Topics in Subclassing

Having subclassing in a language, along with types, 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.

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" runtime error. One of the roles of the type system is to check enough about your progra to ensure that there will be no "method not found" error during execution.

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, of course, that the implementation of a method need not be the same in `BadGuy` as it is in `Megalomaniac`.

Suppose we have a function (in some class, somewhere) that expects a `BadGuy` and does something.

```
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 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 (type) information, and an unexpected question gets raised: what method actually gets called during execution.

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 identity (BadGuy g) {
  return g;
}
```

Operationally, the function simply returns its input. 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 `identity` and `Megalomaniac`s.

```
Megalomaniac adolf = Megalomaniac.create("Adolf");
BadGuy result = identity(adolf);
```

This is where we've lost information. Java, which can only use the type of `identity` to make its static checks, has to go with the guaranteed result of the function, which is that the result is a `BadGuy`. If you try to write:

```
Megalomaniac adolf = Megalomaniac.create("Adolf");
Megalomanic result = identity(adolf);
```

you get a type error, because `identity` is not guaranteed to return a `Megalomaniac`—looks at its type. Of course, we could write `identity` to give it the type

```
public static Megalomaniac identity(Megalomaniac);
```

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 give `identity` the type:

```
public static Megalomaniac identity (BadGuy);
```

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

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` will automatically take a Megalomaniac and return a Megalomaniac. Consider the following function, which is definitely not an identity:

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

If you give it a `BadGuy`, it creates a new `BadGuy` that looks the same, but if you give it a `Megalomaniac`, you get back a `BadGuy`, not a `Megalomaniac`—there is no method `hatchWorldDominationPlan()` in the result. Again, this is the best you can do:

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

So in the `identity` case, calling `identity` with a `Megalomaniac` loses information, while not so with `notIndentity`. How can you regain the information you have lost, when you have lost it? Note that the types cannot help you determine if you have lost information, by the way. It's only your knowledge of what the code does that can guide you here.

To regain type information, we can try to *downcast* the object, that try to view the object as an object of some other class—in the above case with `identity`, when we pass in a `Megalomaniac` and get back a `BadGuy` (having lost type information), we can try to downcast the result to `Megalomaniac`.

A downcast (which we already encountered when talking about the `equals` method back in Lecture 4) is an expression

```
(C) e
```

where `C` is a class name, and `e` is an expression that yields an object. A downcast executes as follows: first evaluate `e` down to an object, then try to see if you can view the result as an object of class `C`—if so, then it gives back the object at that type, otherwise, it throws a `ClassCastException`.

Let's look at some examples. We know that `identity` returns the same object it is passed as an argument, so it should be the case that if we give it a `Megalomaniac`, it should hand us back something that we can downcast back to a `Megalomaniac`:

```
Megalomaniac adolf = Megalomaniac.create("Adolf");
BadGuy result = identity(adolf);
((Megalomaniac) result).hatchWorldDominationPlan();
```

or equivalently:

```
Megalomaniac adolf = Megalomaniac.create("Adolf");
Megalomaniac result = (Megalomaniac) identity(adolf)
result.hatchWorldDominationPlan();
```

And everyting executes very nicely.

But of course, we can only downcast to a `Megalomaniac` if what we have is truly a `Megalomaniac` instance. The following fails, since `identity` returns a `BadGuy` when given a `BadGuy`, and a `BadGuy` cannot be downcast to a `Megalomaniac`—there is no `hatchWorldDominationPlan()` method in the structure representing the object:

```
BadGuy mrpink = BadGuy.create("Mr. Pink");
Megalomaniac result = (Megalomaniac) identity(mrpink);
result.hatchWorldDominationPlan();
```

The above code throws a `ClassCastException` in the second line.

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

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

Again, the result from `notIdentity` having no method `hatchWorldDominationPlan()` is the reason for the failure.

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 `(C) obj` would succeed, and false if `(C) obj` would fail.

Part of the problem above, the reason why we have information loss, is that the contract for `identity` is not quite the right one—`identity` guarantees something stronger than what the contract captures. **Exercise:** Can you think about what you would like the *actual* contract for `identity` to say?

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. But *which* method `m` does the *invocation* actually execute?

The answer is rather blindingly obvious, but it has some interesting consequences later on. The choice of which method to invoke is called *dynamic dispatching*, and simply says that when you call `obj.m()`, the method `m()` that gets called is the actual one that is in the structure representing object `obj` at execution. Thus, the method invoked (hence *dispatch*) is the one that is in the object at run-time (hence *dynamic*)

This seems silly to state like this. But then it's easy to forget what is actually there at runtime! In particular, in the presence of subclassing, trying to predict behavior is near impossible!

Recall the function `announceAndDoEvil()` earlier on.

```
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 `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 `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 actual object being passed in to `announceAndDoEvil()`.

That's very powerful, very useful. One of the hallmarks of object-oriented programming. And 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.[1]

Things will get even worse with dynamic dispatch once we toss in inheritance.

Midterm Review

Here are the main topics we have seen in class. I expect you to be able to answer basic questions about each, including definitions, relationship between various notions. I also expect you to be able to recognize any of the notions below, and to use them in code fragments that I may ask you to write.

- ADT and specification: signatures, algebraic specifications, reasoning with algebraic specifications, implementing specifications, public versus private members of a class, static methods for creators, dynamic methods for operations (understand the difference), code against an ADT to allow substituting implementations, defining equality for ADTs, implicit specifications for equality.
- Errors and testing: classification of errors, exceptions, use of the type system to guarantee some classes of errors do not occur, exceptions, black-box versus white-box testing, unit testing versus integration testing, tests generation, testing via a specification.
- Subclassing: definition of subclass, abstract classes, recipe to derive an implementation from an ADT specification, nested classes, tree subclassing hierarchies, non-tree subclassing hierarchies, interfaces, generic interfaces, downcasting, dynamic dispatch
- Functional iterators: definition, implementation.

[1] One way around this is to restrict the extent to which `BadGuy` can be subclassed. Languages have ways to do that. We'll see them later.