1Z0-808 Exam Topic Reviewer

TopicId: 1020
Topic: Inheritance and Method Overriding

August 5, 2025

Inheritance: The "is-a" Relationship

Welcome back. Today we're tackling Inheritance, one of the four main pillars of Object-Oriented Programming. For the 1Z0-808 exam, this isn't just theory—it's a minefield of tricky questions related to constructors, access rules, and method signatures. Pay close attention. Inheritance models an "is-a" relationship. A `Dog` is-an `Animal`. This allows us to reuse code and create a logical hierarchy, but the compiler enforces strict rules.

0.1 The `extends` Keyword and Code Reusability

We use the `extends` keyword to create a subclass (or child class) from a superclass (or parent class). The subclass inherits all `public` and `protected` members (fields and methods) from its superclass.

```
// Superclass
public class Animal {
    String name;
    public void eat() {
        System.out.println("This animal eats food.");
    }
}

// Subclass
public class Dog extends Animal {
    public void bark() {
        System.out.println("Woof!");
    }
}
```

Here, a `Dog` object can call both `bark()` and the inherited `eat()` method. **Exam Trap:** Java does not support multiple inheritance for classes. A class can only `extends` one other class.

0.2 Constructor Chaining: The First Rule of Subclassing

This is one of the most tested areas of inheritance. The rule is simple: **A constructor's very first action must be to call another constructor**. It can be a call to another constructor in the same class using `this()` or a call to a superclass constructor using `super()`.

- **Implicit Call:** If you do NOT provide an explicit call to `super()` or `this()`, the compiler will automatically insert a no-argument `super();` call for you.
- **Compile Error Trap:** If the superclass does *not* have a no-argument constructor (e.g., it only has a constructor that takes parameters), and you fail to explicitly call that constructor with the required arguments from the subclass constructor, your code will **fail to compile**.

Example of a Compile Error:

```
class Vehicle {
    String type;
    // No-argument constructor is NOT present.
    public Vehicle(String type) {
        this.type = type;
    }
}
class Car extends Vehicle {
    // This constructor implicitly tries to call super()
    // but Vehicle() does not exist!
    public Car() { // COMPILE ERROR!
        System.out.println("Car created");
    }
}
```

Correct Implementation:

```
class Car extends Vehicle {
    public Car() {
        super("Sedan"); // Explicitly call the parent constructor
        System.out.println("Car created");
    }
}
```

0.3 Method Overriding: Changing Behavior

Overriding allows a subclass to provide its own implementation of a method inherited from its superclass. To correctly override a method, you must follow these rules precisely. The exam will test every single one.

- **Method Signature:** Must be identical (same name, same number and type of parameters).
- **Return Type:** Must be the same or a **covariant return type**. A covariant return is a subtype of the original return type.

  ```
  // Superclass
  class Shape {
      public Object getInfo() { return new Object(); }
  }
  // Subclass with covariant return (String is a subtype of Object)
  class Circle extends Shape {
      @Override
      public String getInfo() { return "A circle"; } // VALID
  }
  ```

- **Access Modifier:** Cannot be more restrictive. The visibility must be the same or wider. (`public` ¿ `protected` ¿ `default` ¿ `private`)
- **Exceptions:** May throw fewer or narrower checked exceptions (subclasses of the original exception). Cannot throw new or broader checked exceptions.

- `final` **Methods:** Cannot be overridden.
- `static` **Methods:** Cannot be overridden. This is called **method hiding**. The subclass method just hides the parent one. We will explore this more with Polymorphism.
- `private` **Methods:** Cannot be overridden because they are not visible to the subclass.

The `@Override` annotation is your friend. It asks the compiler to check if you've correctly followed the rules. If not, you get a compile error. The exam may show code without it to trick you.

0.4 The `super` Keyword: Accessing Parent Members

The `super` keyword is a reference to the immediate parent class object.

- `super()`: Calls the parent class constructor (as we saw).
- `super.methodName()`: Calls the parent's version of an overridden method.
- `super.fieldName`: Accesses a parent's field, especially useful if the subclass has a field with the same name (hiding).

```
class Employee {
    String getDetails() { return "Employee"; }
}
class Manager extends Employee {
    @Override
    String getDetails() {
        return "Manager, " + super.getDetails(); // Calls Employee's getDetails()
    }
}
```

Key Takeaways for the 1Z0-808 Exam

- **Constructor Chaining is King:** Always check if a superclass has a no-arg constructor if you see a subclass that doesn't explicitly call `super(...)`.
- **Memorize Override Rules:** Access modifiers, return types (covariant!), and exceptions are common traps.
- **Static vs. Final:** `final` methods cannot be overridden. `static` methods cannot be overridden (they are hidden).
- **The `super` keyword:** Know its two primary uses: calling parent constructors and calling parent members.

Master these rules. The exam will give you code that looks plausible but violates one of these principles. Your job is to be the compiler and spot the error.