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

class’s 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

*WHEN METHODS CAN’T BE DEFINED AS OVERLOADED METHODS*

The overloaded methods give you the flexibility of defining methods with the same name that can be passed a different set of arguments. But it doesn’t make sense to define overloaded methods with a difference in only their return types or access or non-access modifiers.

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.

EXAM TIP When the Java compiler differentiates methods, it doesn’t consider their return types. So you can’t define overloaded methods with the same parameter list and different return types.

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 nonaccess

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;

}

}

Let’s revisit the rules for defining overloaded methods.

In the next section, we’ll create overloaded versions of special methods, called constructors, which are used to create objects of a class.

EXAM TIP If a parent or child class defines static initializer block(s), they execute before all parent and child class constructors and instance initializers—first for the parent and then for the child class.

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) {}

}

**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 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.**

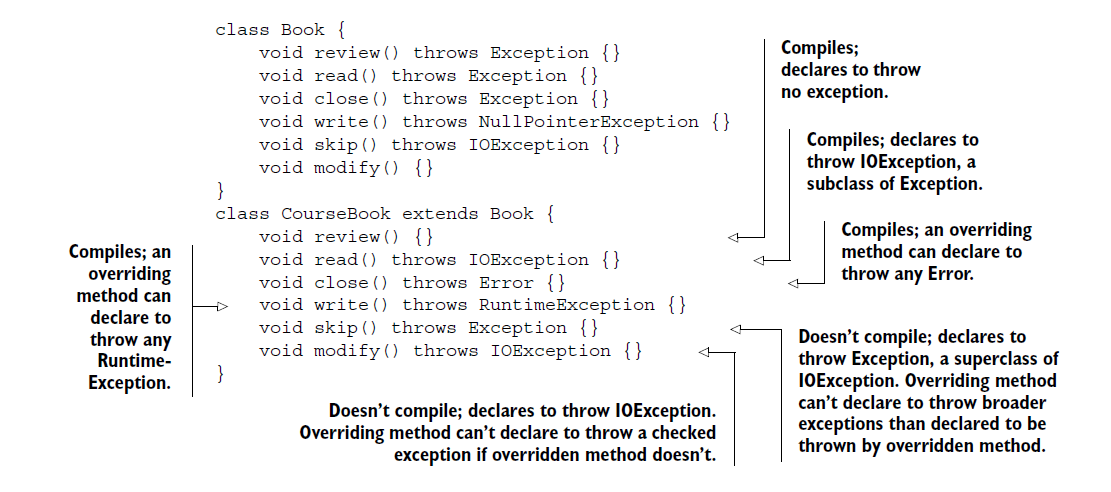
EXCEPTIONS THROWN

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

For example

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{

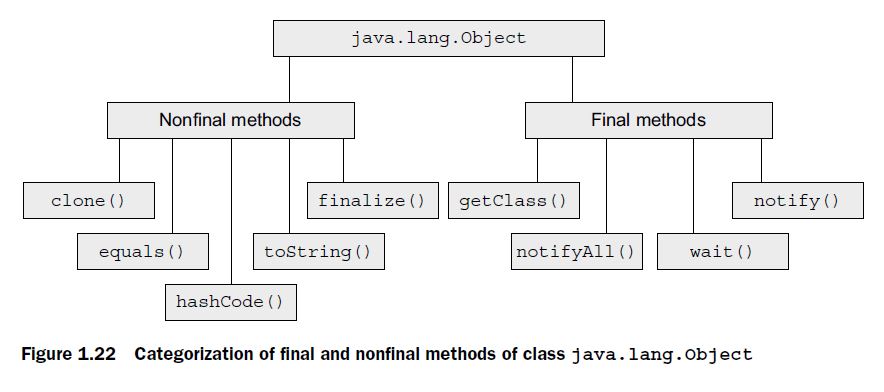
void print(){}

}// 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.