, and `public`. The accessibility of the static nested class depends on its access modifier. For example, a `private` static nested class can't be accessed outside its outer class. The access of a static nested class also depends on the accessibility of its outer class. If the outer class is defined with the default access, an inner nested class with `public` access won't make it accessible outside the package in which its outer class is defined.

Members of outer class accessible to static nested class

A static nested class can access only the static members of its outer class. An example follows.

```
class Outer {
    static int outerStatic = 10;
    int outerInstance = 20;

    static class StaticNested {
        static int innerStatic = outerInstance;
        int innerInstance = outerInstance;;
    }
}
```

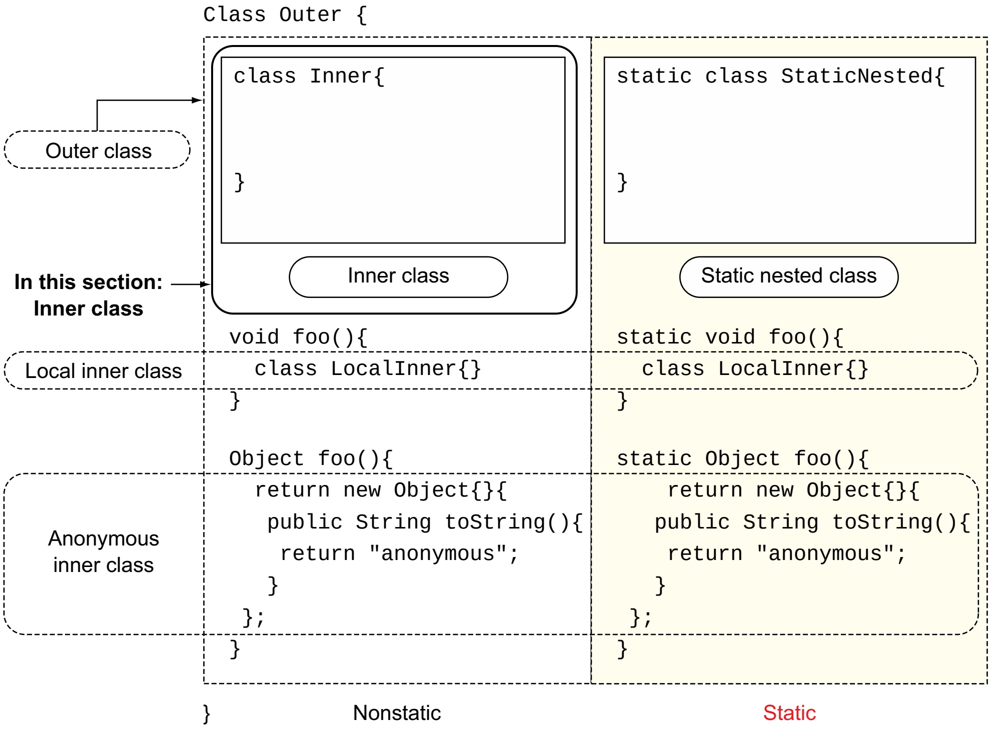
Can't access instance variables from a static nested class

Rules to remember about static nested classes

- To create an object of a static nested class, you need to prefix its name with the name of its outer class (necessary only if you're outside the outer class).
- A static nested class can define both static and nonstatic members.
- You need not create an object of a static nested class to access its static members. They can be accessed the way static members of a regular class are accessed.
- You should create an object of a static nested class to access its nonstatic members, by using the operator `new`.
- A static nested class can be defined using any access modifier.
- A static nested class can define constructor(s).

2.4.3 *Inner class (also called member class)*

The definition of an *inner class* is enclosed within another class, also referred to as an *outer class*. An inner class is an *instance member* of its outer class. An instance of an *inner class* shares a special bond with its *outer class* and can't exist without its instance. Figure 2.9 illustrates the placement of an inner class within an outer class.



```
Class Outer {
    class Inner{

    }

    void foo(){
      class LocalInner{}
    }

    Object foo(){
      return new Object(){
       public String toString(){
        return "anonymous";
       }
     };
    }
```

```
    static class StaticNested{

    }

    static void foo(){
      class LocalInner{}
    }

    static Object foo(){
        return new Object(){
       public String toString(){
        return "anonymous";
       }
     };
    }
}
```

Figure 2.9 Placement of an inner class within an outer class

You'd usually create an inner class to encapsulate partial functionality of your main class such that the existence of the inner class instance isn't possible without its outer class instance. This is in contrast to a nested static class, which can be used without an instance of its outer class.

For example, the following code defines a class `Tree` and an inner class `TreeSort`. `Tree` defines operations to add, remove, and sort objects based on a condition. Instead of defining methods and variables to sort the tree elements within class `Tree`, it encapsulates sorting functionality within class `TreeSort`. Class `TreeSort` would always work with `Tree` and might not be needed without class `Tree`:

```
class Node {
    Object value;
    Node left, right;
}
class Tree {
    Tree() {}
    Node rootNode;
    void addElement(Object value) {
        //.. code //
    }
    void removeElement(Object value) {
        //.. code //
    }
    void sortTree(boolean ascending) {
        new TreeSort(ascending).sort();
    }

    class TreeSort{
        boolean ascendingSortOrder = true;
        TreeSort(boolean order) {
            ascendingSortOrder = order;
        }
        void sort() {
            // outer class's rootNode and sort tree values
            // sorting code can be complex
        }
    }
}
```

Instances of Node used to store Tree elements

Defining sorting code in a separate inner class makes class Tree simpler and cleaner.

Figure 2.10 illustrates the bare-bones *inner class* `Inner`, defined within another class, `Outer`, and how the compiler generates separate class files for the outer and inner classes.

Throughout this section, I refer to the concept of outer and inner classes as *outer class* and *inner class*. I refer to names of the outer class and inner class by using `code font`.

NOTE You can create an *outer class* and *inner class* using any names. The names of these classes are chosen as `Outer` and `Inner` so it's easy for you to recognize them.

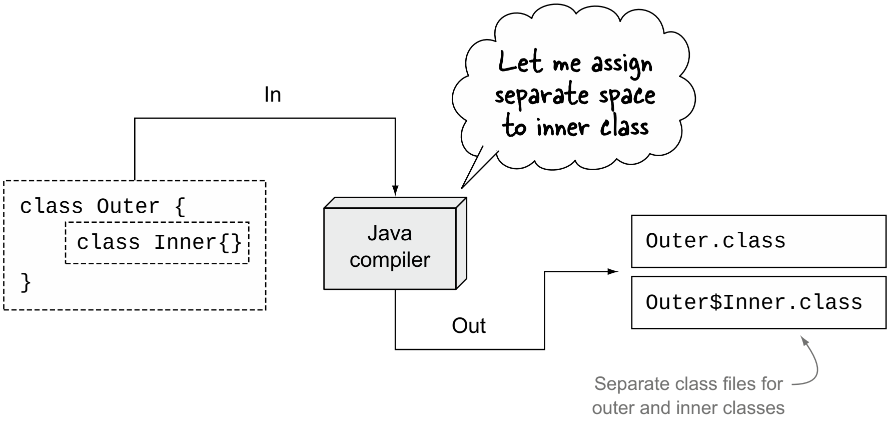

In
Let me assign separate space to inner class
class Outer {
class Inner{}
}
Java compiler
Out
Outer.class
Outer$Inner.class
Separate class files for outer and inner classes

Figure 2.10 The compiler generates separate class files for an outer class and inner class. The name of the inner class is prefixed with the name of the outer class and a $ sign.

Characteristics of inner classes

Because an *inner class* is a member of its outer class, an *inner class* can be defined using any of the four access levels: `public`, `protected`, *default access,* and `private`. Like a regular top-level class, an inner class can also define constructors, variables, and methods. But an inner class can't define nonfinal static variables or methods, as shown in figure 2.11.

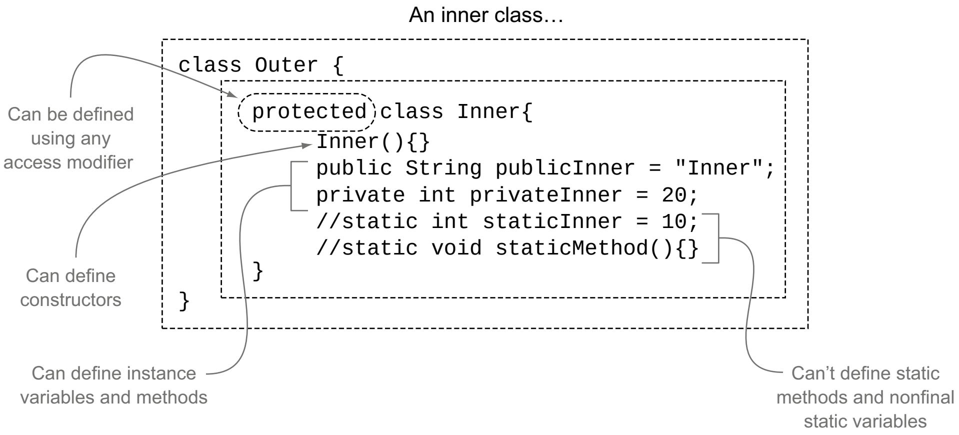

An inner class…
class Outer {
protected class Inner{
Inner(){}
public String publicInner = "Inner";
private int privateInner = 20;
//static int staticInner = 10;
//static void staticMethod(){}
}
}
Can be defined using any access modifier
Can define constructors
Can define instance variables and methods
Can't define static methods and nonfinal static variables

Figure 2.11 Characteristics of an *inner class*: it can be defined using any access modifier, can define constructors, and can define instance variables and methods. An inner class can define static members variables but not static methods.

Creation of an inner class

Whenever you instantiate an *inner class*, remember that an instance of an inner class can't exist without an instance of the *outer class* in which it's defined. Let's look at creating an inner class:

- Within an outer class, as an instance member
- Within a method of an outer class
- Within a static method of an outer class
- Outside the outer class

First, a definition of a bare-bones outer class and inner class follows:

```
class Outer {
    class Inner {}
}
```

Bare-bones outer class

Bare-bones inner class

Class `Outer` can instantiate inner class `Inner` as its instance member (additions in bold), as follows:

```
class Outer {
    Inner objectInner = new Inner();
    class Inner {}
}
```

Creation of object of class Inner in class Outer, as its instance member

In the previous code, like all instance variables, `objectInner` can access an instance of its outer class and its members, `Outer`. Similarly, an instance of an inner class created within an *instance* method of an outer class can access the instance of its outer class. So, you can instantiate class `Inner` within an instance method of class `Outer`, as follows (additions in bold):

```
class Outer {
    Inner in = new Inner();
    class Inner {}
    void aMethod () {
        Inner objectInner = new Inner();
    }
}
```

Instantiation of class Inner in class Outer within its method

EXAM TIP You *must* have an outer class instance to create an inner class instance.

Now, let's try to instantiate class `Inner` within a static method of class `Outer` (additions in bold):

```
class Outer {
    class Inner {}
    static void staticMethod() {
        Inner in = new Inner();
    }
}
```

❶ **Won't compile**

The code at ❶ doesn't compile because in method `staticMethod()` there's no outer class instance to tie the inner class instance to, which is required for creation of its *inner class* `Inner`. But it's possible to instantiate class `Outer` in the method `staticMethod()`. When you have an `Outer` instance, you can instantiate `Inner`:

```
class Outer {
    class Inner {}
    static void staticMethod () {
        Outer outObj = new Outer();
        Inner innerObj = outObj.new Inner();
    }
}
```

❶ **Instance of Outer can be created in method staticMethod()**

❷ **Instance of Inner is created by calling operator new on Outer instance**

The code at ❶ creates `outObj`, an `Outer` instance in the static method `staticMethod()`. `outObj` is used to create an instance of class `Inner` at ❷ because you need an outer class instance to create an inner class instance. It's interesting to note that the operator `new` is called on `outObj` to create the `innerObj` instance. Have you invoked the operator `new` on an object earlier? The operator `new` is always used to instantiate a class. But to instantiate an *inner class* by using an instance of an *outer class*, you should invoke the operator `new` on the *outer class*'s instance.

Following is another way of creating the instance `innerObj`, using a single line of code:

```
class Outer {
    class Inner {}
    static void staticMethod () {
        Inner innerObj = new Outer().new Inner();
    }
}
```

❶ **Single line of code creates inner class object in outer class's static method**

The code at ❶ may look bizarre, because it contains two occurrences of the operator `new`. The first occurrence of this operator is used to create an instance of `Outer`. The second occurrence is used to create an instance of class `Inner`.

If another class wants to create an instance of class `Inner`, it needs an instance of class `Outer`. If it doesn't have one, it should first create one, just like with method `staticMethod()` in the previous code snippet:

```
class Foo {
    Inner inner;
    Foo () {
        Outer outer = new Outer();
        inner = outer.new Inner();
    }
}
```

Won't compile

What makes the preceding code fail compilation? `Inner` isn't a top-level class, so its variable type should include the name of its outer class, `Outer`, so class `Foo` can find this class. The following code compiles successfully and creates an `Inner` instance:

```
class Foo {
    Outer.Inner inner;
    Foo () {
        Outer outer = new Outer();
        inner = outer.new Inner();
    }
}
```

Outside its outer class, the type of inner class should include the name of its outer class.

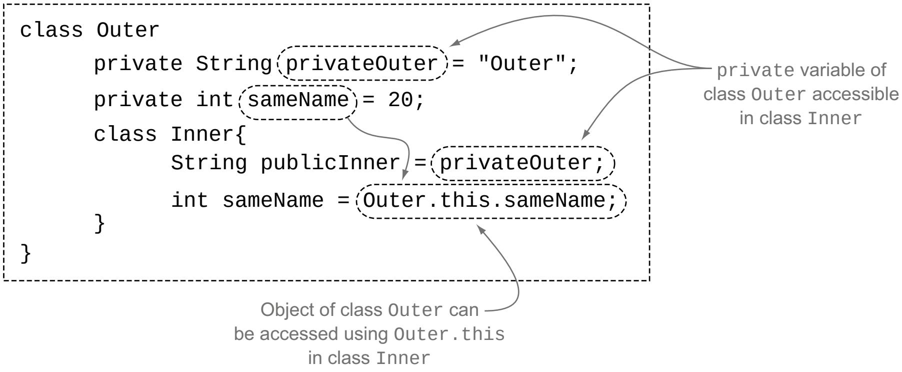
Similarly, a static method of class `Foo` can instantiate `Inner`, as follows:

```
class Foo {
    public static void main(String args[]) {
        Outer outer = new Outer();
        Outer.Inner inner = outer.new Inner();
    }
}
```

EXAM TIP The accessibility of an *inner class* outside its *outer class* depends on the access modifier used to define the inner class. For example, an inner class with default access can't be accessed by classes in different packages than the outer class.

WHAT CAN AN INNER CLASS ACCESS?

An inner class is a part of its outer class. Therefore an inner class can access all variables and methods of an outer class, including its private members and the ones that it inherits from its base classes. An inner class can also define members with the same name as its outer class, as shown in figure 2.12.



```
class Outer
      private String privateOuter = "Outer";
      private int sameName = 20;
      class Inner{
            String publicInner = privateOuter;
            int sameName = Outer.this.sameName;
      }
}
```

private variable of class Outer accessible in class Inner
Object of class Outer can be accessed using Outer.this in class Inner

Figure 2.12 An inner class can access all the members of its outer class, including its private members. Outer class members with the same name as inner class members can be accessed using `Outer.this`, where `Outer` is the name of the outer class.

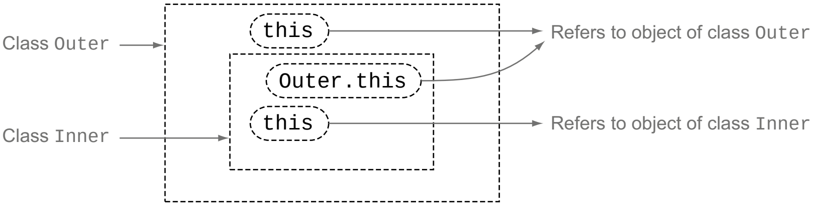
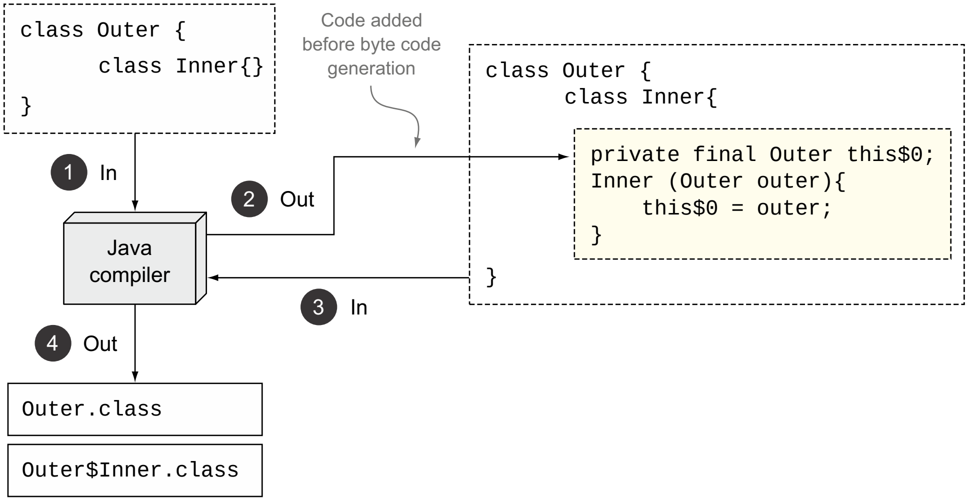

Class Outer
Class Inner
this
Outer.this
this
Refers to object of class Outer
Refers to object of class Inner

Figure 2.13 An inner class uses `this` to refer to its own object and `<name_of_its_outer_class>.this` to refer to its outer class's object.

An object uses the reference `this` to refer to its own object. An inner class can use the reference `this` to refer to its own object, and the name of its outer class followed by `.this` to refer to the object of its outer class, as shown in figure 2.13.

CAN AN INNER CLASS COEXIST WITH ONLY ITS OUTER CLASS?

Yes, an inner class can exist only with an object of its outer class. When a compiler compiles an inner class, it seems to insert code in the inner class, which defines an instance variable of its outer class, initialized using its constructor, as illustrated in figure 2.14.



```
class Outer {
      class Inner{}
}
```

```
class Outer {
      class Inner{
          private final Outer this$0;
          Inner (Outer outer){
              this$0 = outer;
          }
}
```

Figure 2.14 Java instantiates an inner class by passing it an outer class instance.

Rules to remember about inner classes

- You can create an object of the inner class within an outer class or outside an outer class.
- When an inner class is created outside its outer class, its type name should include the name of its outer class, followed by a dot (`.`) and then the name of the inner class.
- To create an inner class with a static method of an outer class, or outside an outer class, call the operator new on the object of the outer class to instantiate the inner class.
- An inner class can't define static methods. It can define final static variables but nonfinal static variables aren't allowed.
- Members of the inner class can refer to all variables and methods of the outer class.
- An inner class can be defined with all access modifiers.
- An inner class can define constructors.
- An inner class can define variables and methods with any access level.

It's time to attempt a quick exercise on inner classes in the following "Twist in the Tale" exercise.

Twist in the Tale 2.6

Apart from modifying the code used in an earlier example, I have also modified the class names for this exercise. Your task is to examine the following code and determine the correct answer option.

```
class Flower {
    String color = "red";
    Petal[] petals;
    private class Petal {
        public Petal() {System.out.println(color);}
        String color = "purple";             // line 1
        static final int count = 3;          // line 2
    }
    Flower() {
        petals = new Petal[2];               // line 3
    }
    public static void main(String args[]) {
        new Flower();
    }
}
```

- a Code prints `red` twice.
- b Code prints `purple` twice.
- c Code prints `red` three times.
- d Code prints `purple` three times.
- e Code prints nothing.
- f Code fails compilation due to code at (#1).
- g Code fails compilation due to code at (#2).
- h Code fails compilation due to code at (#3).

Anonymous classes are another type of inner class. As their title suggests, they don't have a name. In the next section, you'll see why we need unnamed inner classes, and when and how they are created.

2.4.4 Anonymous inner classes

As the name implies, an *anonymous inner class* isn't defined using an *explicit* name. An anonymous inner class is created when you combine instance creation with inheriting a class or implementing an interface. Anonymous classes come in handy when you wish to override methods for only a particular instance. They save you from defining new classes. The anonymous class might override none, few, or all methods of the inherited class. It must implement all methods of an implemented interface. The newly created object can be assigned to any type of variable—static variable, instance variable, local variable, method parameter, or returned from a method. Let's start with an example of an anonymous inner class that extends a class.

Anonymous inner class that extends a class

Let's start with a class `Pen`:

```
class Pen{
    public void write() {
        System.out.println("Pen-write");
    }
}
```

This is how a class, say, `Lecture`, would *usually* instantiate `Pen`:

```
class Lecture {
    Pen pen = new Pen();
}
```

Let's replace this *usual* object instantiation by overriding method `write`, *while* instantiating `Pen`. To override method `write()` for this particular instance, we insert a class definition between `()` and `;` (`()` and `;` are in bold):

```
class Lecture {
    Pen pen = new Pen()
    {
        public void write() {
            System.out.println("Writing with a pen");
        }
    }
    ;
}
```

❶ Create an anonymous class that extends class Pen.

❷ Begin class definition for anonymous class.

❸ Override method write() for instance referred to by pen.

❹ End class definition for anonymous class.

❺ End of creation of object referred to by variable pen, marked by semicolon.

The preceding code creates an anonymous class, which extends class `Pen`. The object reference `pen` refers to an object of this anonymous class.

The code at ❶ creates an object. Note that this line of code isn't followed by a semicolon. Instead, it's followed by an opening brace at ❷, which starts the definition of the anonymous class that extends `Pen`. The code at ❸ overrides method `write()` from the *base class* `Pen`. The closing brace at ❹ marks the end of the definition of the anonymous class. The code at ❺ defines the semicolon, which is used to mark the end of object creation, which started at ❶. I've deliberately placed the semicolon on a separate line so you can clearly identify the start and end of the anonymous class. It's usual to place this semicolon after the closing brace, used to mark the end of the anonymous class.

You can create an anonymous class even if you don't override any methods of class `Pen`:

```
class Lecture {
    static Pen pen = new Pen(){};

    public static void main(String args[]) {
        System.out.println(new Pen());
        System.out.println(pen);
    }
}
```

Create an anonymous class, which extends Pen but doesn't override its methods.

Prints a value similar to Pen@1034bb5; new Pen() returns an object of Pen.

Prints a value similar to Lecture$1@15f5897; pen refers to an instance of anonymous class that extends Pen.

Similarly, you can pass an anonymous class instance to a method parameter. Now let's see an example of method `notes()` in class `Lecture`, which accepts a method parameter of type `Pen`:

```
class Pen{
    public void write() {
        System.out.println("Pen-write");
    }
}
```

```
class Lecture {
    public void notes(Pen pen) {
        pen.write();
    }
}
```

Method notes() accepts parameter of type Pen.

Here's how another class—say, `Student`–calls `notes()` from class `Lecture`, subclassing `Pen`, and passing the object to it:

```
class Student {
    void attendLecture() {
        Lecture lecture = new Lecture();
        lecture.notes(new Pen() {
            public void write() {
                System.out.println("Okay! I am writing");
            }
        }
        );
    }
}
```

Call notes() by passing it newly baked object of anonymous subclass of Pen.

The preceding code can seem to be more complex than the previous example. To understand it better, let's build this code, line by line. In class `Student`, you call `notes()` on object reference `lecture`:

```
class Student {
    void attendLecture() {
        Lecture lecture = new Lecture();
        lecture.notes(/* need to pass Pen object */);
    }
}
```

In the next step, I'll replace the comment in the preceding code with `new Pen(){}`, so I'm ready to subclass `Pen` (additions in bold):

```
class Student {
    void attendLecture() {
        Lecture lecture = new Lecture();
        lecture.notes(new Pen(){});
    }
}
```

I'll insert the definition of overridden method `write()` within the curly brace `{}` of the `Pen` anonymous inner class declaration, included in the method parameter to `notes()` (additions in bold):

```
class Student {
    void attendLecture() {
        Lecture lecture = new Lecture();
        lecture.notes(new Pen(){public void write() {
                System.out.println("Okay! I am writing");
            }});
    }
}
```

Let's indent the code, to improve the readability:

```
class Student {
    void attendLecture() {
        Lecture lecture = new Lecture();
        lecture.notes(new Pen(){
            public void write() {
                System.out.println("Okay! I am writing");
            }
        });
    }
}
```

You can use an anonymous inner class to return a value from a method:

```
class Outer{
    Object foo() {
        return new Object() {
            public String toString() {
                return "anonymous";
            }
        };
    }
}
```

Create an anonymous class that subclasses class Object.

Override method toString() from class Object.

ANONYMOUS INNER CLASS THAT IMPLEMENTS AN INTERFACE

Until now, you should have read that you can't instantiate interfaces; you can't use the keyword `new` with an interface. Think again. Examine the following code, in which the class `BirdSanctuary` instantiates the interface `Flyable` by using the keyword `new`:

```
interface Flyable{
    void fly();
}

class BirdSanctuary {
    Flyable bird = new Flyable(){
        public void fly() {
            System.out.println("Flying high in the sky");
        }
    };
}
```

Use new to instantiate interface.

Don't worry; you've been reading correctly that you can't use the operator `new` with an interface. The catch in the preceding code is that `bird` refers to an object of an anonymous inner class, which implements the interface `Flyable`.

The anonymous class used to instantiate an interface in the preceding code saved you from creating a class beforehand, which implemented the interface `Flyable`.

EXAM TIP An anonymous inner class can extend at most one class or implement one interface. Unlike other classes, an anonymous class can neither implement multiple interfaces, nor extend a class and implement an interface together.

How to access additional members in anonymous classes

By using an anonymous class, you can override the methods from its base class or implement the methods of an interface. You can also define new methods and variables in an anonymous class (in bold):

```
interface Flyable{
    void fly();
}

class BirdSanctuary {
    Flyable bird = new Flyable(){
        public void fly() {
            System.out.println("Flying high in the sky");
        }
        public void hungry(){
            System.out.println("eat");
        }
    };

}
```

You can't call the additional member, method `hungry()`, using the reference variable `bird`. Why? The type of the reference variable `bird` is `Flyable`. So the variable `bird` can access only the members defined in interface `Flyable`. The variable `bird` can't access additional methods and variables that are defined in anonymous classes that implement it.

Anonymous class defined within a method

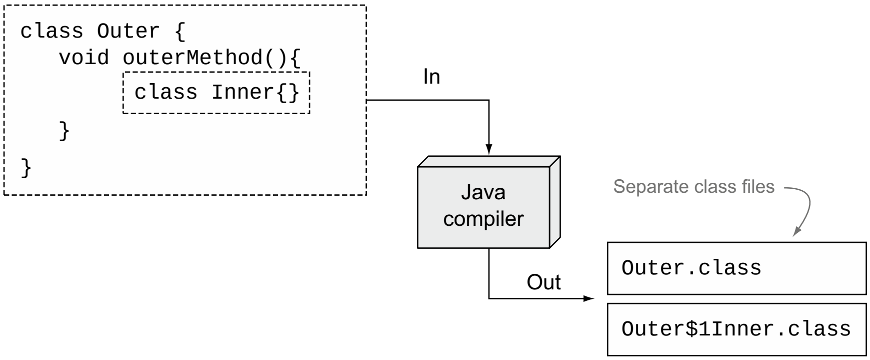
When an anonymous inner class is defined within a method, it can access only the final variables of the method in which it's defined. This is to prevent reassignment of the variable values by the inner class. Examine the following example.

```
class Pizza{
    Object margarita() {
        String ingredient = "Cheese";
        return new Pizza() {
            public String toString() {
                System.out.println(ingredient);
                return "margarita";
            }
        };
    }
}
```

❶ **Won't compile**

The code at ❶ will compile if `ingredient` is modified to be defined as a `final` local variable.

The last type of inner class on this exam is a method local inner class, which can be created within methods and code blocks like initializer blocks, conditional constructs, or loops. I'll discuss these in the next section.



```
class Outer {
   void outerMethod(){
         class Inner{}
   }
}
```

Figure 2.15 Compiler generates separate class files for an outer class and method local inner class. The name of the method local inner class is prefixed with the name `Outer` class, a $ sign, and an integer.

2.4.5 Method local inner classes

The *method local inner classes* are defined within static or instance methods of a class. Though these classes can also be defined within code blocks, they are typically created within methods. So their discussion will be limited to their creation in methods. Figure 2.15 shows the definition of a bare-bones method local inner class, `Inner`, defined within method `outerMethod()` of class `Outer`. The Java compiler generates separate class files for the classes `Outer` and `Inner`. Because a class can define method local inner classes with the same name in separate methods, the Java compiler adds a number to the name of the compiled file for the local inner classes.

A class can define multiple method local inner classes, with the same class name, in separate methods, as follows:

```
class Outer {
    void outerMethod () {
        class Inner { }
    }
    static void outerMethod2 () {
        class Inner { }
    }
}
```

Class Inner defined in method outerMethod()

Class Inner defined in method outerMethod2()

For the preceding code, the Java compiler will generate three class files: `Outer.class`, `Outer$1Inner.class`, and `Outer$2Inner.class`.

CHARACTERISTICS OF METHOD LOCAL INNER CLASSES

Recall that none of the variables within a method can be defined using any *explicit* modifier (`public`, `protected`, `private`). Similarly, method local inner classes can't be defined using any explicit access modifier. But a method local inner class can define its own constructors, variables, and methods by using any of the four access levels:

```
class Outer {
    private int privateOuter = 10;
    void outerMethod () {
        class Inner {
            protected Inner() {}
            public int publicInner = 100;
            int privateInner = privateOuter;
        }
    }
}
```

Can't be defined using an explicit access modifier

Can define its constructs, variables, and methods using any access level

Can access all members, including private members, of its outer class

But a method local inner class can't define static variables or static methods.

Creation of a local inner class

A *method local inner class* can be created only within the method in which it's defined. Also, its object creation can't appear before the declaration of the *local inner class*, as shown in the following code:

```
class Outer {
    void outerMethod () {
        //Inner in1 = new Inner();
        class Inner {}
        Inner in2 = new Inner();
    }
}
```

Won't compile

Will compile

What can a method local inner class access?

A method local inner class can access all variables and methods of its *outer class*, including its private members and the ones that it inherits from its base classes. A method local inner class can also define members with the same name as its outer class. In this case, the members of the outer class can be referred to by using the name of the outer class followed by the implicit reference `this`. Class `Inner` can access members of class `Outer` by using `Outer.this`, as shown in the following code:

```
class Outer {
    private int privateOuter = 10;
    void outerMethod () {
        class Inner {
            protected Inner() {}
            public int publicInner = 100;
            int privateInner = Outer.this.privateOuter;
        }
    }
}
```

Local inner class can access all members of its outer class, including private members.

2.4.6 Disadvantages of inner classes

Using inner classes is an advanced concept, and it can be difficult for inexperienced programmers to identify, implement, and maintain them.

On translation to byte code, inner classes can be accessed by classes in the same package. Because inner classes can access the private members of their outer class, they could break the designed encapsulation. You need to be careful when you create the inner and static nested classes.

2.5 Summary

This chapter covers abstract classes, `static` and `final` keywords, enumerated types, and nested and inner classes. The choice of identifying abstract classes isn't straightforward. This chapter showed you simple examples so you're well aware of their need, importance, advantages, and shortcomings.

Application of the `static` and `final` nonaccess modifiers are important design decisions. You should know about the Java entities that can use these modifiers, together with how they change the default behavior of the entities. Incorrect design decisions can make an application inefficient and difficult to extend or maintain.

This chapter covered enums that are used to create a new type with a finite and predefined set of objects. The definition of an enum can be as simple as including only the name of enum constants, or as complex as including variables, constructors, and methods. The exam is sure to test you on the finer details of enums, all covered in this chapter.

Static nested classes, inner classes, anonymous inner classes, and method local inner classes were also covered. An inner class shares an intimate relation with its outer class. Inner classes help you objectify the functionality of a class. Identification of an inner class is also an important design decision. It can help you further organize your code and allow limited access to your inner classes. But an overdose of the inner and nested classes can make your application difficult to work with and manage.

REVIEW NOTES

This section lists the main points covered in this chapter.

Abstract classes

- An *abstract class* is defined by using the keyword `abstract`. It defines variables to store the state of an object. It may define abstract and nonabstract methods.
- An abstract class must not necessarily define abstract methods. But if it defines even one abstract method, it must be marked as an abstract class.
- An abstract class can't be instantiated.
- An abstract method doesn't have any implementation. It represents a behavior that's required by all derived classes of an abstract class. Because the base class doesn't have enough details to implement an abstract method, the derived classes are left to implement it in their own specific manner.
- An abstract class can't be instantiated.
- An abstract class *forces* all its nonabstract-derived classes to implement the incomplete functionality in their own unique manner.

- A base class should be defined as an abstract class so it can implement the available details but still prevent itself from being instantiated.
- An abstract class can be extended by both abstract and concrete classes. If an abstract class is extended by another abstract class, the derived abstract class *might* not implement the abstract methods of its base class.
- If an abstract class is extended by a concrete class, the derived class *must* implement all the abstract methods of its base class, or it won't compile.
- A derived class must call its superclass's constructor (implicitly or explicitly), irrespective of whether the superclass or derived class is an abstract class or concrete class.
- An abstract class can't define abstract static methods. Because static methods belong to a class and not to an object, they aren't inherited. A method that can't be inherited can't be implemented. Hence this combination is invalid.
- Efficient use of an abstract class lies in the identification of an abstract class in your application design so you can define common code for your objects and leave the ones that are more specific, by defining them as abstract. You can enforce the derived classes to implement these abstract methods.

Nonaccess modifier—static

- Static members (fields and methods) are common to all instances of a class, and aren't unique to any instance of a class.
- Static members exist independently of any instances of a class, and may be accessed even when no instances of the class have been created.
- Static members are also known as *class fields* or *class methods* because they are said to belong to their class, and not to any instance of that class.
- A static variable and method can be accessed using the name of an object reference variable or the name of a class.
- A static method and variable can't access nonstatic variables and methods of a class. But the reverse works: nonstatic variables and methods can access static variables and methods.
- Static classes and interfaces are a type of nested classes and interfaces.
- You can't prefix the definition of a top-level class or an interface with the keyword `static`. A top-level class or interface is one that isn't defined within another class or interface.

Nonaccess modifier—final

- You can't reinitialize a final variable defined in any scope—class, instance, local, or method parameter.
- An instance final variable can be initialized either with its declaration in the initializer block or in the class's constructor.

- A static final variable can be initialized either with its declaration or in the class's static initializer block.
- You can't initialize a final instance variable in an instance method because it can't be guaranteed to execute only once. Such a method won't compile.
- You can't initialize a final static variable in a static method because it can't be guaranteed to execute only once. Such a method won't compile.
- If you don't initialize a final local variable in a method, the compiler won't complain, as long as you don't use it.
- If you try to access the value of a final local variable before assigning a value to it, the code won't compile.
- The Java compiler considers initialization of a final variable complete *only* if the initialization code will execute in *all* conditions. If the Java compiler can't be sure of execution of code that assigns a value to your final variable, it will complain (code won't compile) that you haven't initialized a final variable. If an `if` construct uses constant values, the Java compiler can predetermine whether the `then` or `else` blocks will execute. In this case, it can predetermine whether these blocks of code will execute to initialize a final variable.
- A final instance variable defined in a base class can't be initialized in the derived class. If you try to do so, your code won't compile.
- Final methods defined in a base class can't be overridden by its derived classes.
- Final methods are used to prevent a derived class from overriding the implementation of a base class's method.
- Private final methods in a base class aren't inherited by derived classes. A method defined using the same method signature in a derived class isn't an overridden method, but a new method.
- A final class can't be extended by any other class.
- A class is defined as final so that it can't be extended by any other class. This prevents objects of derived classes from being passed on to reference variables of their base classes.
- An interface can't be defined as final because an interface is abstract, by default. A Java entity can't be defined both as final and abstract.

Enumerated types

- Enumerated types are also called *enums.*
- An enum enables you to create a *type*, which has a *fixed* set of *constants.*
- An enum can never be instantiated using the keyword `new`.
- Unlike a class, which is defined using the keyword `class`, an enumerated type is defined using the keyword `enum`, and can define multiple variables and methods.
- If you define a variable of an enum type, it can be assigned constant values only from that enum.
- All enums extend the abstract class `java.lang.Enum`, defined in the Java API.

- Because a class can extend from only one base class, an attempt to make your enum extend any other class will fail its compilation.
- The enum constants are implicit static members.
- An enum can implement any interface, but its constants should implement the relevant interface methods.
- An enum can define an abstract method. Just ensure that you override it for all your enum constants.
- You can add instance variables, class variables, instance methods, and class methods to your enums.
- An enum can't use instance variables in the overridden methods for a particular enum constant.
- You can override nonfinal methods from class `java.lang.Enum`, for individual (or all) enum constants.
- Your enums can also define constructors, which can be called from within the enum.
- You can define multiple constructors in your enums.
- Enum constants can define new methods, but these methods can't be called on the enum constant.
- You can define an enum as a top-level enum or within another class or interface.
- You can't define an enum local to a method.
- An enum can define a main method.

Static nested classes

- This class isn't associated with any object of its outer class. Nested within its outer class, it's accessed like any other static member of a class—by using the class name of the outer class.
- A static nested class is accessible outside the class in which it's defined by using names of both the outer class and inner class.
- You can define both static and nonstatic members in a static nested class.
- A static nested class can define constructors.
- To access the static members of a static nested class, you need not create an object of this class. You need an object to access the instance members of this class.
- The accessibility of the nested static class depends on its access modifier. For example, a private static nested class can't be accessed outside its class.
- A static nested class can access only the static members of its outer class. Similarly, the outer class can access only the static members of its nested inner class. An attempt to access instance members on either side will fail compilation unless it's accessed through an instance of the outer or static nested class.
- All access levels can be used with this class—`public`, `protected`, *default*, and `private`.

Inner classes

- An inner class is an *instance member* of its outer class.
- An object of an *inner class* shares a special bond with its *outer class* and can't exist without its instance.
- An inner class can be defined using any of the four access levels—`public`, `protected`, *default*, and `private`.
- Members of an inner class can refer to all variables and methods of an outer class.
- An inner class can define constructors.
- An inner class can define variables and methods with any access.
- An inner class can't define static methods and nonfinal static variables.
- You can create an object of an inner class within an outer class or outside an outer class.
- Outside the outer class an inner class is instantiated using

```
Outer.Inner inner = new Outer().new Inner();
```

Anonymous inner classes

- An anonymous inner class is created when you combine object instance creation with inheriting a class or implementing an interface.
- An anonymous inner class might override none, few, or all methods of the inherited class.
- An anonymous inner class must implement all methods of the implemented interface.
- An instance of an anonymous class can be assigned to any type of variable (static variable, instance variable, or local variable) or method parameter, or be returned from a method.
- The following line creates an anonymous inner class that extends `Object` and assigns it to a reference variable of type `Object`:

```
Object obj = new Object(){};
```

- The following line calls a method, say `aMethod()`, passing to it an instance of an anonymous class that implements `Runnable`:

```
aMethod(new Runnable() {
    public void run() {}
});
```

- When an anonymous inner class is defined within a method, it can access only the final variables of the method in which it's defined. This is to prevent reassignment of the variable values by the inner class.
- Though you can define variables and methods in an anonymous inner class, they can't be accessed using the reference variable of the base class or interface, which is used to refer to the anonymous class instance.

Method local inner classes

- Method local inner classes are defined within a static or instance method of a class.
- A class can define multiple method local inner classes, with the same class name, but in separate methods.
- Method local inner classes can't be defined using any explicit access modifier.
- A method local inner class can define its own constructors, variables, and methods by using any of the four access levels—`public`, `protected`, *default*, and `private`.
- A method local inner class can be created only within the method in which it's defined. Also, its object creation can't appear before its declaration.
- A method local inner class can access all variables and methods of its *outer class*, including its private members and the ones that it inherits from its base classes. It can only access the final local variables of the method in which it's defined.
- A method local inner class can define members with the same name as its outer class. In this case, the members of the outer class can be referred to by using `Outer.this`.

SAMPLE EXAM QUESTIONS

Q 2-1. Select the correct statement(s) based on the following code:

```
enum Keywords {
    ASSERT(1.4),                             // line1
    DO, IF, WHILE;                           // line2
    double version = 1.0;                    // line3

    Keywords() {                             // constructor 1
        this.version = 1.0;                  // constructor 1
    }                                        // constructor 1

    Keywords(double version) {               // constructor 2
        this.version = version;              // constructor 2
    }                                        // constructor 2

    public static void main(String args[]) {
        Keywords[] keywords = Keywords.values();
        for (Keywords val:keywords) System.out.println(val);
    }
}
```

- **a** The enum keywords won't compile due to code at (#1).
- **b** The enum keywords won't compile due to code at either (#2) or (#3).
- **c** If you swap the complete code at (#1) and (#2) with code at (#3), enum keywords will compile successfully.
- **d** The enum keywords will fail to compile due to the declaration of multiple constructors.
- **e** None of the above

Q 2-2. Consider the following definition of class Foo:

```
abstract class Foo {
    abstract void run();
}
```

Which of the classes correctly subclass Foo? (Choose all that apply.)

a
```
class Me extends Foo {
    void run() {/* ... */}
}
```
b
```
abstract class You extends Foo {
    void run() {/* ... */}
}
```
c
```
interface Run {
    void run();
}
class Her extends Foo implements Run {
    void run() {/* ... */}
}
```
d
```
abstract class His extends Foo {
    String run() {/* ... */}
}
```

Q 2-3. Which lines of code, when inserted at //INSERT CODE HERE, will print the following:

```
BASKETBALL:CRICKET:TENNIS:SWIMMING:

enum Sports {
    TENNIS, CRICKET, BASKETBALL, SWIMMING;
    public static void main(String args[]) {
    // INSERT CODE HERE
    }
}
```

a `for (Sports val:Sports.values()) System.out.print(val+":");`

b `for (Sports val:Sports.orderedValues()) System.out.print(val+":");`

c `for (Sports val:Sports.naturalValues()) System.out.print(val+":");`

d `for (Sports val:Sports.ascendingValues()) System.out.print(val+":");`

e None of the above

Q 2-4. Given that classes Outer and Test are defined in separate packages and source code files, which code options, when inserted independently at //INSERT CODE HERE, will instantiate class Inner in class Test? (Choose all that apply.)

```
// Source code-Outer.java
package ejava.ocp;
public class Outer {
    public static class Inner{}
}
```

```
// Source code-Test.java
package ejava.exams;
import static ejava.ocp.Outer.Inner;
class Test {
    //INSERT CODE HERE
}
```

- **a** `Inner inner = new Inner();`
- **b** `Outer.Inner inner = new Outer.Inner();`
- **c** `Outer.Inner inner = new Inner();`
- **d** `Outer.Inner inner = Outer.new Inner();`

Q 2-5. Given the following definition of classes `Outer` and `Inner`, select options that can be inserted individually at `//INSERT CODE HERE`. (Choose all that apply.)

```
class Outer {
    void aMethod() {
        class Inner {
            // INSERT CODE HERE
        }
    }
}
```

- **a** `protected Inner() {}`
- **b** `final static String name = "eJava";`
- **c** `static int ctr = 10;`
- **d** `private final void Inner() {}`
- **e** `Outer outer = new Outer();`
- **f** `Inner inner = new Inner();`
- **g** `static void print() {}`
- **h** `static final void print() {}`

Q 2-6. Given the following definition of enum `Size`, select the commented line number(s) in class `MyClass`, where you can insert the enum definition individually. (Choose all that apply.)

```
enum Size {SMALL, MEDIUM, LARGE}
class MyClass {
    // line1
    void aMethod() {
        //line2
    }
    class Inner {
        //line3
    }
    static class StaticNested{
       //line4
    }
}
```

- a The code at (#1)
- b The code at (#2)
- c The code at (#3)
- d The code at (#4)

Q 2-7. Given the following code, which option, when inserted at `/*INSERT CODE HERE*/`, will instantiate an anonymous class referred to by the variable `floatable`?

```
interface Floatable {
    void floating();
}
class AdventureCamp {
    Floatable floatable = /*INSERT CODE HERE*/
}
```

- a `new Floatable();`
- b `new Floatable(){};`
- c `new Floatable() { public void floating() {}};`
- d `new Floatable() public void floating() {};`
- e `new Floatable(public void floating() ) {}};`
- f `new Floatable() { void floating() {}};`
- g None of the above

Q 2-8. What is the output of the following code? (Choose all that apply.)

```
interface Admissible {}                           // line1
class University {
    static void admit(Admissible adm) {
        System.out.println("admission complete");
    }
    public static void main(String args[]) {
        admit(new Admissible(){});               // line2
    }
}
```

- a The class prints `admission complete`.
- b The class doesn't display any output.
- c The class fails to compile.
- d `University` will print `admission complete` if code at (#2) is changed to the following:

  ```
  admit(new Admissible());
  ```

- e Class `University` instantiates an anonymous inner class.
- f If `Admissible` is defined as a class, as follows, the result of the preceding code will remain the same:

  ```
  class Admissible {}
  ```

Q 2-9. Which of the following statements are correct? (Choose all that apply.)

- **a** An abstract class must define variables to store the state of its object.
- **b** An abstract class might define concrete methods.
- **c** An abstract class might not define abstract methods.
- **d** An abstract class constructor might be called by its derived class.
- **e** An abstract class can extend another nonabstract class.
- **f** An abstract class can't define static final abstract methods.

Q 2-10. Which of the classes, when inserted at `//INSERT CODE HERE`, will create an instance of class `Inner`? (Choose all that apply.)

```
class Outer {
    class Inner {}
}
class Test {
    //INSERT CODE HERE
}
```

- **a** `Outer.Inner inner = new Outer.Inner();`
- **b** `Outer.Inner inner = Outer().new Inner();`
- **c** `Outer.Inner inner = Outer.new Inner();`
- **d** `Outer.Inner inner = new Outer().new Inner();`
- **e** `Outer.Inner inner = new Inner();`
- **f** `Outer outer = new Outer();`
  `Inner inner = new Outer.Inner();`
- **g** `Outer outer = new Outer();`
  `Outer.Inner inner = outer.new Inner();`
- **h** `Outer outer = new Outer();`
  `Inner inner = new outer.Inner();`

Q 2-11. Select the incorrect options. (Choose all that apply.).

- **a** An anonymous inner class always extends a class, implicitly or explicitly.
- **b** An anonymous inner class might not always implement an interface.
- **c** An anonymous inner class is a direct subclass of class `java.lang.Object`.
- **d** You can make an anonymous inner class do both—explicitly extend a user-defined class and an interface.
- **e** An anonymous inner class can implement multiple user-defined interfaces.

Q 2-12. Select the correct options for the classes `Satellite` and `Moon`:

```
abstract class Satellite{
    static {
        ctr = (int)Math.random();              // line1
    }
    static final int ctr;                      // line2
}
class Moon extends Satellite{
    public static void main(String args[]) {
        System.out.println(Moon.ctr);          // line3
    }
}
```

- **a** Code at only (#1) fails compilation.
- **b** Code at either (#1) or (#2) fails compilation.
- **c** Code at (#3) fails compilation.
- **d** Code compiles and executes successfully.

Q 2-13. What is the output of the following code?

```
enum BasicColor {
    RED;
    static {
        System.out.println("Static init");
    }
    {
        System.out.println("Init block");
    }
    BasicColor(){
        System.out.println("Constructor");
    }
    public static void main(String args[]) {
        BasicColor red = BasicColor.RED;
    }
}
```

a
```
Init block
Constructor
Static init
```

b
```
Static init
Init block
Constructor
```

c
```
Static init
Constructor
Init block
```

d
```
Constructor
Init block
Static init
```

Q 2-14. What is the output of the following code?

```
enum Browser {
    FIREFOX("firefox"),
    IE("ie"){public String toString() {return "Internet Browser";}},
    NETSCAPE("netscape");
    Browser(String name){}
    public static void main(String args[]) {
        for (Browser browser:Browser.values())
            System.out.println(browser);
    }
}
```

- **a**
  ```
  FIREFOX
  Internet Browser
  NETSCAPE
  ```
- **b**
  ```
  FIREFOX
  INTERNET BROWSER
  NETSCAPE
  ```
- **c**
  ```
  FIREFOX
  IE
  NETSCAPE
  ```
- **d**
  ```
  firefox
  ie
  netscape
  ```

Q 2-15. What is the output of the following code?

```
class Base {
    static {
        System.out.print("STATIC:");
    }
    {
        System.out.print("INIT:");
    }
}
class MyClass extends Base {
    static {
        System.out.print("static-der:");
    }
    {
        System.out.print("init-der:");
    }
    public static void main(String args[]) {
        new MyClass();
    }
}
```

- **a** `STATIC:INIT:static-der:init-der:`
- **b** `INIT:STATIC:init-der:static-der:`
- **c** `STATIC:static-der:INIT:init-der:`
- **d** `static-der:init-der:STATIC:INIT:`

Q 2-16. Select the correct statements. (Choose all that apply.).

- **a** An abstract class can't define static final variables.
- **b** The abstract methods defined in an abstract base class must be implemented by all its derived classes.

c An abstract class enforces all its concrete derived classes to implement its abstract behavior.

d An abstract class might not define static methods.

e The initialization of the final variables defined in an abstract base class can be deferred to its derived classes.

ANSWERS TO SAMPLE EXAM QUESTIONS

A 2-1. e

[2.3] Use the static and final keywords
[2.5] Use enumerated types

Explanation: The code compiles successfully. An enum can define and use multiple constructors. The declaration of enum constants must follow the opening brace of the enum declaration. It can't follow the definition of variables or methods.

A 2-2. a, b

[2.1] Identify when and how to apply abstract classes
[2.2] Construct abstract Java classes and subclasses

Explanation: When a class extends another class or implements an interface, the methods in the derived class must be either valid overloaded or valid overridden methods.

Option (c) is incorrect. The concrete class `Her` extends `Foo` and implements `Run`. To compile `Her`, it must implement `run()` with `public` access so that it implements `run()` with default access in class `Foo` and `run()` with `public` access in interface `Run`:

```
class Her extends Foo implements Run {
    public void run() {}
}
```

Because class `Her` in option (c) defines `run()` with default access, it fails to implement public `run()` from interface `Run` and fails compilation.

Option (d) is incorrect. Method `run()` defined in class `His` and method `run()` defined in class `Foo` don't form either valid overloaded or overridden methods.

A 2-3. e

[2.5] Use enumerated types

Explanation: Option (a) is incorrect. The code in this option will print the natural order of the definition of the enum (the order in which they were defined):

```
TENNIS:CRICKET:BASKETBALL:SWIMMING:
```

Options (b), (c), and (d) define nonexistent enum methods. Code in these options won't compile.

A 2-4. a

[2.4] Create top-level and nested classes
[2.5] Use enumerated types

Explanation: Due to the following static `import` statement, only the nested static class `Inner` is visible in class `Test`:

```
import static ejava.ocp.Outer.Inner;
```

Class `Outer` isn't visible in class `Test`. Options (b) and (c) will instantiate class `Inner` if the following `import` statement is included in class `Test`:

```
import ejava.ocp.Outer;
```

A 2-5. a, b, d, e, f

[2.4] Create top-level and nested classes

Explanation: You can define final static variables in a method local inner class, but you can't define non-final static variables, static methods, or static final methods. You can define constructors with any access modifier in a local inner class.

A 2-6. a, d

[2.4] Create top-level and nested classes
[2.5] Use enumerated types

Explanation: You can't define an enum within a method or a nonstatic inner class.

A 2-7. c

[2.4] Create top-level and nested classes

Explanation: To instantiate an anonymous class that can be referred to by the variable `floatable` of type `Floatable`, the anonymous class must implement the interface, implementing all its methods.

Because interface `Floatable` defines method `floating()` (methods defined in an interface are implicitly public and abstract), it must be implemented by the anonymous class. Only option (c) correctly implements method `floating()`. Following is its correctly indented code, which should be more clear:

```
new Floatable() {
    public void floating() {
    }
};
```

Option (b) doesn't implement `floating()`. Option (a) tries to instantiate the interface `Floatable`, which isn't allowed. Option (f) looks okay, but isn't because it makes the method `floating()` more restrictive by not defining it as public.

A 2-8. a, e, f

[2.4] Create top-level and nested classes

Explanation: Options (b) and (c) are incorrect because the class compiles successfully and prints a value.

Option (d) is incorrect because it tries to instantiate the interface `Admissible` and not an instance of the anonymous inner class that implements `Admissible`.

Option (e) is correct because code at (#2) instantiates an anonymous inner class, which implements the interface `Admissible`. Because the interface `Admissible` doesn't define any methods, code at (#2) doesn't need to implement any methods.

Option (f) is correct. If `Admissible` is defined as a class, the anonymous inner class at (#2) will subclass it. Because `Admissible` doesn't define any abstract methods, there aren't any added complexities.

A 2-9. b, c, e, f

[2.1] Identify when and how to apply abstract classes

Explanation: Note the use of can, must, and might or might not in these options. Note that this isn't a test on English grammar or vocabulary. The meaning of an exam question will completely change depending on whether it uses "can" (feasible), "must" (mandatory), or "might" (optional) in a question.

Option (a) is incorrect because it isn't mandatory for an abstract class to define instance variables.

Option (d) is incorrect because the constructor of all base classes—concrete or abstract—must be called by their derived classes.

Option (f) is a correct statement. The combination of `final` and `abstract` modifiers is incorrect. An abstract method is meant to be overridden in the derived classes, whereas a final method can't be overridden.

A 2-10. d, g

[2.4] Create top-level and nested classes

Explanation: An inner class is a member of its outer class and can't exist without its instance. To create an instance of an inner class outside either the outer or inner class, you must do the following:

- If using a reference variable, use type `Outer.Inner`.
- Access an instance of the `Outer` class.
- Create an instance of the `Inner` class.

To create instances of both the `Outer` and `Inner` classes on a single line, you must use the operator `new` with both the `Outer` class and `Inner` class, as follows:

```
new Outer().new Inner();
```

If you already have access to an instance of `Outer` class, say, `outer`, call `new Inner()` by using `outer`:

```
outer.new Inner();
```

A 2-11. c, d, e

[2.4] Create top-level and nested classes

Explanation: Note that you have to select incorrect statements in this question. This can be confusing, because most of the time on the exam, you're asked to select correct options.

Option (a) is a correct statement. If an anonymous inner class extends a class, it subclasses it explicitly. When it implements an interface, it implicitly extends class `java.lang.Object`. If an anonymous inner class doesn't subclass a class implicitly, it implicitly extends `java.lang.Object`.

Option (b) is a correct statement. An anonymous inner class might not always implement an interface.

Option (c) is an incorrect statement. An anonymous inner class isn't *always* a direct subclass of class `java.lang.Object`, if it extends any other class explicitly.

Option (d) is an incorrect statement. You can't make an anonymous inner class extend both a class and an interface explicitly. I deliberately used the words *user-defined classes* and *user-defined interfaces* so you wouldn't assume that an anonymous class implicitly subclasses a class from a Java API.

Option (e) is an incorrect statement. You can't make an anonymous class implement multiple interfaces explicitly.

A 2-12. d

[2.2] Construct abstract Java classes and subclasses
[2.3] Use the static and final keywords

Explanation: No compilation or runtime issues exist with this code. A static initializer block can access and initialize a static variable; it can be placed before the static variable declaration. A static variable defined in a base class is accessible to its derived class. Even though class `Moon` doesn't define the static variable `ctr`, it can access the static variable `ctr` defined in its base class `Satellite`.

A 2-13. a

[2.3] Use the static and final keywords
[2.5] Use enumerated types

Explanation: The creation of enum constants happens in a static initializer block, before the execution of the rest of the code defined in the `static` block. Here's the decompiled code for enum `BasicColor`, which shows how enum constants are initialized in the static block. To initialize an enum constant, its constructor is called. Note that the contents of the default constructor and instance initializer blocks are added to the new constructor implicitly defined during the compilation process:

```
final class BasicColor extends Enum
{
    public static BasicColor[] values()
    {
        return (BasicColor[])$VALUES.clone();
    }

    public static BasicColor valueOf(String s)
    {
        return (BasicColor)Enum.valueOf(BasicColor, s);
    }

    private BasicColor(String s, int i)
    {
        super(s, i);
        System.out.println("Init block");
        System.out.println("Constructor");
    }

    public static void main(String args[])
    {
        BasicColor basiccolor = RED;
    }

    public static final BasicColor RED;
    private static final BasicColor $VALUES[];

    static
    {
        RED = new BasicColor("RED", 0);
        $VALUES = (new BasicColor[] {
            RED
        });
        System.out.println("Static init");
    }
}
```

NOTE If you try to compile the preceding code it won't compile because classes can't directly extend java.lang.Enum. I've included this code just to show you how the Java compiler modifies code of an enum and adds additional code to it. It explains why an enum constructor executes before its static block.

A 2-14. a

[2.5] Use enumerated types

Explanation: An enum extends class java.lang.Enum, which extends class java.lang.Object. Each enum constant inherits method toString() defined in class java.lang.Enum. Class java.lang.Enum overrides method toString() to return the enum constant's name.

An enum constant can override any of the methods that are inherited by it. The enum `Browser` defines a constructor that accepts a `String` method parameter, but it doesn't use it. All enum constants, except enum constant `IE`, print the name of the constant itself.

A 2-15. c

[2.3] Use the static and final keywords

Explanation: When you instantiate a derived class, the derived class instantiates its base class. The static initializers execute when a class is loaded in memory. So the order of execution of static and instance initializer blocks is as follows:

- 1) Base class static initializer block
- 2) Derived class static initializer block
- 3) Base class instance initializer block
- 4) Derived class instance initializer block

A 2-16. c, d

[2.1] Identify when and how to apply abstract classes
[2.3] Use the static and final keywords

Explanation: Option (a) is incorrect. An abstract class can define static final variables.

Option (b) is incorrect. The abstract methods defined in an abstract base class must be implemented by all its concrete derived classes. Abstract derived classes might not implement the abstract methods from their abstract base class.

Option (e) is incorrect. The initialization of a final variable defined in an abstract base class must complete in the class itself—with its initialization, in its initializer block, or in its constructor. It can't be deferred to its derived class.