Static vs. Dynamic Type

By Chris Jeng, op@berkeley.edu; edits by Dennis Zhao, denniszhao@berkeley.edu

The General Approach

Suppose we have a generic initialization and method call:

```
StaticClass myInstance = new DynamicClass();
myInstance.method1(p); // p is of type Parameter
```

If you're thinking wtf are these classes, great! They aren't actual Java classes, I just made them up here to try to make it clear what they are. So imagine that they do exist. The ~~object~~ *variable* (thanks to Rohin) `myInstance` has static type `StaticClass`, and dynamic type `DynamicClass`, *always*. Recall that the dynamic class of any object must always be either the same as or a subclass of its static class.

So how do we know if the second line will compile? How do we know if it will run? I'm going to try to provide a generalized method of following Java as it dives into the compiling part, the running part (and… that's it, Java just does those two things). Also, although I just said I'll make a "generalized" method, I'll try to make it as a concrete as possible, referring back to `StaticClass` and `DynamicClass` and `Parameter`, because we both know those are just arbitrary but fixed classes.

The Short Version

1) Do a static lookup on `myInstance`. If it passes, proceed. If not, we have a compile error.
2) Run `method1` as defined in `DynamicClass`.

(This is generally true, but in the case of the difficult Batman problems (see examples down below), we'll have to understand the long version to know why it outputs what it does)

The Long Complicated Version

0) Is the computer powered on? If not, it won't compile, and therefore, it won't run (running only occurs if compilation succeeds).
1) First, we need to follow the compilation. For the method call, this is called *static lookup*. That means we check `StaticClass` to see if it has a method that takes a single input of type `Parameter` (what is `Parameter`? `Parameter` is the static type of `p`.). In other words, the compiler looks inside the definition of the class `StaticClass` and sees if there is a method with the same method header (i.e. it takes in a `Parameter`, is called `method1`, and has the same return type). There are two possibilities: it found it, or it didn't find it.

   a. FOUND IT – If the compiler found such a method, that's great! Then the compiler knows that the code is OK, and gives it a little certificate saying "compiler-certified", and so we proceed to step 2).
   b. DIDN'T FIND IT – If the compiler couldn't find a method called method1 that takes in a single input of type `Parameter`, then the compiler is *almost* angry. It's almost angry because it does one last thing before it says, "NO THIS DOESN'T COMPILE". It'll check to see if `Parameter` is a subclass of anything. Suppose for a moment, that `Parameter` was actually a subclass of `ParameterParent`. Then if there is a method called `method1` that takes in a `ParameterParent` as input instead, the compiler won't be angry. If it finds such a `method1`, the compiler will remember this super-cast* and give the certification, and proceed to step 2. Otherwise, the compiler will continue looking for methods that take in a superclass of `Parameter`, or a superclass of `ParameterParent`, etc. and if it can't find anything, then it'll freak out and say compile error.

2) Lastly, we follow the runtime. We do something called *dynamic lookup*. That means we run the method according to its definition inside `DynamicClass`. But what if there are multiple definitions (we call this overloading) of `method1`? Which one do we run? Almost always, we run the `method1` that takes in an input of type `Parameter`. However, if we ever hit the special case* where the compiler had to check for a superclass, then we would instead run the method that takes in an input of type `ParameterParent`. The takeaway idea is, **Java runs whatever was compiled, if it even compiled.**

*This is a bit of a weird case, to see more about this, see Example 2 below.

The below is just a few more run-throughs of understanding what such a pair of lines call. These examples were taken from the Sunday Guerrilla Section.

Examples

Example 1

```
Superhero s = new Batman();
s.punch();
```

In line 1, we created `s`.

`s` has a static type `Superhero` and a dynamic type `Batman`. What happens when we call `s.punch()`? This can be broken down into two steps: compile-time and run-time.

Compile-Time

For this code to compile, it will need to pass the compilation check. For line 1, all that is required to pass the compilation check is that the `Batman` class extends the `Superhero` class. This is true, so we go on to line 2.

For line 2, the compile-time needs to check the static type of s and see if it has a [visible] method `punch()`. Notice that if the `punch()` method were not visible in our context of calling it, this would be an error. For example, if the `punch()` method were declared `private`, then if we were to call these two lines from some other package, obviously we can't call `s.punch()`. So suppose that whoever wrote these two lines did a good job, so that the method is properly visible.

If `Superhero` has a `punch()` method, then all is good, and the compilation check passes.

If `Superhero` doesn't have a `punch()` method, then all is *not* good, and the compilation fails. But what if `Batman` has a definition of `punch()`? Nope, doesn't matter. The compiler is interested in the static type only.

(Notice that both the above two possibilities didn't depend on whether or not `Batman` had its own definitions of `punch()`! In other words, during compile time, the compiler gives no ****'s about the dynamic type, other than the inheritance check in line 1)

Run-Time

Remember, we will only get to the run-time if it passes the compile-time (why would any normal computer try something impossible when it already knows it's impossible?). Suppose that we do get to the run-time. Then what happens?

Definition of *dynamic lookup* – Dynamic lookup just means that when we call the method of an object, we always end up executing the method that's defined in the dynamic type of the object.

So that means that `s.punch()` will end up calling the `punch()` method defined in `Batman`, if it exists. If no such method is defined in `Batman`, then Java will know to run the inherited version of the method, which is just the method defined in `Superhero`.

Example 2

Suppose we had these two lines:

```
Superhero s = new Batman();
s.punch((Batman) s);
```

Let's take this in its two steps:

Compile-Time

Line 1 works because `Batman` is a subclass of `Superhero` (same as Example 1). Line 2 is the tricky one:

The compiler checks the static type of s to see if it has a method that takes an input of type `Batman`. Why of type `Batman`? Because the `(Batman)` cast temporarily changes the static type of `s` inside line 2. So when the compiler goes to `Superhero.java` to check to see if a `punch(Batman b)` method exists, it can't find it! The compiler is suspicious, if the following step doesn't work, the compiler would error. The last check the compiler makes is to see if there is a method with the header `punch(Superhero s)`, because it will continue checking for inputs of a type that is the parent of the most specific. It would find that there does indeed exist a `punch(Superhero s)`, so the compiler *REMEMBERS THIS* and says ok, compilation check passed. To sum this, the compiler was looking for a `punch(Batman b)`, but because it couldn't find it, it settled for the next less-specific input: `punch(Superhero s)`.

Run-Time

This is the tricky part. For the dynamic lookup, we would normally expect that the method that is run is the `punch(Batman b)` defined in the `Batman` class. However, what is actually run is the `punch(Superhero s)` method defined in `Batman`. Why isn't it the `Batman` one?! I thought dynamic lookup means we run the method in the dynamic class?! Yes, you are almost 100% correct, the more correct way to put it is:

During runtime, Java will run the method in the dynamic class as successfully found in the static lookup part of compile-time.

Notice that this is really tricky because we're passing in a parameter of type `Batman`, but the punch method with parameter of type `Superhero` is actually being run. You can think of this as, when you have an instance that has a static type and a relatively-more-specific dynamic type, like in this example, the static class can effectively "bottleneck" the methods that the instance can actually run.

Taking the thought further:

Question: Is there any way to actually run the `punch(Batman b)` method in `Batman` if we swapped out line2 for something else?

Yes, it's possible. All we need to do is somehow make the static lookup go directly to `Batman`. How do we do that? We just cast line2 so that everything is treated like a `Batman`. In other words, change…

```
s.punch((Batman) s);
```

to this…

```
((Batman) s).punch((Batman) s);
```

This would work if we had visibility of the method (notice that the method is declared private). What can access a private method? Only something inside the class. So in order to compile, run, and print this line for us to see, we would have to put line1 and modified-line2 in a `public static void main(String[] args)` method body **inside the same file** as the definition of `Batman` (probably `Batman.java`).

*Extra note: dynamic lookup only applies to non-private and non-static methods. *