On Enforcing Static Properties

Last time, we saw the main property that the Java type system enforces, namely if a program type checks, then at no point during the execution of the program does the system attempt to invoke a method `meth` on an object that does not provide method `meth`.

Unfortunately, Java does not actually get this quite right. There is a hole in the type system. The problem is the subtyping rule for arrays. Recall that subtyping for arrays says that whenever `S` is a subtype of `T`, then `S[]` is a subtype of `T[]`. Now, this means that the following code type checks, and in fact works perfectly fine.

```
public class Test1 {

  public static void main (String[] argv) {
    Integer[] a = {1,2,3,4,5};
    show (a);
  }

  public static void show (Object[] a) {
    for (Object i : a) {
      System.out.println (i.toString());
    }
  }
}
```

In particular, it is fine to pass the array of integers to `show`, because `Integer` is a subclass of `Object`, and therefore `Integer` is a subtype of `Object`.

The problem is that the following also type checks, for exactly the same reason:

```
public class Test2 {

  public static void main (String[] argv) {
    Integer[] a = {1,2,3,4,5};
    update (a);
    System.out.println (a[0].intValue());
  }
```

```
    public static void update (Object[] a) {
      a[0] = new Object();
    }
  }
```

This code is very similar to the code above, except that the method to which we pass the array of integers modifies the first element of the array, making it hold a new `Object`. You should make sure that you understand why the code above type checks. In particular, because `Integer` is a subtype of `Object`, it is perfectly fine to pass an array of `Integer`s to a method expecting an array of `Object`s. And because the array `a` is an array of `Object`, it is perfectly fine to change the first element in the array into a different `Object`.

The problem is that passing an object (including an array) to a method in Java only passes a reference to that object. The object is not actually copied. So when method `update` updates the array through its argument `a`, it ends up modifying the underlying array `a` in method `main`. But that means that when we come back from the `update` method, array `a` is an array of integers where the first element of the array is not an integer but rather an object. And when we attempt to invoke method `intValue` on that first element, Java will choke because that first element, being a plain `Object`, does not in fact implement the `intValue` method! That contradicts the guarantee I made above.

Java will report an exception at this point. (Actually, I am lying. Java will report an exception slightly earlier, namely when you attempt the update in method `update`—it will catch the fact that you are attempting to modify an array by putting in an object that is not a subclass of the class at which the array was *originally* defined, and throw an `ArrayStoreException`. (Here, because we are trying to put an `Object` into an array originally created to hold `Integer`s.)

The point remains: the type system does not fully do its job, and has to delegate to the runtime system that responsability of catching this particular problem.

Well, that's ugly. Well, you may ask: could we have done the check in the type system, instead of throwing a runtime exception? (Recall that lecture we had about why it was a good idea to report errors early, in particular, when the program is being compiled, as opposed to when the code executes? That's exactly what's happening here.) It turns out, yes, we can check for this problem during type checking, but the Java developers did not get that right at the time when Java was developed, and because of backwards compatibility cannot really correct the problem. The solution, of course, is to not allow the subtyping rule that says that when `S` is a subtype of `T`, then `S[]` is a subtype of `T[]`.

Interestingly enough, that's the solution they took when the time came to add generics to the language. Consider the first example above, except using `LinkedList`s instead of arrays.

```
  import java.util.*;
  public class Test3 {
```

```
  public static void main (String[] argv) {
    LinkedList<Integer> lst = new LinkedList<Integer>();
    lst.add(1);
    lst.add(2);
    lst.add(3);
    show (lst);    // does not type check
  }

  public static void show (LinkedList<Object> a) {
    for (Object i : a) {
      System.out.println (i.toString());
    }
  }
}
```

The above code fails to type check. It is *not the case* that if `S` is a subtype of `T`, then `LinkedList<S>` is a subtype of `LinkedList<T>`. This seems counterintuitive, but it prevents us from writing the code such as in `Test2`, that updates an aggregate structure and forces us to do a runtime check and possibly throw an exception. Bottom line: we cannot write code such as that in `Test2` using generics.

Unfortunately, we've thrown the baby out with the bathwater. Code such as that in `Test3` is actually quite useful, and works fine. It's only when we update an aggregate structure that problems may occur. Generics offer a way out of it, reinstating some amount of subtyping. But we have to be explicit about where we want the subtyping. Consider the type for `show` in `Test3`. Suppose we wanted to be explicit about the kind of subtyping we allowed here. Roughly, we would like it to say that `show` accepts any linked list with some type of element `T` that is a subclass of `Object`. We don't care and don't know what that type of element `T` is, so we'll write it down as a question mark. We therefore get the code:

```
import java.util.*;
public class Test4 {

  public static void main (String[] argv) {
    LinkedList<Integer> lst = new LinkedList<Integer>();
    lst.add(1);
    lst.add(2);
    lst.add(3);
    show (lst);
  }

  public static void show (LinkedList<? extends Object> a) {
    for (Object i : a) {
```

```
        System.out.println (i.toString());
      }
    }
  }
```

And this type checks perfectly fine. The subtyping rule for generics is as follows: if `S` is a subtype of `T`, then `LinkedList<S>` is a subtype of `LinkedList<? extends T>`. Wrap your head around this rule, and above example.

This lets us reinstate subtyping for generics. Have we added too much? Can we write the update example? We can check that the following code does not type check:

```
public class Test5 {

  public static void main (String[] argv) {
    LinkedList<Integer> lst = new LinkedList<Integer>();
    lst.add(1);
    lst.add(2);
    lst.add(3);
    update (lst);
    System.out.println (lst[0].intValue());
  }

  public static void update (LinkedList<? extends Object> a) {
    a.set(0,new Object());   // does not type check
  }
}
```

The reason for the type checking failure here is a bit subtle. Note that the type of `a`, as far as Java is concerned is `LinkedList<T>` for some unknown `T`. (That's what the `?` says.) Now, method `set` in `LinkedList<E>` has signature:

```
public E set (int index, E elenent)
```

So in order for the invocation of `set` to type check, it must be the case that `new Object()` be an expression returning a value of type `T`, where `T` is an unknown type. Java cannot establish that `new Object()` has type `T`, because, and that's the key, `T` is unknown!

Leaving aside the details, the main consequence of this is that the `<? extends T>` notation permits the use of subtyping in some instances, and disallows it in the cases where it could cause an exception.

The upshot: there are two ways to deal with subtyping for aggregate structures, and Java in fact uses them both—one rule for arrays, and a different rule for generics. That's confusing, and forces you to remember how to deal with situations differently in those two cases. But that's what we have to live with.

Design Patterns for Static Properties

Not every property we care about will be enforced by the type system. However, there are ways to get Java to enforce a property we care about by a disciplined use of the primitives that the language provides. Some of these uses can be captured by what we call design patterns. We will have much more to say about design patterns in the coming weeks, but let's just focus on design pattern to enforce static properties, for the time being.

We already saw a kind of design pattern for enforcing a property, namely the *Immutability* design pattern, that enforces that a class is immutable. The design pattern calls for writing a class such that:

(1) fields are private

(2) methods in the class do not update the fields

The first point ensures that a client of the class cannot mutate a field by directly accessing it. The second point ensures that the code of the class itself cannot mutate a field by. Put together, these two properties are sufficient to make the class immutable.

Another pattern that enforces a static property is the *Singleton* design pattern. It enforces that a class has only a single instance. This can be useful when the instance is meant to abstract over some resource that cannot be shared, such as the screen, for example.

There are a couple of ways to implement the Singleton design pattern. The easiest way, and a way that does not rely on runtime exceptions to achieve its goal, is to simply make private the constructor of the class, create a private instance of the class, and define a static method (typically called `getInstance`) that returns that single instance. Because the constructor is private, no one else can create a new object of the class. Here is the "skeleton" of a singleton class:

```
public class SomeClass {

  private static SomeClass uniqueInstance = new SomeClass ();

  private SomeClass ()  { ... }

  public static SomeClass getInstance () {
    return uniqueInstance;
  }

  ...
}
```

If we want the constructor of the class to take arguments, and those arguments are not known until the first time `getInstance` is invoked, then we need to rely on a slightly different approach, such as:

```
public class SomeClass {

  private static SomeClass uniqueInstance = null;

  private SomeClass (...)  { ... }

  public static SomeClass getInstance (...) {
    if (uniqueInstance==null)
      uniqueInstance = new SomeClass(...);
    return uniqueInstance;
  }

  ...
}
```

This is not quite as satisfactory, because whenever `getInstance` is called with different arguments, it will still return the original instance, silently ignoring the arguments. This can be confusing to a programmer.