1Z0-808 Exam Topic Reviewer

TopicId: 1021
Topic: Polymorphism and Type Casting

August 5, 2025

Polymorphism: One Object, Many Forms

Polymorphism is the ability of an object to take on many forms. In practice, it means a reference variable of a superclass type can point to an object of any of its subclass types. This is the foundation of flexible and decoupled code, but for the exam, it's a topic that tests your understanding of the difference between reference type and object type.

0.1 Runtime Polymorphism: The Core Concept

This is the most common form of polymorphism, also known as dynamic method dispatch. The key principle to memorize is:

- The **reference type** determines what methods you are *allowed* to call at compile time.
- The **actual object type** determines *which version* of the method will be executed at runtime.

```
class Animal {
    public void makeSound() { System.out.println("Animal sound"); }
}
class Cat extends Animal {
    @Override
    public void makeSound() { System.out.println("Meow"); }
    public void purr() { System.out.println("Purrrr"); }
}
public class Test {
    public static void main(String[] args) {
        Animal myPet = new Cat(); // Reference is Animal, Object is Cat
        myPet.makeSound(); // OK. Prints "Meow".
        // myPet.purr();  // COMPILE ERROR! The Animal reference does not know 'p
    }
}
```

In the example, even though the object is a `Cat`, the compiler only sees an `Animal` reference. Therefore, it only allows calls to methods defined in the `Animal` class. At runtime, the JVM sees the object is actually a `Cat` and executes the overridden `makeSound` method from the `Cat` class.

0.2 Overloading vs. Overriding: A Critical Distinction

The exam loves to confuse these two concepts. Do not fall for it.

Method Overriding (Runtim
Happens in two classes (supercla
Method signature must be identi
Return type can be covariant
Resolved at runtime by the JVM

0.3 Object Type Casting: Changing Perspectives

Casting allows you to convert a reference from one type to another. It's how we solve the problem in the previous example where we couldn't call the `purr()` method.

- **Upcasting:** Casting from a subclass to a superclass. It's always safe and done implicitly. `Cat myCat = new Cat(); Animal myAnimal = myCat; // Implicit upcast`
- **Downcasting:** Casting from a superclass back to a subclass. It's risky and must be done explicitly. You are telling the compiler, "I know better than you; this object is really a subclass."

If you lie to the compiler and the object isn't what you claim it is, you get a `ClassCastException` at runtime.

```
Animal myPet = new Cat();
Cat myCat = (Cat) myPet; // Explicit downcast. This is SAFE.
myCat.purr(); // Now this is valid.

Animal anotherPet = new Animal();
// The object is an Animal, not a Cat.
Cat anotherCat = (Cat) anotherPet; // Throws ClassCastException at RUNTIME.
```

0.4 The `instanceof` Operator: Your Safety Net

To avoid the dreaded `ClassCastException`, use the `instanceof` operator to check the actual type of an object before you cast it.

```
public void doAnimalStuff(Animal animal) {
    animal.makeSound();
    if (animal instanceof Cat) {
        Cat cat = (Cat) animal; // This is now guaranteed to be safe.
        cat.purr();
    }
}
```

Exam Traps for `instanceof`:

- `null instanceof AnyType` always evaluates to `false`.
- The compiler is smart. If it can determine that a cast is impossible, it will throw a compile-time error. This happens when the two types are unrelated classes.

  ```
  String s = "Hello";
  // A String can never be an Integer.
  if (s instanceof Integer) { ... } // COMPILE ERROR: Inconvertible types
  ```

Key Takeaways for the 1Z0-808 Exam

- **Reference vs. Object:** Know what the compiler allows (based on reference type) versus what the JVM executes (based on object type). This is the essence of polymorphism.
- **Overriding vs. Overloading:** Do not mix them up. Know the key differences from the comparison table.
- **Casting Rules:** Upcasting is implicit and safe. Downcasting is explicit and risky.
- **Safe Casting:** Always use `instanceof` before downcasting to prevent a `ClassCastException`. Be aware of compile-time cast checks.