Java Constructor Chaining, Static & Instance Initializers, Anonymous Classes, Method Overriding, Nested Classes, Overloading

---

1. Constructor Chaining Basics

Example 1: Missing Default Constructor

```java
class Base {
    // No default constructor
    public Base(int x) {
        System.out.println("Base class constructor called with value: " + x);
    }
}
class Derived extends Base {
    public Derived(int x) {
        // super(x); // ERROR: Will not compile if not called, as there is no default const
        System.out.println("Derived class constructor called with value: " + x);
    }
}
public class Chaining {
    public static void main(String[] args) {
        Base obj1 = new Derived(10); // Compilation error if super(x) is not called
    }
}
```

Error: - If the `Base` class does not have a default constructor, the first statement in the `Derived` constructor must be a call to `super(...)` with appropriate arguments. - Otherwise, Java tries to call `super()` by default, which does not exist, causing a compilation error.

Example 2: Correct Chaining with super(x)

```java
class Base {
    public Base(int x) {
        System.out.println("Base class constructor called with value: " + x);
    }
}
class Derived extends Base {
    public Derived(int x) {
        super(x); // OK: Calls Base(int x)
        System.out.println("Derived class constructor called with value: " + x);
    }
}
```

```java
public class Chaining {
    public static void main(String[] args) {
        Base obj1 = new Derived(10); // Output: Base... then Derived...
    }
}
```

Example 3: Default Constructor in Base

```java
class Base {
    public Base() {
        System.out.println("Base class constructor called");
    }
}
class Derived extends Base {
    public Derived(int x) {
        System.out.println("Derived class constructor called with value: " + x);
    }
}
public class Chaining {
    public static void main(String[] args) {
        Base obj1 = new Derived(10); // Output: Base... then Derived...
    }
}
```

---

2. Constructor Chaining with this() and super()

Example 4: Chaining in Same Class

```java
class Base {
    public Base() {
        System.out.println("Base class constructor called");
    }
    public Base(int x) {
        this(); // Calls Base()
        System.out.println("Base class constructor called with value: " + x);
    }
}
class Derived extends Base {
    public Derived(int x) {
        super(x);
        System.out.println("Derived class constructor called with value: " + x);
    }
}
```

Example 5: Chaining in Derived Class

```
class Derived extends Base {
    public Derived(int x) {
        super(x);
        System.out.println("Derived class constructor called with value: " + x);
    }
    public Derived(int x, int y) {
        this(x); // Calls Derived(int x)
        System.out.println("Derived class constructor called with values: " + x + ", " + y);
    }
}
```

---

3. Static and Instance Initializer Blocks

Example 6: Static Initializer

```
class Base {
    static {
        System.out.println("Base class static block executed");
    }
    public Base() {
        System.out.println("Base class constructor called");
    }
}
```

- Static blocks run once when the class is loaded.

Example 7: Instance Initializer

```
class Base {
    { // Instance initializer
        System.out.println("Base class instance initializer block executed");
    }
    public Base() {
        System.out.println("Base class constructor called");
    }
}
```

- Instance blocks run every time an object is created, before the constructor.

---

4. Execution Order: Static, Instance, Constructors

Example 8: Full Order

```
class Base {
    static { System.out.println("Base static"); }
    { System.out.println("Base instance"); }
    public Base() { System.out.println("Base constructor"); }
}
class Derived extends Base {
    static { System.out.println("Derived static"); }
    { System.out.println("Derived instance"); }
    public Derived() { System.out.println("Derived constructor"); }
}
public class Chaining {
    public static void main(String[] args) {
        Base obj = new Derived();
    }
}
```

Output:

```
Base static
Derived static
Base instance
Base constructor
Derived instance
Derived constructor
```

---

5. Static Variable Initialization Order

Example 9: Static Variable Initialization

```
class Base {
    static int a = 10;
    static int b;
    static {
        System.out.println("Static block: a=" + a + ", b=" + b);
        b = 20;
    }
    static {
        System.out.println("Static block 2: b=" + b);
    }
}
```

- Static variables are initialized in the order they appear.

- Static blocks can access static variables, but variables declared after the block are not yet initialized.

---

6. Common Errors and Gotchas

- If a base class does not have a default constructor, every derived class constructor must explicitly call a base constructor with arguments.
- Only one call to `super()` or `this()` is allowed and must be the first statement in a constructor.
- Static blocks run only once per classloader, even if multiple objects are created.
- Instance blocks run every time an object is created, before the constructor.
- Static variables must be initialized before use in static blocks.
- Final static variables can be initialized in static blocks.

---

7. Additional Example: Multiple Constructors and Initializers

```java
class Example {
    static { System.out.println("Static block"); }
    { System.out.println("Instance block"); }
    public Example() { System.out.println("Default constructor"); }
    public Example(int x) { this(); System.out.println("Constructor with x: " + x); }
}
public class Test {
    public static void main(String[] args) {
        Example e1 = new Example();
        Example e2 = new Example(5);
    }
}
```

Output:

```
Static block
Instance block
Default constructor
Instance block
Default constructor
Constructor with x: 5
```

---

8. Why and What: When to Use Each

- **Constructor chaining**: To avoid code duplication and ensure proper initialization order.
- **Static blocks**: For static initialization logic, e.g., loading drivers, initializing static resources.
- **Instance blocks**: For common code shared by all constructors, before constructor logic.

---

9. Summary Table

Feature	When Runs	How Often	Use Case
Static block	Class load	Once per class	Static resource setup
Instance block	Object creation	Every object	Common init for all constructors
Constructor	Object creation	Every object	Custom object setup
Constructor chaining	Constructor call	As needed	Avoid code duplication

---

10. Practice: Predict the Output

Try to predict the output for the following code:

```
class A {
    static { System.out.println("A static"); }
    { System.out.println("A instance"); }
    public A() { System.out.println("A constructor"); }
}
class B extends A {
    static { System.out.println("B static"); }
    { System.out.println("B instance"); }
    public B() { System.out.println("B constructor"); }
}
public class Test {
    public static void main(String[] args) {
        A obj = new B();
    }
}
```

---

11) Anonymous Classes in Java

Anonymous classes in Java are used for creating a one-time, unnamed subclass of a class or implementing an interface on the fly. They are typically used when you need to override methods or provide specific behavior for a short-lived object, often as an argument to a method or for event handling, without formally declaring a new class.

Common Use Cases

- **Event listeners** (e.g., in GUI programming)
- **Runnable or Callable** implementations for threads
- **Customizing behavior** of objects for a single use

Example: Event Listener

```
Button btn = new Button();
btn.addActionListener(new ActionListener() {
    @Override
    public void actionPerformed(ActionEvent e) {
        System.out.println("Button clicked!");
    }
});
```

Example: Anonymous Class Extending a Class

```
class Base {
    static int baseStaticVar = 10;
    static {
        System.out.println("Base class static block executed");
    }
    {
        System.out.println(baseStaticVar2); // Will print as static variables are already in
        System.out.println("Base class instance initializer block executed");
    }
    static final int baseStaticVar2 = 20;
    public Base() {
        System.out.println("Base class constructor called");
    }
    public Base(int x) {
        this(); // Calls the default constructor
        System.out.println("Base class constructor called with value: " + x);
    }
}
class Derived extends Base {
    static {
        System.out.println("Derived class static block executed");
```

```
    }
    {
        System.out.println("Derived class instance initializer block executed");
    }
    public Derived(int x) {
        super(x);
        System.out.println("Derived class constructor called with value: " + x);
    }
    public Derived(int x, int y) {
        this(x);
        System.out.println("Derived class constructor called with values: " + x + ", " + y);
    }
}
public class Chaining {
    public static void main(String[] args) {
        // Anonymous class extending Derived for extra initialization
        Base obj1 = new Derived(10) {
            {
                System.out.println("Anonymous class extending Derived created");
            }
        };
    }
}
```

Output Explanation

- All static blocks execute first (Base, then Derived).
- Instance initializer blocks execute before constructors.
- The anonymous class block executes after the Derived constructor.

Why Use Anonymous Classes?

- To quickly override or extend behavior for a single use without creating a named class.
- Useful for callbacks, event handling, or customizing library classes on the fly.

Common Errors

- Anonymous classes cannot have explicit constructors, but can use instance initializer blocks `{ ... }` for setup.
- Cannot declare static initializers or static members (except static final constants) inside anonymous classes.

Additional Example: Runnable

```
Runnable r = new Runnable() {
    @Override
    public void run() {
        System.out.println("Running in a thread!");
    }
};
new Thread(r).start();
```

Anonymous classes are a powerful feature for concise, one-off customizations in Java!

---

12. Access Modifiers and Packages

Access modifiers control the visibility of classes, methods, and variables in Java: - `public`: Accessible from anywhere. - `private`: Accessible only within the same class. - `protected`: Accessible within the same package and subclasses. - Default (no modifier): Accessible within the same package, but not from subclasses in other packages.

Example:

```
public class Access {
    // ... see code above ...
}
```

Why: - To encapsulate and protect data, and to control API exposure.

---

13. Covariant Return Types

Java allows an overriding method to return a subtype (covariant type) of the return type declared in the original overridden method.

Example:

```
class Api {
    Number value() { return 42; }
}
class Impl extends Api {
    @Override
    Integer value() { return 7; } // Integer is a subclass of Number
}
```

Why: - Allows more specific return types in subclasses, improving type safety and usability.

---

14. Narrowing Visibility (Not Allowed)

You cannot reduce the visibility of an inherited method when overriding.

Example (Error):

```
class Api {
    protected Number value() { return 42; }
}
class Impl extends Api {
    @Override
    private Integer value() { return 7; } // ERROR: Cannot reduce visibility
}
```

Error: - "Cannot reduce the visibility of the inherited method from Api."

Why: - To ensure that the contract of the base class is preserved in subclasses.

---

15. Protected Methods and Package Access

Example:

```
// pkg1
package pkg1;
class Base {
    protected void hook() { System.out.println("Base hook"); }
}
// pkg2
package pkg2;
import pkg1.Base;
class Derived extends Base {
    void test(Derived otherDerived, Base baseRef) {
        this.hook(); // OK
        otherDerived.hook(); // OK
        baseRef.hook(); // ERROR: Not accessible if baseRef is not a Derived
    }
}
```

Why: - Protected methods are accessible in subclasses, but not via references of the base type from outside the package.

---

16. Method Overriding and Access

Public vs Private Methods

- If a method is `public` in the base class, it can be overridden in the subclass.
- If a method is `private`, it is not inherited and cannot be overridden.

Example:

```
class A {
    public void f() { System.out.println("A.f"); }
    public void call() { f(); }
}
class B extends A {
    public void f() { System.out.println("B.f"); }
}
A x = new B();
x.call(); // Prints B.f
```

If f() is private in A: - B's f() is a new method, not an override. - A.call() always calls A's own f().

Why: - Private methods are statically bound; public/protected are dynamically bound (polymorphism).

---

17. Overriding with Different Return Types

You can override a method and change the return type if the new return type is a subclass (covariant return type). You cannot change the access modifier to be more restrictive.

Example:

```
class A {
    public void f() { System.out.println("A.f"); }
    public void call() { f(); }
}
class B extends A {
    public void f() { System.out.println("B.f"); }
    // public int call() { f(); return 0; } // Allowed if return type is compatible
}
```

---

18. Overriding and Method Resolution

If both base and derived classes define a method with the same signature, the method in the derived class overrides the base class method (unless the base method is private).

Example:

```
class A {
    public void f() { System.out.println("A.f"); }
    public void call() { f(); }
}
class B extends A {
    public void f() { System.out.println("B.f"); }
    public void call() { f(); }
}
A x = new B();
x.call(); // Prints B.f
```

---

19. Package-Private and Private Constructors

- Classes or constructors with no modifier are package-private (accessible only within the same package).
- Private constructors are not accessible outside the class.

Example:

```
// lib
package lib;
class Helper { public Helper(){} }
// pkg2
package pkg2;
import lib.Helper;
public class User {
    public static void main(String[] args) {
        new Helper(); // ERROR: Not accessible
    }
}
```

Example (private constructor):

```
package lib;
public class Helper { private Helper(){} }
// ...
new Helper(); // ERROR: Constructor is private
```

---

20. Inner and Nested Classes

- **Inner class**: Non-static, can access all members (even private) of the outer class.
- **Static nested class**: Can only access static members of the outer class.

Example:

```
public class Outer {
    private int secret = 99;
    class Inner {
        private int innerSecret = 7;
        void show() {
            System.out.println("Outer class secret: " + secret);
            System.out.println("Inner class secret: " + innerSecret);
        }
    }
    static class Nested {
        void show() {
            System.out.println("Outer class secret: " + new Outer().secret);
        }
    }
    void probe() {
        Inner inner = new Inner();
        System.out.println(inner.innerSecret); // Accessible
    }
    public static void main(String[] args) {
        Outer outer = new Outer();
        outer.probe();
        new Nested().show();
        Inner inner = outer.new Inner();
        inner.show();
    }
}
```

Why: - Inner classes are used for logically grouping classes and for accessing outer class members.

---

21. Method Overloading and Resolution

Java resolves overloaded methods at compile time based on the argument types.

Example:

```
public class Outer {
    static void f(long x) { System.out.println("long"); }
    static void f(Integer x) { System.out.println("Integer"); }
    static void f(int... x) { System.out.println("int..."); }
    public static void main(String[] args) {
        f(1); // int -> long (widening)
        f((short)1); // short -> long (widening)
        f(Integer.valueOf(1)); // Integer
```

```
        f((byte)1); // byte -> long (widening)
        f(new int[]{1}); // int[] -> int... (varargs)
    }
}
```

Output:

```
long
long
Integer
long
int...
```

Why: - Java chooses the most specific applicable method. Widening is preferred over boxing, and varargs is the least preferred.

---

22. Java's Overload Resolution Priority

When the compiler tries to pick the best match for a method call, it follows this order:

1. **Exact primitive match (same type)** — highest priority
2. **Primitive widening** (e.g., `int` → `long`) byte → short → char → int → long → float → double
3. **Boxing/unboxing** (primitive wrapper)
4. **Varargs/Ellipsis** — lowest priority

Rule: - Primitive widening beats boxing, and boxing beats varargs.

Example: Overload Resolution

```
public class OverloadDemo {
    static void f(long x) { System.out.println("long"); }
    static void f(Integer x) { System.out.println("Integer"); }
    static void f(int... x) { System.out.println("int..."); }
    public static void main(String[] args) {
        f(1); // int → long (widening)
        f((short)1); // short → long (widening)
        f(Integer.valueOf(1)); // Integer
        f((byte)1); // byte → long (widening)
        f(new int[]{1}); // int[] → int... (varargs)
    }
}
```

Output:

```
long
long
```

```
Integer
long
int...
```

Why: - Java chooses the most specific applicable method. Widening is preferred over boxing, and varargs is the least preferred.

MCQ's - Access Modifiers

```
# Java Access Control & Overriding Questions

---

## 1) Can you narrow visibility when overriding?

**Files / packages:**

```java
// pkg1/Api.java
package pkg1;
public class Api {
    protected Number value() { return 42; }
}

// pkg2/Impl.java
package pkg2;
import pkg1.Api;

public class Impl extends Api {
    @Override
    Integer value() { return 7; } // Q: compile?
}
```

---

2) Protected across packages: "via-subclass" vs "via-reference"

Files / packages:

```java
// pkg1/Base.java
package pkg1;
public class Base {
    protected void hook() { System.out.println("Base.hook"); }
}
```
```

```java
// pkg2/Child.java
package pkg2;
import pkg1.Base;

public class Child extends Base {
    void test(Child otherChild, Base baseRef) {
        this.hook();        // (A) ok?
        otherChild.hook();  // (B) ok?
        baseRef.hook();     // (C) ok?
    }
}
```

Question: Which lines compile?

---

3) "Private methods aren't inherited" (so this isn't overriding)

```java
class A {
    private void f() { System.out.println("A.f"); }
    public void call() { f(); }     // calls A.f
}

class B extends A {
    public void f() { System.out.println("B.f"); }
    public static void main(String[] args) {
        A x = new B();
        x.call();                   // what prints?
    }
}
```

---

4) Public constructor, but class itself isn't public

Files / packages:

```java
// lib/Helper.java
package lib;
class Helper {          // package-private top-level class
    public Helper() {}  // constructor is public
}
```

```java
// app/App.java
package app;
import lib.Helper;
```

```java
public class App {
    public static void main(String[] args) {
        new Helper();   // Q: compile?
    }
}
```

---

5) Private of inner vs outer; static nested vs inner

```java
public class Outer {
    private int secret = 99;

    class Inner {
        private int innerSecret = 7;
        void show() {
            System.out.println(secret);       // (A) ok?
        }
    }
    static class Nested {
        void show(Outer o) {
            System.out.println(o.secret);     // (B) ok?
        }
    }
    void probe() {
        Inner in = new Inner();
        System.out.println(in.innerSecret);   // (C) ok?
    }
    public static void main(String[] args) {
        new Outer().new Inner().show();
        new Nested().show(new Outer());
        new Outer().probe();
    }
}
```

Question: Do (A), (B), and (C) compile and run?