Quiz yourself: Handling overloaded Java methods with arguments and return values

JAVA SE

Quiz yourself: Handling overloaded Java methods with arguments and return values

Do you know how the compiler selects which method to invoke when it's forced to choose due to overloading?

by Mikalai Zaikin and Simon Roberts

March 15, 2021

If you have worked on our quiz questions in the past, you know none of them is easy. They model the difficult questions from certification examinations. We write questions for the certification exams, and we intend that the same rules apply: Take words at their face value and trust that the questions are not intended to deceive you but to straightforwardly test your knowledge of the ins and outs of the language.

The objective is to create methods with arguments and return values, including overloaded methods. Given the following classes:

```
class GenericEngine { public String engType="

class CombustionEngine extends GenericEngine
    public String engType="CE-002"; }

class JetEngine extends CombustionEngine {
    public String engType="JE-003"; }

public class Car {
    public void setEngine(Object o) {
        System.out.print("I have unknown engi
    }

    public void setEngine(GenericEngine ge) {
        System.out.printf(
            "I have generic engine: %s", ge.e
    }
```

```
    public void setEngine(CombustionEngine ce
        System.out.printf(
            "I have combustion engine: %s", c
    }
}
```

And this code fragment:

```
JetEngine e = new JetEngine();
new Car().setEngine(e);
```

What is the result? Choose one.

A. `I have unknown engine` — The answer is A.

B. `I have generic engine: GE-001` — The answer is B.

C. `I have combustion engine: CE-002` — The answer is C.

D. `I have generic engine: CE-002` — The answer is D.

E. `I have combustion engine: JE-003` — The answer is E.

Answer. This question investigates how methods are selected for invocation and also how variables are resolved. The relevant rules for overloaded methods and for field access are documented in *Java Language Specification* sections 15.12.2, "Compile-time step 2: Determine method signature" and 15.11.1, "Field access using a primary."

The code fragment above constructs a `JetEngine` and initializes a variable of that same type to refer to the object. It then calls a `setEngine` method, using the variable as the argument.

There are three overloaded methods called `setEngine`, and although none of them takes an argument of *exactly* the type `JetEngine`, the argument of each overload is a parent of `JetEngine`. Therefore, any of them *could* accept the argument. But one method must be selected, and the compiler performs that selection. The specification describes how the method is selected from the candidate overloads in as many as three stages.

The first stage (which happens to be where the decision is made in this example) is to try to identify the target method based on the provided argument types, without using any autoboxing/unboxing or variable argument list-handling rules.

If the first stage were to fail, a second stage would look for a target method by applying autoboxing/unboxing, and finally, the third stage would look for a match by applying variable argument list-handling rules.

So, the first stage selects the most specific method based on the types of the parameters. *Java Language Specification* section 15.12.2.5, "Choosing the most specific method," says the following:

> If more than one member method is both accessible and applicable to a method invocation, it is necessary to choose one to provide the descriptor for the run-time method dispatch. The Java programming language uses the rule that the most *specific method* is chosen.

There's a fairly long and detailed definition in the specification of what's considered "most specific," but in this context, the phrase simply means "nearest to the actual argument type."

Given a `JetEngine` as an actual parameter, `CombustionEngine` is the most specific formal parameter type, and `Object` is the least specific. Therefore (and given that no `setEngine(JetEngine e)` method is defined), the compiler will generate code to invoke the `setEngine(CombustionEngine ce)` method.

In light of this discussion, you know that the output will start with the message `I have combustion engine`. Consequently, options A, B, and D are incorrect.

Next, consider which engine type message is printed. Each of the three engine variants has the same `engType` variable, and each subclass hides the parent's class variable. This represents highly dubious coding style, and this question illustrates fairly convincingly why it's considered bad.

It might seem as if the three variables called `engType` are overrides in the same sense that occurs in the definitions of methods with the same name in a hierarchy of classes; however, this is not the case. Why? A single instance of the `JetEngine` object actually contains three independent variables called `engType`, each with different values and each visible from different scopes. *Java Language Specification* section 15.11.1 referenced earlier, about field access for using a primary, states the following:

> Note that only the type of the *[p]rimary* expression, not the class of the actual object referred to at run time, is used in determining which field to use.

The primary expression is the part that comes before the last dot. In the following `setEngine` method, the primary expression is `ce`, and its type is `CombustionEngine`:

```
public void setEngine(CombustionEngine ce) {
    System.out.printf("I have combustion engi
}
```

Consequently, the `engType` variable that is printed is the one embedded in the `CombustionEngine` part of the object, and that has the value `CE-002`. Because of this, you can see that option C is correct and option E is incorrect.

This behavior might be surprising, but the central point is that the late-binding effect applies only to the invocation of a nonprivate, nonfinal instance method on an object, not to direct field access. The behavior can perhaps be improved in several ways:

You could render this behavior less surprising if you simply avoided using the same variable name.

You could change this behavior to print a more expected message if you avoided making direct reference to the variable `engType` and instead invoked a method `getType()` and ensured that this method is overridden in all three classes. However, this is a cumbersome solution, duplicating an identical `getType` method in every class.

Another possibility would be to have a single `engType` variable defined in the base `GenericEngine` class. This variable is configured appropriately via a chain of constructors in the three classes. The following example implements this approach by using a `final` field.

```
class GenericEngine {
    public final String engType;
    protected GenericEngine(String engType) {
        this.engType = engType; }
    public GenericEngine() { this("GE-001");
}

class CombustionEngine extends GenericEngine
    protected CombustionEngine(String engType
    public CombustionEngine() { this("CE-002"
}

class JetEngine extends CombustionEngine {
    protected JetEngine(String engType) { sup
    public JetEngine() { this("JE-003"); }
}
```

However, the best solution might be simply to avoid using class inheritance in this situation entirely.

If the protected constructor of the `GenericEngine` class in the last code block were made public, that would allow all three engine types to be handled directly by that class. Of course, there might be other constraints on a more complete design, but a common mantra in modern software engineering is to prefer delegation over inheritance. Using delegation for code reuse is an approach you should understand, but it's more complex than can be discussed in the context of this question.

Conclusion: The correct option is option C.

Mikalai Zaikin

Mikalai Zaikin is a lead Java developer at IBA IT Park in Minsk, Belarus. During his career, he has helped Oracle with development of Java certification exams, and he has been a technical reviewer of several Java certification books, including three editions of the famous *Sun Certified Programmer for Java* study guides by Kathy Sierra and Bert Bates.

Simon Roberts

Simon Roberts joined Sun Microsystems in time to teach Sun's first Java classes in the UK. He created the Sun Certified Java Programmer and Sun Certified Java Developer exams. He wrote several Java certification guides and is currently a freelance educator who publishes recorded and live video training through Pearson InformIT (available direct and through the O'Reilly Safari Books Online service). He remains involved with Oracle's Java certification projects.

Share this Page