Chapter6 Class Design

Monday, October 13, 2025 11:27 AM

Modifier	Description	Chapter covered
`final`	The class may not be extended.	Chapter 6
`abstract`	The class is abstract, may contain `abstract` methods, and requires a concrete subclass to instantiate.	Chapter 6
`sealed`	The class may only be extended by a specific list of classes.	Chapter 7
`non-sealed`	A subclass of a sealed class permits potentially unnamed subclasses.	Chapter 7
`static`	Used for `static` nested classes defined within a class.	Chapter 7

```
public class Mammal {}

public final class Rhinoceros extends Mammal {}

public class Clara extends Rhinoceros {}  // DOES NOT COMPILE
```

On the exam, pay attention to any class marked `final`. If you see another class extending it, you know immediately the code does not compile.

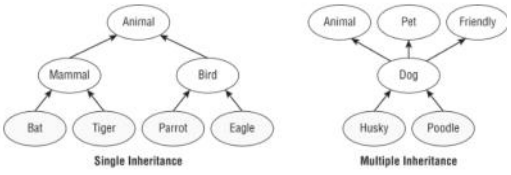
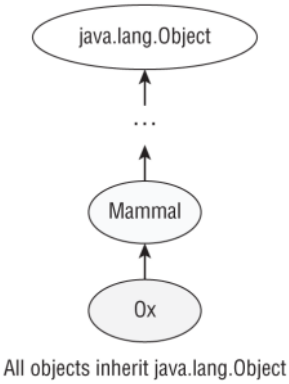


FIGURE 6.2 Types of inheritance



All objects inherit java.lang.Object

Trying to declare a top-level class with `protected` or `private` class will lead to a compiler error, though.

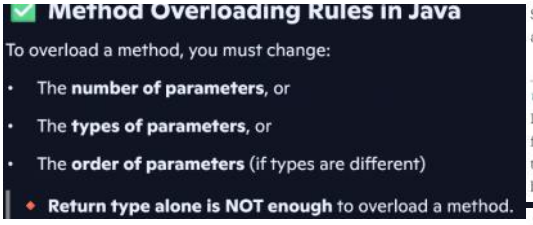
```
// ClownFish.java
protected class ClownFish{} // DOES NOT COMPILE

// BlueTang.java
private class BlueTang {} // DOES NOT COMPILE
```

Does that mean a class can never be declared `protected` or `private`? Not exactly. In Chapter 7, we present nested types and show that when you define a class inside another, it can use any access modifier.

!!!Attention

Like method parameters, constructor parameters can be any valid class, array, or primitive type, including generics, but may not include `var`. For example, the following does not compile:

```
public class Bonobo {
   public Bonobo(var food) {  // DOES NOT COMPILE
   }
}
```



Calling `this()` has one special rule you need to know. If you choose to call it, the `this()` call must be the first statement in the constructor. The side effect of this is that there can be only one call to `this()` in any constructor.

```
3:    public Hamster(int weight) {
4:       System.out.println("chew");
5:       // Set weight and default color
6:       this(weight, "brown");     // DOES NOT COMPILE
7:    }
```

Even though a print statement on line 4 doesn't change any variables, it is still a Java statement and is not allowed to be inserted before the call to `this()`. The comment on line 5 is just fine. Comments aren't considered statements and are allowed anywhere.

Here we summarize the rules you should know about constructors that we covered in this section. Study them well!

- A class can contain many overloaded constructors, provided the signature for each is distinct.
- The compiler inserts a default no-argument constructor if no constructors are declared.
- If a constructor calls `this()`, then it must be the first line of the constructor.
- Java does not allow cyclic constructor calls.

```
   public Hamster(int weight) {  // Second constructor
      this(weight, "brown");
   }
```

Success! Now Java calls the constructor that takes two parameters, with `weight` and `color` set as expected.

THIS VS. THIS()

Despite using the same keyword, `this` and `this()` are very different. The first, `this`, refers to an instance of the class, while the second, `this()`, refers to a constructor call within the class. The exam may try to trick you by using both together, so make sure you know which one to use and why.

There's one last rule for overloaded constructors that you should be aware of. Consider the following definition of the `Gopher` class:

```
public class Gopher {
   public Gopher(int dugHoles) {
      this(5);  // DOES NOT COMPILE
   }
}
```

The compiler is capable of detecting that this constructor is calling itself infinitely. This is often referred to as a *cycle* and is similar to the infinite loops that we discussed in Chapter 3, "Making Decisions." Since the code can never terminate, the compiler stops and reports this as an error. Likewise, this also does not compile.

```
public class Gopher {
   public Gopher() {
      this(5);  // DOES NOT COMPILE
   }
   public Gopher(int dugHoles) {
      this();   // DOES NOT COMPILE
   }
}
```

Here we summarize the rules you should know about constructors that we covered in this section. Study them well!

- A class can contain many overloaded constructors, provided the signature for each is distinct.
- The compiler inserts a default no-argument constructor if no constructors are declared.
- If a constructor calls `this()`, then it must be the first line of the constructor.
- Java does not allow cyclic constructor calls.

```
    this(5);  // DOES NOT COMPILE
  }
  public Gopher(int dugHoles) {
    this();   // DOES NOT COMPILE
  }
}
```

Like calling `this()`, calling `super()` can only be used as the first statement of the constructor. For example, the following two class definitions will not compile:

```
public class Zoo {
   public Zoo() {
      System.out.println("Zoo created");
      super();     // DOES NOT COMPILE
   }
}

public class Zoo {
   public Zoo() {
      super();
      System.out.println("Zoo created");
      super();     // DOES NOT COMPILE
   }
}
```

The first class will not compile because the call to the parent constructor must be the first statement of the constructor. In the second code snippet, `super()` is the first statement of the constructor, but it is also used as the third statement. Since `super()` can only be called once as the first statement of the constructor, the code will not compile.

We conclude this section by adding two constructor rules to your skill set.

- If a constructor calls `super()` or `this()`, then it must be the first line of the constructor.
- If the constructor does not contain a `this()` or `super()` reference, then the compiler automatically inserts `super()` with no arguments as the first line of the constructor.

Initialize Class X

1. Initialize the superclass of X.
2. Process all `static` variable declarations in the order in which they appear in the class.
3. Process all `static` initializers in the order in which they appear in the class.

Let's try an example with no inheritance. See if you can figure out what the following application outputs:

```
1:  public class ZooTickets {
2:     private String name = "BestZoo";
3:     { System.out.print(name + "-"); }
4:     private static int COUNT = 0;
5:     static { System.out.print(COUNT + "-"); }
6:     static { COUNT += 10; System.out.print(COUNT + "-"); }
7:
8:     public ZooTickets() {
9:        System.out.print("z-");
10:    }
11:
12:    public static void main(String... patrons) {
13:       new ZooTickets();
14:    } }
```

The output is as follows:

```
0-10-BestZoo-z-
```

Initialize Instance of X

1. Initialize *Class* X if it has not been previously initialized.
2. Initialize the superclass instance of X.
3. Process all instance variable declarations in the order in which they appear in the class.
4. Process all instance initializers in the order in which they appear in the class.
5. Initialize the constructor, including any overloaded constructors referenced with `this()`.

cuted. Remember, constructors are executed from the bottom up, but since the first line of every constructor is a call to another constructor, the flow ends up with the parent constructor executed before the child constructor.

```
1:  class GiraffeFamily {
2:     static { System.out.print("A"); }
3:     { System.out.print("B"); }
4:
5:     public GiraffeFamily(String name) {
6:        this(1);
7:        System.out.print("C");
8:     }
9:
10:    public GiraffeFamily() {
11:       System.out.print("D");
12:    }
13:
14:    public GiraffeFamily(int stripes) {
15:       System.out.print("E");
16:    }
17: }
18: public class Okapi extends GiraffeFamily {
19:    static { System.out.print("F"); }
20:
21:    public Okapi(int stripes) {
22:       super("sugar");
23:       System.out.print("G");
24:    }
25:    { System.out.print("H"); }
26:
27:    public static void main(String[] grass) {
28:       new Okapi(1);
29:       System.out.println();
30:       new Okapi(2);
31:    }
```

AFBECGH HG
BECGH HG

!! First instantiate all instance fields(including all block initializing) in order, and then execute constructor. This is after the super call from constructor.

To override a method, you must follow a number of rules. The compiler performs the following checks when you override a method:

1. The method in the child class must have the same signature as the method in the parent class.
2. The method in the child class must be at least as accessible as the method in the parent class.
3. The method in the child class may not declare a checked exception that is new or broader than the class of any exception declared in the parent class method.
4. If the method returns a value, it must be the same or a subtype of the method in the parent class, known as *covariant return types*.

While these rules may seem confusing or arbitrary at first, they are needed for consistency. Without these rules in place, it is possible to create contradictions within the Java language.

Remember that a method signature is composed of the name of the method and method parameters. It does not include the return type, access modifiers, optional specifiers, or any declared exceptions.

METHOD OVERRIDING INFINITE CALLS

You might be wondering whether the use of `super` in the previous example was required. For example, what would the following code output if we removed the `super` keyword?

```
public double getAverageWeight() {
   return getAverageWeight()+20;  // StackOverflowError
}
```

In this example, the compiler would not call the parent `Marsupial` method; it would call the current `Kangaroo` method. The application will attempt to call itself infinitely and produce a `StackOverflowError` at runtime.

Rule #3: Checked Exceptions

The third rule says that overriding a method cannot declare new checked exceptions or checked exceptions broader than the inherited method. This is done for polymorphic reasons similar to limiting access modifiers. In other words, you could end up with an object that is more restrictive than the reference type it is assigned to, resulting in a checked exception that is not handled or declared. One implication of this rule is that overridden methods are free to declare any number of new unchecked exceptions.

Let's try an example:

```
public class Reptile {
   protected void sleep() throws IOException {}

   protected void hide() {}

   protected void exitShell() throws FileNotFoundException {}
}

public class GalapagosTortoise extends Reptile {
   public void sleep() throws FileNotFoundException {}

   public void hide() throws FileNotFoundException {} // DOES NOT CO

   public void exitShell() throws IOException {}      // DOES NOT CO
}
```

A simple test for covariance is the following: given an inherited return type A and an overriding return type B, can you assign an instance of B to a reference variable for A without a cast? If so, then they are covariant. This rule applies to primitive types and object types alike. If one of the return types is `void`, then they both must be `void`, as nothing is covariant with `void` except itself.

!! Does not work for primitive types.
Exp int and short.
Even:
short b= 1;
int a = b
In Overriding this will throw an error.
It works only with reference objects, and for primitive types need to be the exact return type.

```
public class Bear {
   public static void sneeze() {
      System.out.println("Bear is sneezing");
   }
   public void hibernate() {
      System.out.println("Bear is hibernating");
   }
   public static void laugh() {
      System.out.println("Bear is laughing");
   }
}

public class SunBear extends Bear {
   public void sneeze() {           // DOES NOT COMPILE
      System.out.println("Sun Bear sneezes quietly");
   }
   public static void hibernate() { // DOES NOT COMPILE
      System.out.println("Sun Bear is going to sleep");
   }
   protected static void laugh() {  // DOES NOT COMPILE
      System.out.println("Sun Bear is laughing");
   }
}
```

Easy so far. But there are some rules you need to be aware of:

- Only instance methods can be marked `abstract` within a class, not variables, constructors, or `static` methods.
- An abstract class can include zero or more abstract methods, while a non-abstract class cannot contain any.
- A non-abstract class that extends an abstract class must implement all inherited abstract methods.
- Overriding an abstract method follows the existing rules for overriding methods that you learned about earlier in the chapter.

Like the `final` modifier, the `abstract` modifier can be placed before or after the access modifier in class and method declarations, as shown in this `Tiger` class:

```
abstract public class Tiger {
   abstract public int claw();
}
```

The `abstract` modifier cannot be placed after the `class` keyword in a class declaration or after the return type in a method declaration. The following `Bear` and `howl()` declarations do not compile for these reasons:

```
public class abstract Bear {    // DOES NOT COMPILE
   public int abstract howl();  // DOES NOT COMPILE
}
```

```
public abstract class Animal {
   abstract String getName();
}

public abstract class BigCat extends Animal {
   protected abstract void roar();
}

public class Lion extends BigCat {
   public String getName() {
      return "Lion";
   }
   public void roar() {
      System.out.println("The Lion lets out a loud ROAR!");
   }
}
```

While it is not possible to declare a method `abstract` and `private`, it is possible (albeit redundant) to declare a method `final` and `private`.

Creating Constructors in Abstract Classes

Even though abstract classes cannot be instantiated, they are still initialized through constructors by their subclasses. For example, consider the following program:

```
abstract class Mammal {
   abstract CharSequence chew();
   public Mammal() {
      System.out.println(chew());  // Does this line compile?
   }
}

public class Platypus extends Mammal {
   String chew() { return "yummy!"; }
   public static void main(String[] args) {
      new Platypus();
   }
}
```

```
public abstract class Turtle {
   public abstract long eat()      // DOES NOT COMPILE
   public abstract void swim() {}; // DOES NOT COMPILE
   public abstract int getAge() {  // DOES NOT COMPILE
      return 10;
   }
   public abstract void sleep;     // DOES NOT COMPILE
   public void goInShell();        // DOES NOT COMPILE
}
```

```
public abstract final class Tortoise {  // DOES NOT COMPILE
   public abstract final void walk();   // DOES NOT COMPILE
}
```

abstract and *private* Modifiers

A method cannot be marked as both `abstract` and `private`. This rule makes sense if you think about it. How would you define a subclass that implements a required method if the method is not inherited by the subclass? The answer is that you can't, which is why the compiler will complain if you try to do the following:

```
public abstract class Whale {
   private abstract void sing();  // DOES NOT COMPILE
}
```

sible (albeit redundant) to declare a method `final` and `private`.

```
public abstract class Whale {
   private abstract void sing();  // DOES NOT COMPILE
}

public class HumpbackWhale extends Whale {
   private void sing() {
      System.out.println("Humpback whale is singing");
   } }
```

Creating immutable objects.

```
import java.util.*;
public final class Animal {  // Not an immutable object declaration
   private final ArrayList<String> favoriteFoods;

   public Animal() {
      this.favoriteFoods = new ArrayList<String>();
      this.favoriteFoods.add("Apples");
   }

   public List<String> getFavoriteFoods() {
      return favoriteFoods;
   } }
```

```
import java.util.*;
public final class Animal {  // An immutable object declaration
   private final List<String> favoriteFoods;

   public Animal() {
      this.favoriteFoods = new ArrayList<String>();
      this.favoriteFoods.add("Apples");
   }

   public int getFavoriteFoodsCount() {
      return favoriteFoods.size();
   }

   public String getFavoriteFoodsItem(int index) {
      return favoriteFoods.get(index);
   } }
```

We carefully followed the first three rules, but unfortunately, a malicious caller could still modify our data.

```
var zebra = new Animal();
System.out.println(zebra.getFavoriteFoods());  // [Apples]

zebra.getFavoriteFoods().clear();
zebra.getFavoriteFoods().add("Chocolate Chip Cookies");
System.out.println(zebra.getFavoriteFoods());  // [Chocolate Chip Co
```

```
var favorites = new ArrayList<String>();
favorites.add("Apples");

var zebra = new Animal(favorites);  // Caller still has access to fa
System.out.println(zebra.getFavoriteFoodsItem(0));  // [Apples]

favorites.clear();
favorites.add("Chocolate Chip Cookies");
System.out.println(zebra.getFavoriteFoodsItem(0));  // [Chocolate Ch
```

Whoops! It seems like `Animal` is not immutable anymore, since its contents can change after it is created. The solution is to make a copy of the list object containing the same elements.

```
public Animal(List<String> favoriteFoods) {
   if (favoriteFoods == null || favoriteFoods.size() == 0)
      throw new RuntimeException("favoriteFoods is required");
   this.favoriteFoods = new ArrayList<String>(favoriteFoods);
}
```

The copy operation is called a *defensive copy* because the copy is being made in case other code does something unexpected. It's the same idea as defensive driving: prevent a problem before it exists. With this approach, our `Animal` class is once again immutable.

2. Compile-time type vs. Runtime type

Java separates **what's *visible*** at compile time from **what's *executed*** at runtime.

Stage	Controlled by	Determines
Compile-time	Declared type (`A`)	What methods and fields you *can access*
Runtime	Actual object (`B`)	What *implementation* gets executed

3. Why this rule exists — type safety

Imagine if this were allowed:

```
java

A obj = new A();
obj.showB(); // allowed? (A doesn't have showB)
```

That would blow up at runtime, because not all `A` objects are `B`.

To prevent this, Java's type system enforces:

> You can only call members that are guaranteed to exist in the declared type.

This keeps Java **statically type-safe**.

6. Quick analogy

Think of the declared type as the *contract* and the actual object as the *employee*:

- You hire someone (`obj`) under the contract "A".
- Even if the person *knows more* (`B` skills), you can only *legally* ask them to do tasks written in the contract (A's methods).
- At runtime, they might perform their task in their own unique way (B's override).

!!EXAM

1. E,D -> Doesn't understood deeply the requirements.
2. A,B,F ->
3. B,C
4. F
5. ~~C~~ E

5. Which of the following completes the constructor so that this code prints out `50`?

```
class Speedster {
   int numSpots;
}
public class Cheetah extends Speedster {
   int numSpots;

   public Cheetah(int numSpots) {
      // INSERT CODE HERE
   }

   public static void main(String[] args) {
      Speedster s = new Cheetah(50);
      System.out.print(s.numSpots);
   }
```

5. E. The code compiles, making option F incorrect. An instance variable with the same name as an inherited instance variable is hidden, not overridden. This means that both variables exist, and the one that is used depends on the location and reference type. Because the `main()` method uses a reference type of `Speedster` to access the `numSpots` variable, the variable in the `Speedster` class, not the `Cheetah` class, must be set to `50`. Option A is incorrect, as it reassigns the method parameter to itself. Option B is incorrect, as it assigns the method parameter the value of the instance variable in `Cheetah`, which is `0`. Option C is incorrect, as it assigns the value to the instance variable in `Cheetah`, not `Speedster`. Option D is incorrect, as it assigns the method parameter the value of the instance variable in `Speedster`, which is `0`.

```
    // INSERT CODE HERE
  }

  public static void main(String[] args) {
    Speedster s = new Cheetah(50);
    System.out.print(s.numSpots);
  }
}
```

A. `numSpots = numSpots;`
B. `numSpots = this.numSpots;`
C. `this.numSpots = numSpots;`
D. `numSpots = super.numSpots;`
E. `super.numSpots = numSpots;`
F. The code does not compile regardless of the code inserted into the constructor.
G. None of the above.

correct, as it assigns the method parameter the value of the instance variable in `Cheetah`, which is `0`. Option C is incorrect, as it assigns the value to the instance variable in `Cheetah`, not `Speedster`. Option D is incorrect, as it assigns the method parameter the value of the instance variable in `Speedster`, which is `0`. Options A, B, C, and D all print `0` at runtime. Option E is the correct answer, as it assigns the instance variable `numSpots` in the `Speedster` class a value of `50`. The `numSpots` variable in the `Speedster` class is then correctly referenced in the `main()` method, printing `50` at runtime.

6. D,E
7. A
8. D
9. E B

9. Which of the following statements about overridden methods are true? (Choose all that apply.)
   A. An overridden method must contain method parameters that are the same or covariant with the method parameters in the inherited method.
   B. An overridden method may declare a new exception, provided it is not checked.
   C. An overridden method must be more accessible than the method in the parent class.
   D. An overridden method may declare a broader checked exception than the method in the parent class.
   E. If an inherited method returns `void`, then the overridden version of the method must return `void`.
   F. None of the above.

9. B, E. The signature must match exactly, making option A incorrect. There is no such thing as a covariant signature. An overridden method must not declare any new checked exceptions or a checked exception that is broader than the inherited method. For this reason, option B is correct, and option D is incorrect. Option C is incorrect because an overridden method may have the same access modifier as the version in the parent class. Finally, overridden methods must have covariant return types, and only `void` is covariant with `void`, making option E correct.

10. A,C
11. Tacbf -> C
12. B C

12. How many lines of the following program contain a compilation error?

```
1:  public class Rodent {
2:    public Rodent(Integer x) {}
3:    protected static Integer chew() throws Exception
4:      System.out.println("Rodent is chewing");
5:      return 1;
6:    }
7:  }
8:  class Beaver extends Rodent {
9:    public Number chew() throws RuntimeException {
10:     System.out.println("Beaver is chewing on wood
11:     return 2;
12:   } }
```

A. None
B. 1
C. 2
D. 3
E. 4
F. 5

12. C. The code doesn't compile, so option A is incorrect. The first compilation error is on line 8. Since `Rodent` declares at least one constructor, and it is not a no-argument constructor, `Beaver` must declare a constructor with an explicit call to a `super()` constructor. Line 9 contains two compilation errors. First, the return types are not covariant since `Number` is a supertype, not a subtype, of `Integer`. Second, the inherited method is `static`, but the overridden method is not, making this an invalid override. The code contains three compilation errors, although they are limited to two lines, making option C the correct answer.

13. A,G
14. B,E F

14. Which of the following statements about inheritance are correct? (Choose all that apply.)
   A. A class can directly extend any number of classes.
   B. A class can implement any number of interfaces.
   C. All variables inherit `java.lang.Object`.
   D. If class `A` is extended by `B`, then `B` is a superclass of `A`.
   E. If class `C` implements interface `D`, then `C` is a subtype of `D`.
   F. Multiple inheritance is the property of a class to have multiple direct superclasses.

a subtype of that interface, making option E correct. Finally, option F is correct as it is an accurate description of multiple inheritance, which is not permitted in Java.

15. C,
16. uq uq uqcrcrm -> D
17. C,F,F

17. Which of the following are true? (Choose all that apply.)
   A. `this()` can be called from anywhere in a constructor.
   B. `this()` can be called from anywhere in an instance method.
   C. `this.variableName` can be called from any instance method in the class.
   D. `this.variableName` can be called from any `static` method in the class.
   E. You can call the default constructor written by the compiler using `this()`.
   F. You can access a `private` constructor with the `main()` method in the same class.

17. C, F. Calling an overloaded constructor with `this()` may be used only as the first line of a constructor, making options A and B incorrect. Accessing `this.variableName` can be performed from any instance method, constructor, or instance initializer, but not from a `static` method or `static` initializer. For this reason, option C is correct, and option D is incorrect. Option E is tricky. The default constructor is written by the compiler only if no user-defined constructors were provided. And `this()` can only be called from a constructor in the same class. Since there can be no user-defined constructors in the class if a default constructor was created, it is impossible for option E to be true. Since the `main()` method is in the same class, it can call `private` methods in the class, making option F correct.

18. D,F
19. F

19. What is the output of the following code?

```
1:  class Reptile {
2:    {System.out.print("A");}
3:    public Reptile(int hatch) {}
4:    void layEggs() {
5:      System.out.print("Reptile");
6:    } }
7:  public class Lizard extends Reptile {
8:    static {System.out.print("B");}
9:    public Lizard(int hatch) {}
10:   public final void layEggs() {
11:     System.out.print("Lizard");
12:   }
13:   public static void main(String[] args) {
14:     var reptile = new Lizard(1);
15:     reptile.layEggs();
16:   } }
```

A. `AALizard`
B. `BALizard`
C. `BLizardA`
D. `ALizard`
E. The code will not compile because of line 3.
F. None of the above.

19. F. The `Reptile` class defines a constructor, but it is not a no-argument constructor. Therefore, the `Lizard` constructor must explicitly call `super()`, passing in an `int` value. For this reason, line 9 does not compile, and option F is the correct answer. If the `Lizard` class were corrected to call the appropriate `super()` constructor, then the program would print `BALizard` at runtime, with the `static` initializer running first, followed by the instance initializer, and finally the method call using the overridden method.

20. E A

```
1:  class Bird {
2:    int feathers = 0;
3:    Bird(int x) { this.feathers = x; }
4:    Bird fly() {
5:      return new Bird(1);
6:    } }
7:  class Parrot extends Bird {
```

20. E. The program compiles and runs without issue, making options A through D incorrect. The `fly()` method is correctly overridden in each subclass since the signature is the same, the access modifier is less restrictive, and the return types are covariant. For covari-

20. E

```
1:  class Bird {
2:     int feathers = 0;
3:     Bird(int x) { this.feathers = x; }
4:     Bird fly() {
5:        return new Bird(1);
6:     } }
7:  class Parrot extends Bird {
8:     protected Parrot(int y) { super(y); }
9:     protected Parrot fly() {
10:       return new Parrot(2);
11:    } }
12: public class Macaw extends Parrot {
13:    public Macaw(int z) { super(3); }
14:    public Macaw fly() {
15:       return new Macaw(3);
16:    }
17:    public static void main(String… sing) {
18:       Bird p = new Macaw(4);
19:       System.out.print(((Parrot)p.fly()).feathers);
20:    } }
```

A. One line contains a compiler error.
B. Two lines contain compiler errors.
C. Three lines contain compiler errors.
D. The code compiles but throws a `ClassCastException` at runtime.
E. The program compiles and prints `3`.
F. The program compiles and prints `0`.

20. E. The program compiles and runs without issue, making options A through D incorrect. The `fly()` method is correctly overridden in each subclass since the signature is the same, the access modifier is less restrictive, and the return types are covariant. For covariance, `Macaw` is a subtype of `Parrot`, which is a subtype of `Bird`, so overridden return types are valid. Likewise, the constructors are all implemented properly, with explicit calls to the parent constructors as needed. Line 19 calls the overridden version of `fly()` defined in the `Macaw` class, as overriding replaces the method regardless of the reference type. This results in `feathers` being assigned a value of `3`. The `Macaw` object is then cast to `Parrot`, which is allowed because `Macaw` inherits `Parrot`. The `feathers` variable is visible since it is defined in the `Bird` class, and line 19 prints `3`, making option E the correct answer.

21. E,F,G

21. Which of the following are properties of immutable classes? (Choose all that apply.)

A. The class can contain setter methods, provided they are marked `final`.
B. The class must not be able to be extended outside the class declaration.
C. The class may not contain any instance variables.
D. The class must be marked `static`.
E. The class may not contain any `static` variables.
F. The class may only contain `private` constructors.
G. The data for mutable instance variables may be read, provided they cannot be modified by the caller.

21. B, G. Immutable objects do not include setter methods, making option A incorrect. An immutable class must be marked `final` or contain only `private` constructors, so no subclass can extend it and make it mutable, making option B correct. Options C and E are incorrect, as immutable classes can contain both instance and `static` variables. Option D is incorrect, as marking a class `static` is not a property of immutable objects. Option F is incorrect. While an immutable class may contain only `private` constructors, this is not a requirement. Finally, option G is correct. It is allowed for the caller to access data in mutable elements of an immutable object, provided they have no ability to modify these elements.

22. E → D

22. What does the following program print?

```
1:  class Person {
2:     static String name;
3:     void setName(String q) { name = q; } }
4:  public class Child extends Person {
5:     static String name;
6:     void setName(String w) { name = w; }
7:     public static void main(String[] p) {
8:        final Child m = new Child();
9:        final Person t = m;
10:       m.name = "Elysia";
11:       t.name = "Sophia";
12:       m.setName("Webby");
13:       t.setName("Olivia");
14:       System.out.println(m.name + " " + t.name);
15:    } }
```

A. `Elysia Sophia`
B. `Webby Olivia`
C. `Olivia Olivia`
D. `Olivia Sophia`
E. The code does not compile.
F. None of the above.

22. D. The code compiles and runs without issue, making option E incorrect. The `Child` class overrides the `setName()` method and hides the `static name` variable defined in the inherited `Person` class. Since variables are only hidden, not overridden, there are two distinct `name` variables accessible, depending on the location and reference type. Line 8 creates a `Child` instance, which is implicitly cast to a `Person` reference type on line 9. Line 10 uses the `Child` reference type, updating `Child.name` to `Elysia`. Line 11 uses the `Person` reference type, updating `Person.name` to `Sophia`. Lines 12 and 13 both call the overridden `setName()` instance method declared on line 6. This sets `Child.name` to `Webby` on line 12 and then to `Olivia` on line 13. The final values of `Child.name` and `Person.name` are `Olivia` and `Sophia`, respectively, making option D the correct answer.

!!!! In instance methods you could use static variables but not reversed.

23. QPZj -> B

24. 182943 -> B

24. What is printed by the following program?

```
1:  class Antelope {
2:     public Antelope(int p) {
3:        System.out.print("4");
4:     }
5:     { System.out.print("2"); }
6:     static { System.out.print("1"); }
7:  }
8:  public class Gazelle extends Antelope {
9:     public Gazelle(int p) {
10:       super(6);
11:       System.out.print("3");
12:    }
13:    public static void main(String hopping[]) {
14:       new Gazelle(0);
15:    }
16:    static { System.out.print("8"); }
17:    { System.out.print("9"); }
18: }
```

A. `182640`
B. `182943`
C. `182493`
D. `421389`
E. The code does not compile.
F. The output cannot be determined until runtime.

24. C. The code compiles and runs without issue, making options E and F incorrect. First, the class is initialized, starting with the superclass `Antelope` and then the subclass `Gazelle`. This involves invoking the `static` variable declarations and `static` initializers. The program first prints `1`, followed by `8`. Then we follow the constructor pathway from the object created on line 14 upward, initializing each class instance using a top-down approach. Within each class, the instance initializers are run, followed by the referenced constructors. The `Antelope` instance is initialized, printing `24`, followed by the `Gazelle` instance, printing `93`. The final output is `182493`, making option C the correct answer.

25. B,C

26. D, G

26. What is the output of the following code?

```
4:  public abstract class Whale {
5:     public abstract void dive();
6:     public static void main(String[] args) {
7:        Whale whale = new Orca();
8:        whale.dive(3);
9:     }
10: }
11: class Orca extends Whale {
12:    static public int MAX = 3;
```

26. What is the output of the following code?

```
4:  public abstract class Whale {
5:     public abstract void dive();
6:     public static void main(String[] args) {
7:        Whale whale = new Orca();
8:        whale.dive(3);
9:     }
10: }
11: class Orca extends Whale {
12:    static public int MAX = 3;
13:    public void dive() {
14:       System.out.println("Orca diving");
15:    }
16:    public void dive(int… depth) {
17:       System.out.println("Orca diving deeper "+MAX)
18: } }
```

A. `Orca diving`
B. `Orca diving deeper 3`
C. The code will not compile because of line 4.
D. The code will not compile because of line 8.
E. The code will not compile because of line 11.
F. The code will not compile because of line 12.
G. The code will not compile because of line 17.
H. None of the above.

14/26