13 Topics in Subclassing

Having a static type system with subclassing brings a whole range of issues to the fore.

Recall that during program execution (i.e., at run time), an instance of a class `C` is just a structure that carries a value for all the fields specified in `C` along with a function for every method in `C`. The key point is that instances (or objects) carry their own method with them. If you try to invoke method `m` on object `obj` using `obj.m()`, execution proceeds by looking up in the structure representing `obj` a method named `m`. If the object doesn't have such a method, we get a "method not found" error. If it is there, then the method body is evaluated to yield a result. The object, at run time, also carries with it the class as an instance of which it was created, what we will call the *dynamic type* of the object. (Here, dynamic means that it is the type that the object has during execution.)

We contrast this dynamic type with the *static type* of an object, which is the class that an object has according to the annotation we have in our source code. For instance, if we declare `Foo x = ...`, then variable `x` and any value it holds has static type `Foo`. (Here, static means that it is the type that object has given the annotation the appear in the source code, i.e., before we even start executing the program.)

Recall our definition of subclassing: C is a subclass of D if an instance of C can be used in any context where an instance of D is expected, without causing a "method not found" run-time error. One of the roles of the type system is to check enough about your program to ensure that there will be no "method not found" error during execution. It does this by checking that: at any point in the execution of a program, the *dynamic type* of a value is a subclass of its *static type*. The type system can compute the static type of every expression in the program, because those types do not require executing the program, and the type system is set up to reject programs where it may be possible to have an expression return a value of a dynamic type that is not a subclass of the expression's static type.

Let's look at how this works, and along the way illuminate the difference between subclassing and polymorphism. Consider the following running example, perhaps the basis for computer-controlled antagonists in a virtual reality game. We have a class `BadGuy` with a creator `create(name)` and two operations `name()` and `doSomethingEvil()`. I'm keeping it simple for illustration purposes.

```
public class BadGuy {
  // magic Java invocation to allow BadGuy to be a superclass
  protected BadGuy () {};
```

```
  private String thename;

  private BadGuy (String n) { thename = n; }

  public static BadGuy create (String n) { return new BadGuy(n); }

  public String name () { return this.thename; }

  public void doSomethingEvil () {
    System.out.println ("Arh! Arh! I'm killing, maiming, pillaging!");
  }
}
```

```
public class Megalomaniac extends BadGuy {
  // magic Java invocation to allow this class to be a superclass
  protected Megalomaniac () {};

  private String thename;

  private Megalomaniac (String n) { thename = n; }

  public static Megalomaniac create (String n) {
    return new Megalomaniac(n);
  }

  public String name () { return this.thename; }

  public void doSomethingEvil () {
    System.out.println ("Arh! Arh! I'm taking over the world!");
  }

  public void hatchWorldDominationPlan () {
    System.out.println ("Thinking.... Thinking....");
  }
}
```

`Megalomaniac` is a subclass of `BadGuy`, because we said so. Of course, Java checks that it actually is a subclass, by checking that every public method of `BadGuy` is also a public method of `Megalomanic`. Note also that the implementation of a method need not be the same in `BadGuy` as it is in `Megalomaniac`.

Suppose we have a function that takes a `BadGuy` and has that `BadGuy` do something evil:

```
public static void announceAndDoEvil (BadGuy g) {
  System.out.println (g.name() + " is plotting something evil - he says:")
  ;
  g.doSomethingEvil();
}
```

Because `BadGuy` has a `name()` and a `doSomethingEvil()` method, then this works fine. Because `Megalomaniac` is a subclass of `BadGuy`, we expect that we can pass a `Megalomaniac` instance to `announceAndDoEvil()` and not get an error. And indeed, we can — because `Megalomaniac` has both a `name()` and a `doSomethingEvil()` method, there will be no "method not found" error.

All right, the above is all review, that's the basics of subclassing. But subclassing has an unexpected consequence: we may lose static type information.

13.1 Information Loss and Downcasting

In what sense can we lose information? Consider the following function, that does nothing interesting, yet already exhibits interesting consequences:

```
public static BadGuy identityBG (BadGuy g) {
  return g;
}
```

Operationally, the function simply returns its argument. Nothing much going on there. But note that we have to give the function a type. The best type we can give it right now is the type I gave it above. It takes a `BadGuy`, and returns a `BadGuy`. Read it as a contract: it guarantees if you give it a `BadGuy`, it produces a `BadGuy` back to you. In fact, because of subclassing, you know that you can give it a `BadGuy` or any subclass of `BadGuy`. Now, it gives you back, by the same argument, a `BadGuy` or any subclass of a `BadGuy`. But what are you *guaranteed* about the result? Well, the only guarantee you can make is that you're guaranteed to get back a `BadGuy`.

Here's the only you can do with `identityBG` and `Megalomaniac`s.

```
Megalomaniac adolf = Megalomaniac.create("Adolf");
BadGuy adolf1 = identityBG(adolf);
```

This is where we lose static type information. The static type of `adolf1`, the result of calling `identityBG`, is `BadGuy`, because that's what `identityBG` is declared to return. That is so

even if the static type of its argument was `Megalomaniac`. Indeed, during execution, the dynamic type of `adolf` is of course `Megalomaniac` (since that's how it was created), and after calling `identityBG`, the dyamic type of `adolf1` is still `Megalomaniac`, because `identityBG` doesn't change the value it gets as an argument. This is a case where the static type starts diverging from the dynamic type of a value.

And that's the best we can do, statically. If you try to write:

```
Megalomaniac adolf = Megalomaniac.create("Adolf");
Megalomanic adolf2 = identityBG(adolf);
```

you get a type error, because `identityBG` is not guaranteed to return a `Megalomaniac`—look at its type. Of course, we could write a different identity function:

```
public static Megalomaniac identityMM(Megalomaniac g) {
  return g;
}
```

But then we have two identities going around, all sharing the same code, and we don't get to reuse code. That's what subtyping was intended to do. And clearly, we cannot define an identity `identityBM` as:

```
public static Megalomaniac identityBM (BadGuy g) {
  return g;
}
```

because that's just false—if we give `identityBM` a `BadGuy`, we should get a `BadGuy` back. So the above is essentially the best we can do, and as we saw, it loses static type information.

This is where polymorphism comes into play. It is a way to regain code reuse, without losing static type information. Identity can be given the polymorphic type:

```
public static <B extends BadGuy> B identity (B g) {
  return g;
}
```

which says that if you give it a value of static type B which is a subclass of `BadGuy`, then the result will have static type B, for that same exact B. So if you give it something of static type `Megalomaniac`, it gives you back something with static type `Megalomaniac`. If you give it something with static type `BadGuy`, it gives you back something with static type `BadGuy`. No static type information lost, and a single function definition.

Just to be clear about this, it is not the case that any method with a contract that takes a `BadGuy` and returns a `BadGuy` can be given polymorphic type Consider the following function, which is definitely not an identity:

```
public static BadGuy notIdentity (BadGuy g) {
  return BadGuy.create(g.name());
}
```

This function takes a value with static type `BadGuy` or a subclass of it, and gives you back a value with static type `BadGuy`. And that's the best we can do, because at run time, this function takes a value with dynamic type `BadGuy` or a subclass of it, and will always gives you back something of dynamic type `BadGuy`: the result will have no method `hatchWorldDominationPlan()`:

```
Megalomaniac drevil = Megalomaniac.create("Doctor Evil");
BadGuy drevil1 = notIdentity(drevil);
```

So in the `identityBG` case, calling `identityBG` with a `Megalomaniac` loses information, while not so with `notIndentity`.

So in some cases, we lose static type information. Sometimes you can use polymorphism to avoid this information loss (as in `identity` above). But sometimes you can't. Is there a way to regain the lost static information?

To regain static type information, we can try to *cast* the object, that is try to view the object as an object of some other class. In the above example with `identityBG`, when we pass in a value with static type `Megalomaniac` and get back a value with static type `BadGuy` (having lost type information), we can try to cast the result to `Megalomaniac`.

A cast is an expression

```
(C) e
```

where `C` is a class name, and `e` is an expression that yields an object. (We sometimes call a cast a *downcast* when casting the expression to a class that is a subclass of the static type of `e`.) As far as the type system is concerned, the result of a cast is a value with static type `C`. It's an assertion you're making, and the type system takes you at face value.[1] A cast executes as follows: first evaluate `e` down to an object, then check if the dynamic type of the result is a subclass of `C`—if not, the cast throws a `ClassCastException`. The idea is that we can adjust the static type of an expression (presumably to recover static type information that was lost), but the trade-off is that there is some checking that needs to occur during execution, as opposed to doing all the checking during type-checking, before the code executes.

Let's look at some examples. We know that `identityBG` returns the same object it is passed as an argument, so that if you give it a value with dynamic type T it gives you back a value

[1] There is some sanity checking going on. In Java, the casting operator checks that the class `C` you are trying to cast expression `e` to is either a subclass or a superclass of the static type of `e`.

with the same dynamic type T. Sso it should be the case that if we give it a value with static type `Megalomaniac`, it should hand us back a value of static type `BadGuy` that we can downcast back to a `Megalomaniac`:

```
Megalomaniac adolf = Megalomaniac.create("Adolf");
Megalomaniac adolf3 = (Megalomaniac) identityBG(adolf);
adolf3.hatchWorldDominationPlan();
```

And everything executes very nicely. The type system is happy, because `adolf3` has static type `Megalomaniac` after the cast, and therefore the call `adolf3.hatchWorldDominationPlan()` does not cause a type error, and at run time, because the dynamic type of the result of `identityBG(adolf)` is a `Megalomaniac`, the cast succeeds, and execution proceed by invoking `hatchWorldDominationPlan()` on `adolf3`.

Of course, we can only downcast an expression to a `Megalomaniac` if the dynamic type of the expression is a subclass of `Megalomaniac`. The following fails, since `identityBG` returns a value with dynamic type `BadGuy` when given a value with dynamic type `BadGuy`, and a value with dynamic type `BadGuy` cannot be downcast to a `Megalomaniac`—there is no `hatchWorldDominationPlan()` method in the structure representing that value:

```
BadGuy mrpink = BadGuy.create("Mr. Pink");
Megalomaniac mrpink1 = (Megalomaniac) identityBG(mrpink);
mrpink1.hatchWorldDominationPlan();
```

The type checker is happy, because it takes your cast at face value, and accepts your assertion that `mrpink1` has static type `Megalomaniac`. Thus, `mrpink1.hatchWorldDominationPlan()` does not cause a type error. However, at run time, the code throws a `ClassCastException` in the second line, because the dynamic type of `identityBG(mrpink)` is `BadGuy`, which is not a subclass of `Megalomaniac`, and the cast fails.

Similarly, `notIdentity` returns a value with dynamic type `BadGuy` even when given a value with dynamic type `Megalomaniac`, so the following also throws a `ClassCastException` in the second line:

```
Megalomaniac drevil = Megalomaniac.create("Doctor Evil");
Megalomaniac drevil2 = (Megalomaniac) notIdentity(drevil);
drevil2.hatchWorldDominationPlan();
```

That downcasts can throw exception when they fail means that you should wrap downcasts with a try/catch combination to ensure that your whole program does not abort. (Alternatively, you can just check whether the downcast will succeed by checking that the `obj` you want to downcast to class `C` can be viewed as an instance of `C`, by evaluating `obj instanceof C`, which is true if the dynamic type of `obj` is a subclass of `C`.)

13.2 Dynamic Dispatch

Subclassing and type checking ensures that when you try to invoke a method `m` on object `obj` during execution, object `obj` actually has a method `m` to invoke at run time. But *which* method `m` does the program actually invoke?

The answer is rather blindingly obvious, but it has some interesting consequences later on. The choice of Java, and most object-oriented languages, is one that cemented the usefulness of object-oriented languages in the first place. Java uses a method invocation technique called *dynamic dispatching*, which simply says that when you execute `obj.m()`, the method `m()` that gets called is the method that is stored in the structure representing object `obj` at execution. Thus, the method invoked (equivalently, *dispatched*) is the one that is defined in the dynamic type of the object, hence the moniker *dynamic dispatch*.

You all have internalized this behavior by now, I suppose. But it's important to state it clearly, because in the presence of subclassing, it is very easy to lose track of the dynamic type of values (since as we saw above the annotation in the program only tells us what the static type of a value will be, which in general will be a superclass of the dynamic type of a value)

Recall the function `announceAndDoEvil()` from earlier.

```
public static void announceAndDoEvil (BadGuy g) {
  System.out.println (g.name() + " is plotting something evil - he says:")
  ;
  g.doSomethingEvil();
}
```

Now, when you pass in a value with dynamic type `BadGuy` to `announceAndDoEvil()`, for instance,

```
announceAndDoEvil(BadGuy.create("Jaws"))
```

then what gets printed is:

```
Jaws is plotting something evil - he says:
Arh! Arh! I'm killing, maiming, pillaging!
```

But of course, if you pass in a value with dynamic type `Megalomaniac` to `announceAndDoEvil()`, for instance,

```
announceAndDoEvil(Megalomaniac.create("Robespierre"))
```

you get:

```
Robespierre is plotting something evil - he says:
Arh! Arh! I'm taking over the world!
```

So what gets executed depends on the dynamic type of the object being passed in to `announceAndDoEvil()`.

That's very powerful, very useful. It is easy to adjust the behavior of objects by simply giving different definitions for a given method. This is one of the hallmarks of object-oriented programming. But it's also a software engineering nightmare. Why?

Well, consider the definition of the function `announceAndDoEvil()`. It talks about `BadGuy`, and talks about `System.out.println`. So you figure that only have to understand what `BadGuy` does, and what `System.out.println` does, to understand what `announceAndDoEvil()` does. But of course, that's completely wrong. As we saw above, `announceAndDoEvil()` can do anything, because it can be passed a *subclass* of `BadGuy`, and anyone implementing a subclass of `BadGuy` can make `doSomethingEvil()` in that class do anything. So in order to understand `announceAndDoEvil()`, you have to be aware of all the possible subclasses of `BadGuy` in an application. You cannot reason about the code in isolation—you have to have full knowledge of the application that uses that function. That's even worse when you are developing a library and offer something like `announceAndDoEvil()`, because then you cannot guarantee anything to users about what the function can do.[2]

[2]One way around this is to restrict the extent to which `BadGuy` can be subclassed. Languages have ways to do that.