Discussion 4 Notes: Inheritance & Dynamic Method Selection

Anton A. Zabreyko

Spring 2023

1 Inheritance

One of Java's most fundamental features is inheritance, or the ability for classes to "inherit" from other classes. Consider the following example.

```
public class Animal {
    private int age;
    private String name;
    private String hobby;

    public class Animal(int age, String name, String hobby) {
        this.age = age;
        this.name = name;
        this.hobby = hobby;
    }

    public void greet() {
        System.out.println("I'm not really supposed to talk.");
    }

    public void eat() {
        System.out.println("Dinner time!");
    }

}
public class Cat extends Animal {

    public Cat(int age, String name, String hobby) {
        super(age, name, hobby);
    }

    public void meow() {
        System.out.println("Meow!");
    }

    @Overrides
    public void greet() {
        System.out.println("I am a cat.");
    }
}
```

We say that Animal is a **superclass** of Cat, and that Cat is a **subclass** of Animal. This is established by Cat's usage of the **extends** keyword, which indicates that Cat is inheriting from Animal. What does this all actually mean? In essence, it means that Cat gets to have Animal's methods for free! If I were to create a Cat instance *g*, I could write *g.eat*() and I would receive "Dinner time!" in standard out. Note that a class can only extend **one superclass**.

Notice that Cat **overrides** one of Animal's methods. This means that when we call *g.greet*(), it will print "I am a cat." rather than "I'm not really supposed to talk.". There is much more to inheritance than this simple example showcases, but for now we only cover one other element.

1.1 Interfaces & Implements

How might we say that objects have certain traits or characteristics? For example, if we had a Cat class and a Dog class, it might be convenient to have a shared interface between them for calling a cuddle method. Java does possess such a feature, and it is in fact called an **interface**! Consider the following example.

```
inteface Cuddly {
    public void cuddle();
    public void snuggle();
}

public class Cat implements Cuddly {

    public void cuddle() {
        System.out.println("I don't wanna");
    }

    public void snuggle() {
        System.out.println("*scratches you*");
    }
}

public class Dog implements Cuddly {

    public void cuddle() {
        System.out.println("Yaaaaaay!");
    }

    public void snuggle() {
        System.out.println("Yaaaaaay!");
    }

}
```

Perhaps the Cat should not be described as Cuddly. Nonetheless, let us now describe this example. First, observe that Cuddly lists two methods, but does not define them. The idea is that Cat and Dog, which implement the interface (as seen by their usage of the **implements** keyword), provide the actual implementations of the interface's methods. Take note that the interface, and all of its methods, are public. This is a requirement. It is necessary because an interface is intended to be an API, and so having them be anything but public renders that pointless. We will see next week a very important example of interfaces, but for now focus on understanding these basics. Now, we turn to one of the trickiest subjects in Java.

2 Dynamic Method Selection

From the structure of inheritance and polymorphism arises a natural question: when you possess a Java object, and you evoke a particular method signature from it, what method will it actually run? The complexity of this question is apparent in the following example.

```
public class A {

    public void f(int a) {
        System.out.println("A's method!");
    }
}

public class B extends A {

    public void f(int a) {
        System.out.println("B's method!");
    }

}

public static void main(String[] args) {
    A objectA = new A();
    B objectB = new B();
    A mystery = new B();

    objectA.f();
    objectB.f();
    mystery.f();

}
```

Clearly, objectA.f() will be A's method, and objectB.f() will be B's method, but what of mystery's call? To understand this, we must learn Java's rules for dynamic method selection.

2.1 Static and Dynamic Type

First, let us cover some important vocabulary we need to understand. Consider the following variable declaration:

```
Object a = new Integer(0);
```

Here, we have declared a variable with name a of **static type** Object and **dynamic type** Integer. What is the functionality of these different types? The static type is in a sense the "official" type of the object. It is the only thing that the Java compiler considers when checking variable assignments and method invocations. The dynamic type is what the object actually is. So, for this object a, the compiler considers it as an Object, but in reality when the program is run it is an Integer.

2.2 Compile Time

The general procedure is broken down into two phases. First, the compilation phase.

1. Ensure that each variable assignment is valid.
2. Ensure that all method calls resolve to some method of a matching method signature.
3. If no suitable method is found, throw a compiler error.

2.2.1 Variable Assignments

Let us elaborate on the first point. Suppose we have a typical inheritance structure of a superclass Animal, and subclasses Cat and Dog. Consider the following assignments:

```
Animal a = new Animal(); // valid
Animal b = new Cat(); // valid
Cat c = new Cat(); // valid
Cat d = new Animal(); // invalid
Dog e = new Cat(); // invalid
```

It is of course permissible to create an object whose static and dynamic types match, so examples a and c are acceptable. Likewise, we can declare a variable's static type to be of a general class, and then assign its dynamic type to be something more specific. Recall that an object is, abstractly speaking, an API of methods and variables. Thus, any subclasses of an object by inheritance will meet the requirements of that API, and hence can be assigned that type statically. Thus, example b also is acceptable.

It is precisely for this same reason that examples d and e fail. A Cat object is more specific than an Animal, and could have extensions to the API that an Animal does not have. Thus, we cannot assign an Animal to be a Cat. Similarly, Cat and Dog are completely separate from each other, and could have differing extensions to the API. Thus, we cannot assign across the inheritance tree either.

2.2.2 Method Signature Lookup

Now, onto the second and third steps. Recall that a method signature is composed of the function's name and parameters. At compile time, the compiler uses the provided method signature to check whether or not there is a valid function that matches the signature. Consider the following example:

```
public class Animal {
    public void eat() {
        System.out.println("I am eating!");
    }

}

public class Cat extends Animal {
    public void meow() {
        System.out.println("Meow!");
    }
}

public static void main(String[] args) {
    Animal cat = new Cat();
    cat.eat(); // I am eating!
    cat.meow(); // compile error
}
```

The static type of cat is Animal, so the compiler references Animal when checking method signatures. Thus, we are able to find the method cat.eat(), but not cat.meow() because there is no meow method in Animal. So, despite this object in reality being a Cat, we are unable to invoke one of its method. Luckily, Java has a built-in way to get around this. Recall the casting feature. Casting allows us to tell the compiler that a certain object is of another type, and the compiler will simply trust us on it as long as it could reasonably be of that other type. Consider the updated example:

```
public static void main(String[] args) {
    Animal cat = new Cat();
    cat.eat(); // I am eating!
    ((Cat) cat).meow(); // Meow!
}
```

This will work, because we have told the compiler that the cat object is a Cat, and so it passes the compiler check. Note that it does check at runtime whether or not cat is of type Cat.

2.3 Runtime

So, the code has passed compile time. What happens in runtime, when the code is actually executed? Again, we have a three step sequence:

1. If the method identified at compile time is static, simply execute it. Do not check for overriden methods.
2. Check for any overridden non-static methods.
3. Check that all casts were correct.

2.3.1 Static Methods

Recall that when the static keyword is applied to a method, it is independent of any particular instance of that object. In a sense, it belongs to the whole class. It resides in its own special section of memory, and thus as a result, if the compile phase identities a static method during its lookup, it will lock onto that method and execute it during runtime, regardless of any overriden methods in the dynamic type.

2.3.2 Overriden Methods

If the method identified at compile time is non-static, then in runtime there will be a lookup based on the dynamic type for any overriding methods that match the function signature. For example, consider yet again Animal and Cat. Suppose Animal has a method play that takes in an Animal as parameter, and that Cat has a method with the exact same name and input. Then, if we have an object that is statically an Animal and dynamically a Cat, it will invoke Cat's play method rather than Animal's.

2.4 Canonical Example

Let us now illustrate all we have learned with a simple example that covers all cases. Consider the following code.

```
public class A {
    public void f() { System.out.println("A.f"); }
    public static void g() { System.out.println("A.g"); }
}

public class B extends A{
    public void f() { System.out.println("B.f"); }
    public static void g() { System.out.println("B.g"); }
    public static void h() { System.out.println("B.h"); }
}

public static void main(String[] args) {
    A a = new A();
    A fakeA = new B();
    B b = new B();
}
```

As an exercise, fill out the following table. The solution is on the next page.

	.f()	.g()	.h()
a			
fakeA			
b			

	.f()	.g()	.h()
a	A.f	A.g	C
fakeA	B.f	A.g	C
b	B.f	B.g	B.h

Explanation:

a.f(): The static type is A, and the dynamic type is A, so there is no confusion over this case. We will simply use a's f.

a.g(): The static type is A, so at compile time we see that A has a static g method, and lock into it. Thus, at runtime we have A.g().

a.h(): A has no h method, so this results in a compile time error.

fakeA.f(): At compile time, we see that A has a nonstatic method f. At runtime, we see that fakeA is a B, and thus use B's overriden f method.

fakeA.g(): At compile time, we see that A has a static method g, and so we lock into it. Thus, at runtime, despite the fact that fakeA is dynamically a B, we will run A.g.

fakeA.h(): Even though fakeA is dynamically a B, at compile time it is only known to be an A, and so thus the compiler will not be able to find an h method for A.

b.f(): Statically and dynamically a B, we will just run B's f method.

b.g(): Statically a B, we find B's g method, and lock onto it to run in runtime.

b.h(): Statically a B, we find an appropriate h method to run, and then at runtime we run it.