* The instance initializer blocks are executed after an implicit or explicit call to the parent classes constructor:

class Instrument {

Instrument() {

System.out.println("Instrument:constructor");

}

}

class Pencil extends Instrument {

public Pencil() {

System.out.println("Pencil:constructor");

}

{

System.out.println("Pencil:instance initializer");

}

public static void main(String[] args) {

new Pencil();

}

}

The output of the preceding code is

Instrument:constructor

Pencil:instance initializer

Pencil:constructor

RETURN TYPE:

Methods can’t be defined as overloaded methods if they only differ in their return types, as follows:

class Result {

**double** calcAverage(int marks1, int marks2) {

return (marks1 + marks2)/2;

}

**int** calcAverage(int marks1, int marks2) {

return (marks1 + marks2)/2;

}

}

The methods defined in the preceding code aren’t correctly overloaded methods—they won’t compile.

ACCESS MODIFIER

Methods can’t be defined as overloaded methods if they only differ in their access modifiers, as follows:

class Result {

**public** double calcAverage(int marks1, int marks2) {

return (marks1 + marks2)/2;

}

**protected** double calcAverage(int marks1, int marks2) {

return (marks1 + marks2)/2;

}

}

NONACCESS MODIFIER

Methods can’t be defined as overloaded methods if they only differ in their non-access modifiers, as follows:

class Result {

public **synchronized** double calcAverage(int marks1, int marks2) {

return (marks1 + marks2)/2;

}

public **final** double calcAverage(int marks1, int marks2) {

return (marks1 + marks2)/2;

}

}

EXAM TIP

If a parent or child class defines static initializer block(s), the execution start from parent then child class static initializer block.

ACCESS MODIFIERS

A derived class can assign the same or more access but not a weaker access to the overriding

method in the derived class:

class Book {

**protected** void review(int id, List names) {}

}

class CourseBook extends Book {

void review(int id, List names) {}//wrong we shouldn’t decrease the scope of access modifiers

**public** void review(int id, List names) {}//correct

}

**Won’t compile; overriding methods in derived classes can’t use a weaker access.**

NONACCESS MODIFIERS

A derived class can’t override a base class method marked as final:

class Book {

final void review(int id, List names) {}

}

class CourseBook extends Book {

void review(int id, List names) {}

}

**Won’t compile; final methods can’t be overridden.**

ARGUMENT LIST AND COVARIANT RETURN TYPES

When the overriding method returns a subclass of the return type of the overridden method, it’s known as a covariant return type. To override a method, the parameter list of the methods in the base and derived classes must be exactly the same. It you try to use covariant types in the argument list, you’ll end up overloading the methods and not overriding them.

For example

class Book {

void review(int id, **List names**) throws Exception {

System.out.println("Base:review");

}

}

class CourseBook extends Book {

void review(int id, **ArrayList names**) throws IOException {

System.out.println("Derived:review");

}

}

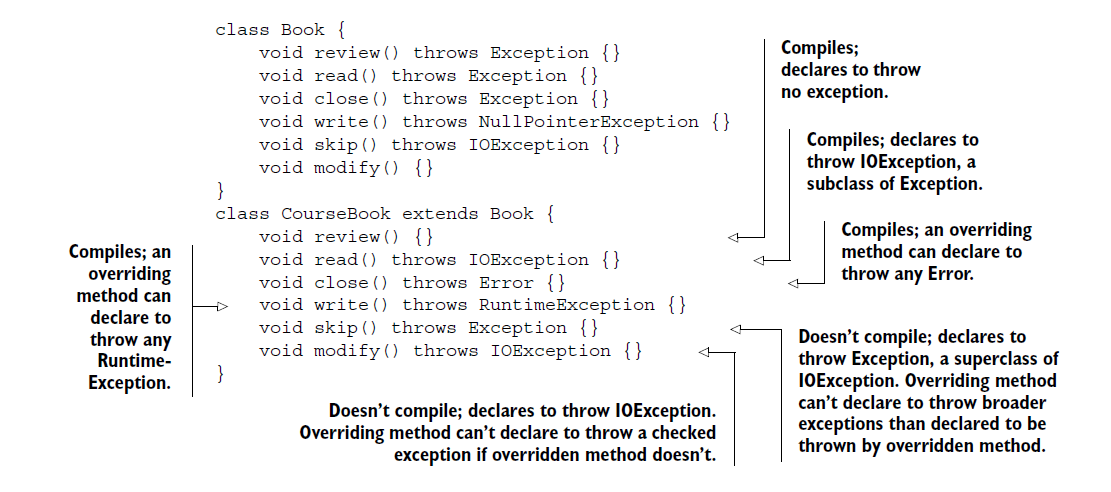
**Not overriding its overloading.**

EXCEPTIONS THROWN

An overriding method must either declare to throw no exception, the same exception, or a subtype of the exception declared in the base class method, or else it will fail to compile. This rule, however, doesn’t apply to error classes or runtime exceptions. We can throw any error or runtime exceptions.

EXAM TIP

An overriding method can declare to throw any Runtime-Exception or Error, even if the overridden method doesn’t.



CAN YOU OVERRIDE ALL METHODS FROM THE BASE CLASS OR INVOKE THEM VIRTUALLY?

The simple answer is no. You can override only the following methods from the base class:

* Methods accessible to a derived class
* Non static base class methods

METHODS ACCESSIBLE TO A BASE CLASS

The accessibility of a method in a derived class depends on its access modifier. For example, a private method defined in a base class isn’t available to any of its derived classes. **Also,** **a method with default access in a base class isn’t available to a derived class in another package.** A class can’t override the methods that it can’t access.

ONLY NONSTATIC METHODS CAN BE OVERRIDDEN

If a derived class defines a static method with the same name and signature as the one defined in its base class, it hides its base class method and doesn’t override it. You can’t override static methods. For example

class Book {

static void printName() {

System.out.println("Book");

}

}

class CourseBook extends Book {

static void printName() {

System.out.println("CourseBook");

}

}

Method printName() in class CourseBook hides printName() in class Book. It doesn’t override it. Because the static methods are bound at compile time, the method print-Name() that’s called depends on the type of the reference variable:

class BookExampleStaticMethod {

public static void main(String[] args) {

Book base = new Book();

base.printName();

Book derived = new CourseBook();

derived.printName();

}

}

*IDENTIFYING METHOD OVERRIDING, OVERLOADING, AND HIDING*

When a class extends another class, it can overload, override, or hide its base class methods. A class can’t override or hide its own methods—it can only overload its own methods.

class Book{

static void print(){}

}

class CourseBook extends Book{

static void print(){}

}//method hiding

-------------------------------

class Book{

static void print(){}

}

class CourseBook extends Book{

void print(){}

}// **compile time error** (instance method cannot override the static method)

-------------------------------------

class Book{

final void print(){}

}

class CourseBook extends Book{

void print(){}

}// **compile time error** (we can’t override final methods)

----------------------------------

class Book{

void print(){}

}

class CourseBook extends Book{

static void print(){}

}//**compile time error** (static method cannot hide the instance method from Book)

------------------------------

class Book{

void print(){}

}

class CourseBook extends Book{

void print(){}

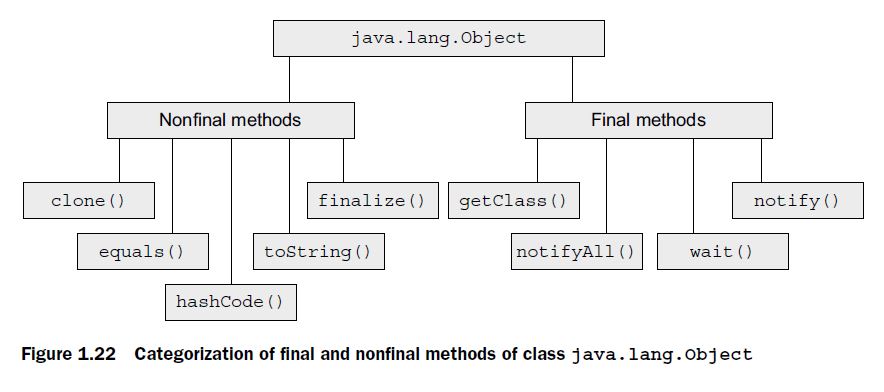
}// perfect overriding

*CAN YOU OVERRIDE BASE CLASS CONSTRUCTORS?*

The simple answer is no. Constructors aren’t inherited by a derived class. Because only inherited methods can be overridden, constructors cannot be overridden by a derived class. If you attempt an exam question that queries you on overriding a base class constructor, you know that it’s trying to trick you.

*EXAM TIP*

Constructors can’t be overridden because a base class constructor isn’t inherited by a derived class.



Following is the code of method toString(), as defined in class Object in the Java API:

public String toString() {

return getClass().getName() + "@" + Integer.toHexString(hashCode());

}

*TWIST IN THE TALE 1.1*

Here are the classes written by Shreya and Harry (residing in separate source code files) that work without any issues:

package library; // Class written by Shreya

public class Book {

protected String author;

}

package building; // Class written by Harry

import library.Book;

class StoryBook extends Book {

{ author = "Selvan"; }

}

On Friday evening, Shreya modified her code and checked it in to the organization’s version control system. Do you think Harry would be able to run his code without any errors when he checks out the modified code on Monday morning, and why? Here’s the modified code:

package library; // Class written by Shreya

class Book {

protected String author;

}

package building; // Class written by Harry

import library.Book;

class StoryBook extends Book {

{ author = "Selvan"; }

}

Nonpublic classes can’t be accessible in different package.

RULES FOR OVERRIDING METHOD EQUALS ()

1. “if x.equals(y) returns true, y.equals(x) must return true”.
2. “x.equals(null) should return false”. It shouldn’t throw NPE.

CORRECT AND INCORRECT OVERRIDING OF METHOD EQUALS ()

Note that the type of parameter passed to equals() is Object. Watch out for exam questions that seem to override equals(), passing it to a parameter type of the class in which it’s defined. In the following example, class Course doesn’t override method equals(), it overloads it:

class Course {

String title;

Course(String title) {

this.title = title;

}

public boolean equals(**Course o**) {//**Course doesn’t override toString(), it overloads it.**

return title.equals(o.title);

}

public static void main(String args[]) {

Object c1 = new Course("eJava");

Object c2 = new Course("eJava");

System.out.println(c1.equals(c2));// **Prints “false”**

}

}

EXAM TIP Use Object as the parameter type to equals(). Using any other type will overload equals().

THE NEED TO OVERRIDE METHOD HASHCODE()

Method hashCode() returns a hash-code value for an object, which is used to efficiently store and retrieve values in collection classes that use hashing algorithms, such as HashMap. Hashing algorithms identify the buckets in which they would store the objects and from which they would retrieve them. A well-written method hashCode() ensures that objects are evenly distributed in these buckets. Objects with the same hash-code values are stored in the same bucket. To retrieve an object, its bucket is identified using its hash-code value. If the bucket contains multiple objects, method equals() is used to find the target object.

OVERRIDING METHOD HASHCODE () CORRECTLY

To override method hashCode() correctly, you must follow the below rules:

1. If two objects are equal according to equals(Object)method, then calling hashCode()method on each of the two objects must produce the same integer result.
2. It’s not required that if two objects are unequal according to method equals(Object), that calling method hashCode() on each of the two objects must produce distinct integer results.

EXAM TIP Read the questions on method hashCode() carefully. You might be questioned on incorrect, inappropriate, or inefficient overriding of hashCode(). For Example

class MyNumber {

long number;

MyNumber(long number) {this.number = number;}

public int hashCode() {

return 1654;

}

}

In the preceding code, method hashCode() returns the same hash-code value for all the objects of MyNumber. This essentially stores all the values in the same bucket.

EFFECTS OF USING MUTABLE OBJECTS AS KEYS

Java recommends using immutable objects as keys for collection classes that use the hashing algorithm. What if you don’t?

class MyNumber {

int number;

MyNumber(int number) {this.number = number;}

public int hashCode() {

return number;

}

public boolean equals(Object o) {

if (o != null && o instanceof MyNumber)

return (number == ((MyNumber)o).number);

else

return false;

}

public static void main(String args[]) {

Map<MyNumber, String> map = new HashMap<>();

MyNumber num1 = new MyNumber(2500);

map.put(num1, "Shreya");

num1.number = 100;// *Modify field number of key num1*

System.out.println(map.get(num1));// *Prints “null”—can’t locate object with modified key.*

}

}

Implicit down casting isn’t allowed. You can’t assign reference variables of a base class to reference variables of its derived classes.

EXAM TIP In the absence of explicit casting, you’ll never get ClassCast-Exception—a RuntimeException.

EXAM TIP You can explicitly cast null to any type. It won’t generate a compilation error or throw a ClassCastException. Book book = (Book)null;

EXAM TIP If you cast an instance to a class outside its inheritance tree, you’ll get a compiler error. If you cast an instance to a class within its inheritance tree, but the types don’t match at runtime, the code will throw a ClassCastException.

*USING THE INSTANCEOF OPERATOR*

The *instanceof* operator is used to logically test whether an object is a valid type of a class or an interface. You should proceed with explicit casting only if this operator returns true, or you risk running into a *ClassCastException* at runtime.

EXAM TIP The operator *instanceof* returns false if the reference variable being compared to is null.

EXAM TIP The *instanceof* operator *never* throws a runtime exception; it returns either true or false. If the instanceof operator uses inconvertible types, the code won’t compile.

Course c = new Course();

Student s = new Student();

System.out.println(c instanceof Student);

**//Won’t compile—can’t use instanceof to compare inconvertible types.**

The instanceof operator is preceded by a value (literal value or a variable name) and

is followed by a class, interface, or enum name.

System.out.println(null instanceof Course);

// **Prints “false”—null can’t be an instance of any class.**

EXAM TIP The literal value null isn’t an instance of any class. So <referenceVariable> instanceof <ClassName> will return false.

NOTE The *import* statement doesn’t embed the contents of the imported class in your class, which means that importing more classes doesn’t increase the size of your own class. It lets you use the simple name for a class or interface defined in a separate package.

It’s important to note that you can’t use the import statement to use multiple classes or interfaces with the same names from different packages. For example, the Java API defines class Date in two commonly used packages: java.util and java.sql. To define variables of these classes in a class, use their fully qualified names with the variable declaration:

class AnnualExam {

java.util.Date date1;

java.sql.Date date2;

}

An attempt to use an import statement to import both these classes in the same class will not compile:

import java.util.Date;

import java.sql.Date;

class AnnualExam { }//**Code to import classes with same name from different packages won’t compile**

EXAM TIP Members of a named package can’t access classes and interfaces defined in the *default* package.

STATIC IMPORTS

You can import an individual static member of a class or an interface, or all its static members, by using the import static statement. Though accessible using an instance, the static members are usually accessed by prefixing their name with the class or interface names. By using static import, you can drop the prefix and just use the name of the static variable or method.

To use all **public** (only public variable or methods can accessed in different package) and static members of class ExamQuestion in class AnnualExam without importing each of them individually.

import static certification.ExamQuestion.\*;

REVIEW NOTES

* A top-level class, interface, or enum can only be defined using the public or default access. They can’t be defined using protected or private access.
* Overloaded methods are bound at compile time. Unlike overridden methods they’re bound at runtime.
* Overloaded methods might define a different return type or access or non-access modifier, but they can’t be defined with only a change in their return types or access or non-access modifiers.
* If present, the call to another constructor must be the first statement in a constructor.
* Whenever you intend to override methods in a derived class, use the annotation @Override. It will warn you if a method can’t be overridden properly or if you’re actually overloading a method rather than overriding it.
* Overridden methods can define the same or covariant return types.
* Static methods can’t be overridden. They’re not polymorphic and they’re bound at compile time.
* In a derived class, a static method with the same signature as that of a static method in its base class hides the base class method.
* A derived class can’t override the base class methods that aren’t accessible to it, such as private methods. Similarly, base constructor is not part of inheritance or base constructor can’t be accessible in derived class, so we can’t override constructor’s but we can overload constructor’s.
* The members of the default package are accessible only to classes or interfaces defined in the same directory on your system. A class from the default package can’t be used in any named packages.
* An import statement can’t be placed before a package statement in a class. Any attempt to do so will cause the compilation of the class to fail.
* The declaration of private variables to store the state of an object is encouraged.
* Expose the functionality of your classes using public methods.
* Create private methods to work as helper methods for the public methods.
* An instance of a non-final class can be casted to **any interface type** using an explicit cast during the compilation phase. But the exact object types are validated during runtime and a ClassCastException is thrown if the object’s class doesn’t implement that interface.

class Bird{}

class Parrot extedns Bird{}

Parrot p = (Parrot)bird;// ClassCastException

class Bird{}

class Parrot{}

Bird b = (Bird)parrot;//Compilation fail’s because objects are in different hierarchy

interface Scavenger{}

class Bird{}

Scavenger sc2 = (Scavenger)bird;// no compilation error, but ClassCastException

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

*Advanced class design*

* Should my application be allowed to create instances of Animal class? If no, define Animal class as an abstract class.
* Does class Animal include behavior that’s common to all its derived classes? If yes, define the relevant method as an abstract method and class Animal as an abstract class.

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

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.

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.

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;**

}

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**

*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.

EXAM TIP Unlike instance variables, which are initialized for each instance, static class variables are initialized only once, when they are loaded into JVM 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.

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.

The static methods can’t access non-static fields or non-static methods directly.

**static int x = count();**

int count() { return 10; }

// Compilation error. Static variable or methods should not access instance variables or methods directly.

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. Static variables can’t be initialized using the constructors of a class.

class AffiliateProgram {

private static int accountOpenBonus;

static {

accountOpenBonus = 5;

}

}

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.

* You can’t prefix the definition of a top-level class or an interface with the keyword static. But define inner class or interface as static.

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.
* To initialize final static variable we can use static block, to initialize final variable we can use constructor.

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;

}

}

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");// **Won’t compile**

}

}

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.

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() {}// **Private methods aren’t inherited.**

}

class Derived extends Base {

final void 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 overriding method, but a new method.

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.”

***Enumerated types***

Users are allowed to use only *existing* enum objects; they can’t create new enum objects. ***Type safety***was the main reasonfor introducing enumerated types in Java version 5.0.

***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 }

-----------------------------------------------------------------------

final class Level extends Enum// **enum is implicitly declared as final.**

{

public static final Level BEGINNER;

public static final Level INTERMEDIATE;

public static final Level EXPERT;

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

private static final Level $VALUES[];

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

static

{

BEGINNER = new Level("BEGINNER", 0);

INTERMEDIATE = new Level("INTERMEDIATE", 1);

EXPERT = new Level("EXPERT", 2);

$VALUES = (new Level[] {

BEGINNER, INTERMEDIATE, EXPERT

});

}

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

public static Level[] values()

{

return (Level[])$VALUES.clone();

}

// **return an array of all enum constants.**

public static Level valueOf(String s)

{

return (Level)Enum.valueOf(Level, s);

}

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

private Level(String s, int i)

{

super(s, i);

}

// **Private constructor**

}

All enums in Java extend the abstract class java.lang.Enum, defined in the Java API.

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;

}

// **Default implementation of toString() returns name of enum constant**

public final String name() {

return name;

}

//.. rest of the code

}

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

Note both methods—toString() and name() defined in java.lang.Enum—return the value of the 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 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.

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 }

But you can make your enum implement any number of interfaces.

You can’t explicitly make a class extend java.lang.Enum:

class MyClass extends java.lang.Enum {}

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.

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.

EXAM TIP An enum can’t define a constructor with public or protected access level. We can define default or private.

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.

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.
* 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 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 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.

***Static nested and inner classes***

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);

//**Object of StaticNested class required to access its instance members**

System.out.println(Outer.StaticNested.innerStatic);

//**Object of StaticNested class not required to access its static members**

}

}

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.

A static nested class can access only the static members of its outer class. Can’t access instance variables from a static nested class directly.

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.
* A static nested class can define both static and non-static members. But non static inner class can define only non-static 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 non-static 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).

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

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.

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 in **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 **non-final 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.

Inner classes save you from defining new classes.

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.

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.

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.

For Inner and static Inner classes: Outer$Inner.class, and Outer$Inner.class.

For method/local Inner class: Outer$1Inner.class, and Outer$2Inner.class.

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. But a method local inner class can’t define static variables or static methods. Method local inner can’t be defined using an explicit access modifier always default.

* 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.
* You can’t define an enum within a method or a non-static inner class.
* An anonymous inner class always extends a class, implicitly (When it implements an interface, it implicitly extends class java.lang.Object.) or explicitly.
* You can’t make an anonymous class implement multiple interfaces explicitly.

***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 non-abstract 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 non-abstract-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.

***Non-access 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 non-static variables and methods of a class. But the reverse works: non-static 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.

***Non-access 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 non-static 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 non-final 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.

***Object-oriented design principles***

interface Runner{

int speed();

double distance = 70;

}

Becomes

interface Runner{

public abstract int speed();

public static final double distance = 70;

}

All methods of an interface are implicitly public and abstract, and its variables are implicitly public, static, and final. Why do you think these implicit modifiers are added to the interface members? Because an interface is used to define a contract, it doesn’t make sense to limit access to its members—and so they are implicitly public. An interface can’t be instantiated, and so the value of its variables should be defined and accessible in a static context, which makes them implicitly static. Because an interface is a contract, its implementations shouldn’t be able to change it, so the interface variables are implicitly final. Interface methods are implicitly abstract so that it’s mandatory for the classes to implement them.

EXAM TIP The declaration of an interface can’t include a class name. An interface can never extend any class.

Can you define a top-level, *protected* interface? No, you can’t.

You can declare a *top-level interface,* with only the following access levels:

* Public
* No modifier (default access)

EXAM TIP All the top-level Java types (classes, enums, and interfaces) can be declared using only two access levels: public and default. Inner or nested types can be declared using any access level.

All members of an interface—variables, methods, inner interfaces, and inner classes are public by default. Interfaces support only the public access modifier. Using other access modifiers results in compilation errors.

Because the interface variables are implicitly final, you can define only *constants* in an interface. Ensure that you initialize these constants, or your code won’t compile:

interface MyInterface {

int number; **//Won’t compile;**

}

You can declare a top-level interface with only the following nonaccess modifiers:

* Abstract
* strictfp

NOTE The strictfp keyword guarantees that results of all floating-point calculations are identical on all platforms.

If you try to declare your top-level interfaces by using the other nonaccess modifiers (final, static, transient, synchronized, or volatile), the interface will fail to compile. All of the following interface declarations fail to compile:

EXAM TIP A *concrete* class must implement all the methods from the interfaces that it implements. An *abstract* class can choose not to implement all the methods from the interfaces that it implements.

A class can define an instance or a static variable with the same name as the variable defined in the interface that it implements.

interface Livable {

boolean status = true;

int ratings = 10;

}

class Home implements Livable {

boolean status;

static int ratings = 7;

Home() {

System.out.println(status);//**false**

System.out.println(Livable.status);//**true**

System.out.println(ratings);//7

System.out.println(Livable.ratings);//11

}

}

EXAM TIP A class can define an instance or a static variable with the same name as the variable defined in the interface that it implements. These variables can be defined using any access level.

EXAM TIP Because interface methods are implicitly public, the implementing class must implement them as public methods, or else the class will fail to compile.

EXAM TIP A class can implement methods with the same name from multiple interfaces. But these must qualify as correctly overloaded methods.

* A class can *implement* multiple interfaces. An interface can *extend* multiple interfaces.
* If a class implements multiple interfaces that define methods with the same name, the interface methods must either qualify as correctly overloaded or overridden methods, or else the class won’t compile.

FRAGILE DERIVED CLASSES

Adding to or modifying a base class can affect its derived classes. Adding new methods to a base class can result in breaking the code of a derived class. Consider this initial arrangement, which works well:

public abstract class Animal {

void move(){}

}

class Lion extends Animal {

void live(){}

}

Now consider a modified arrangement: a new method live() is added to base class Animal. Because live() clashes (because of an incorrectly overridden method) with the existing method live() in its derived class Lion, Lion will no longer compile:

public abstract class Animal {

void move(){}

String live(){

return "live";

}

}

class Lion extends Animal {

void live(){}//**live() in Lion neither overloads nor overrides live() in Animal.**

}

*Preferring Class Inheritance Over Interface Inheritance*

Class inheritance scores better when you want to reuse the implementation already defined in a base class. It also scores better when you want to add new behavior to an existing base class.

*Preferring interface inheritance over class inheritance*

You may prefer interface inheritance over class inheritance when you need to define multiple contracts for classes.

* IMPLEMENTING MULTIPLE INTERFACES
* FRAGILE DERIVED CLASSES

EXAM TIP Class inheritance isn’t always a good choice because derived classes are fragile. If any changes are made to a base class, a derived class might break. Extending classes that are from another package or are poorly documented aren’t good candidates for base classes.

If a base class chooses to modify the implementation details of its methods, the derived classes might not be able to offer the functionality they were supposed to, or they might respond differently. Consider this initial arrangement:

public abstract class Animal {

String currentPosition;

public void move(String newPosition){

currentPosition = newPosition;

}

}

class Lion extends Animal {

void changePosition(String newPosition) {

super.move(newPosition);

System.out.println("New Position:" + newPosition);

}

}

class Test{

public static void main(String args[]) {

new Lion().changePosition("Forest");

}

}

Imagine that Animal adds another line of code to method move(). Let’s see how it changes the code output of class Test (modification in bold):

public abstract class Animal {

String currentPosition;

public void move(String newPosition){

currentPosition = newPosition;

**System.out.println("New Position:" + newPosition);**

}

}

class Lion extends Animal {

void changePosition(String newPosition) {

super.move(newPosition);

System.out.println("New Position:" + newPosition);

}

public static void main(String args[]) {

new Lion().changePosition("Forest");

**//Prints “New Position:Forest” twice**

}

}

As you witnessed in the preceding example, changes to a base class can break the code of its derived classes.

EXAM TIP There isn’t any clear winner when it comes to selecting the better option from class inheritance and interface inheritance. Analyze the given conditions or situations carefully to answer questions on this topic.

EXAM TIP Representing IS-A and HAS-A relationships by using (quick) UML diagrams can help you on the exam. Though you may not see UML diagrams on the exam, creating quick UML diagrams on an erasable board (or something similar) provided to you during the exam will help you answer these questions.

EXAM TIP The key to finding the types that participate in an IS-A relationship is to find your way, up the hierarchy tree, in the direction of the arrows. This technique will not only help you with the exam, but also take you a long way in your professional career.

EXAM TIP Well-designed applications aim for loosely coupled classes and modules.

EXAM TIP Object composition enables you to use the existing functionality of classes without extending them. The approach is simple: create and use objects of other classes in your own class.

You should inherit a class only when you think that the derived class is a type of its base class. For example, it’s correct to say that RacingCar is a type of Car. But it’s incorrect to say that Engine is a type of Car.

EXAM TIP On the exam, all of these approaches (eager initialization, synchronization of the complete method getInstance(), and partial synchronization of method getInstance()) may be presented, and you may be questioned about the right approach for implementing the Singleton pattern. All these approaches are good. Beware of modified code that tries to synchronize a partial getInstance() method, which doesn’t synchronize the code that creates an object of Singleton.

USING ENUMS

By using enums, you can implement the Singleton pattern in a thread-safe manner.

Here’s a simple implementation:

public enum Singleton {

INSTANCE;

public void initCache(){

//..code

}

}

Because enum instances can’t be created by any other class, the enum Singleton will ensure the existence of only *one* of its instances, Singleton.INSTANCE.

NOTE The Singleton pattern is also referred to as an anti-pattern. It has been overused by developers and designers, who make a lot of assumptions about the applications that use it. **It also makes testing difficult**.

* Prefer method invocation over direct constructor calls
* Promote flexibility. Object instantiation logic can be changed without affecting the clients that use objects. They also allow addition of new concrete classes.

|  |  |  |
| --- | --- | --- |
| CLASS | METHOD | DESCRIPTION |
| java.util.Calendar | getInstance() | Gets a calendar using the default time zone and locale. |
| java.util.Arrays | asList() | Returns a fixed-size list backed by the specified array. |
| java.util.ResourceBundle | getBundle() | Overloaded versions of this method return a resource bundle using the specified base name, target locale, class loader, and control. |
| java.sql.DriverManager | getConnection() | Establishes and returns a connection to the given database URL. |
| java.sql.DriverManager | getDriver() | Attempts to locate and return a driver that understands the given URL. |
| java.sql.Connection | createStatement() | Overloaded version of this method creates a statement object for sending SQL statements to the database and generates ResultSet  objects with the given type, concurrency, and holdability. |
| java.sql.Statement | executeQuery() | Executes the given SQL statement, which returns a single ResultSet object. |
| java.text.NumberFormat | getInstance()  getNumberFormat() | Returns a general purpose number format for the current default locale. |
| java.text.NumberFormat | getCurrencyInstance() | Returns a currency format for the current default locale. |
| java.text.NumberFormat | getIntegerInstance() | Returns an integer format for the current default locale. |
| java.util.concurrent  .Executors | newFixedThreadPool()  newCachedThreadPool()  newSingleThreadExecutor() | Creates a thread pool. |

EXAM TIP The DAO pattern decouples classes that define business or presentation logic from the data persistence details.

EXAM TIP The exam might ask you whether it’s common to use the Factory pattern with the DAO pattern. The answer is yes (as shown in the next section). But it isn’t mandatory to use the Factory pattern with the DAO pattern (as shown in this section).

***Benefits of the DAO pattern***

The benefits of the DAO pattern are

* It abstracts and encapsulates all access to a data source. It manages the connection to the data source to obtain and store data.
* It promotes programming to an interface. It completely hides the data access implementation from its clients.
* It decouples the business logic layer and persistence layer. It makes the code independent of any changes to a data source or its vendor (for example, plaintext, XML, LDAP, MySQL, Oracle, or DB2).
* It promotes flexibility. Because the interfaces accessible to client classes don’t change, new implementation classes can be added.
* The DAO pattern might also include Factory pattern classes.
* It prevents tight coupling between client classes and DAO implementation classes. It promotes the creation of cohesive classes.