Unit 15: Method Invocation

After this unit, the student should:

- understand the two step process that Java uses to determine which method implementation will be executed when a method is invoked
- understand that Class Methods do not support dynamic binding

How does Dynamic Binding work?

We have seen that, with the power of dynamic binding and polymorphism, we can write succinct, future-proof code. Recall that example below, where the magic happens in Line 4. The method invocation `curr.equals(obj)` will call the corresponding implementation of the `equals` method depending on the run-time type of `curr`.

```
1  // version 0.1 (with polymorphism)
2  boolean contains(Object[] array, Object obj) {
3    for (Object curr : array) {
4      if (curr.equals(obj)) {
5        return true;
6      }
7    }
8    return false;
9  }
```

How does dynamic binding work? To be more precise, when the method `equals` is invoked on the target `curr`, how does Java decide which method implementation is this invocation bound to? While we have alluded to the fact that the run-time type of the target `curr` plays a role, this is not the entire story. Recall that we may have multiple versions of `equals` due to overloading. So, Java also needs to decide, among the overloaded `equals`, which version of `equals` this particular invocation is bound to.

This unit elaborates on Java's decision process to resolve which method implemented in which class should be executed when a method is invoked. This process is a two-step process. The first occurs during compilation; the second during run time.

During Compile Time

During compilation, Java determines the method descriptor of the method invoked, using the compile-time type of the target.

For example, in the line

```
curr.equals(obj)
```

above, the target `curr` has the compile-time type `Object`.

Let's generalize the compile-time type of the target to C. To determine the method descriptor, the compiler searches for all methods that can be correctly invoked on the given argument.

In the example above, we look at the class `Object`, and there is only one method called `equals`. The method can be correctly invoked with one argument of type `Object`.

What if there are multiple methods that can correctly accept the argument? In this case, we choose the *most specific* one. Intuitively, a method M is more specific than method N if the arguments to M can be passed to N without compilation error. For example, let's say a class `Circle` implements:

```
      boolean equals(Circle c) { .. }

      @Override
      boolean equals(Object c) { .. }
```

Then, `equals(Circle)` is more specific than `equals(Object)`. Every `Circle` is an `Object`, but not every `Object` is a `Circle`.

Once the method is determined, the method's descriptor (return type and signature) is stored in the generated code.

In the example above, the method descriptor `boolean equals(Object)` will be stored in the generated binaries. Note that it does not include information about the class that implements this method. The class to take this method implementation from will be determined in Step 2 during run-time.

During Run Time

During execution, when a method is invoked, the method descriptor from Step 1 is first retrieved. Then, the run-time type of the target is determined. Let the run-time type of the target be R. Java then looks for an accessible method with the matching descriptor in R. If no such method is found, the search will continue up the class hierarchy, first to the parent class of R, then to the grand-parent class of R, and so on, until we reach the root

`Object`. The first method implementation with a matching method descriptor found will be the one executed.

For example, let's consider again the invocation in the highlighted line below again:

```
1  // version 0.1 (with polymorphism)
2  boolean contains(Object[] array, Object obj) {
3    for (Object curr : array) {
4      if (curr.equals(obj)) {
5        return true;
6      }
7    }
8    return false;
9  }
```

Let's say that `curr` points to a `Circle` object during run-time. Suppose that the `Circle` class does not override the method `equals` in `Object`. As a result, Java can't find a matching method descriptor `boolean equals(Object)` in the method `Circle`. It then looks for the method in the parent of `Circle`, which is the class `Object`. It finds the method `Object::equals(Object)` with a matching descriptor. Thus, the method `Object::equals(Object)` is executed.

Now, suppose that `Circle` overrides the method `Object::equals(Object)` with its own `Circle::equals(Object)` method. Since Java starts searching from the class `Circle`, it finds the method `Circle::equals(Object)` that matches the descriptor. In this case, `curr.target(obj)` will invoke the method `Circle::equals(Object)` instead.

Invocation of Class Methods

The description above applies to instance methods. Class methods, on the other hand, do not support dynamic binding. The method to invoke is resolved statically during compile time. The same process in Step 1 is taken, but the corresponding method implementation in class C will always be executed during run-time, without considering the run-time type of the target.