A Note on Type-Checking Java

Last lecture, we saw different kind of errors, and I indicated that some of those errors can be detected at compile-time, that is, before executing the program, and some can only be detected at run-time.

Java, like several programming languages, tries to catch several kind of errors at compile-time, especially so-called type errors. The Java type system is an algorithm (defined by a set of rules) that takes the source code of a program, and without actually running the code, returns either

- a guarantee that when the code will execute it will not cause either a type error or a method-does-not-exist error, or
- points out that there is a possible type error or method-does-not-exist error, and indicate where the error may happen.

Note my use of qualification, such as "*possible* type error", and "where the error *may* happen". This is because the algorithm is good, but not that good. To get a sense for why this is, we need to have a sense of how the type system is working. Knowledge of the type system is also important because it provides certain guarantees, and making sure that these guarantees are preserved were a driving force behind the adoption of some of the features of the language, and the form that those features take.

Before we start, remember that Java is a language with both expressions (that is, fragments of code that return a value, such as `1+1`) and statements (that is, fragments of code that do not return a value, that simply are executed for their side effects.) A method's body is made up of statements that are executed in order. A method call can be either a statement (if the method has return type `void`) or an expression (if the method has a return type that is not `void`).

A program in Java is a set of classes. Each class is type checked in the context of all the other classes in the program. A class is type checked by checking that every method in the class type checks. To type check a method, the type-checking algorithm relies on two functions: $typeOf(E)$ that returns the type of expression E, and $check(S)$ that cheks that statement S is type correct.

The function $typeOf(E)$ is fairly easy to define, depending on what E is:

- The type of a literal integer such as `1`, `2` is just `int`, the type of a Boolean literal such as `true` or `false` is `boolean`.
- The type of a variable is just the type at which that variable was declared.
- The type of a primitive operation such as + depends on the actual operation: for +, for instance, if both arguments have type `int` (obtained by using *typeOf* on the arguments), then the expression has type `int`.
- The type of a method call obj`.`m`(`E_1`,...,`E_n`)` is more complicated. First, we get the type of obj by calling *typeOf* — the result better be a class type C. Then we check if class C declares a method m. If not, then we report a type error ("class does not declare method m"). If yes, then we look up the result type of method m in C as well as the type of the arguments, we check that expressions E_1, ... E_n have type corresponding to the arguments (if not, we report a type error "arguments to method m have the wrong type"), and we finally return that expression obj`.`m`(`E_1`,...,`E_n`)` has type whatever the result type of m is.

Similarly for other kind of expressions.

Once we have *typeOf*(E), defining *check*(S) is easy. That function does not return a type, it simply checks if expression S has a type error in it. If not, then nothing happens. Again, it depends on the form of the statement.

- for an assignment statement of the form x `=` E where E is an expression, we first get the *typeOf*(E), and check that it is the same as the declared type of variable x. (If not, we report a type error.)
- for a method call $obj.m(E_1, \ldots, E_n)$, we do something similar to the expression case. First, we get the type of obj by calling *typeOf* — the result better be a class type C. Then we check if class C declares a method m. If not, then we report a type error ("class does not declare method m"). If yes, then we look up the declared type of the arguments and check that expressions E_1, ... E_n have type corresponding to those declared – if not, we report a type error "arguments to method m have the wrong type".
- for a conditional `if (`b`)` S_1 `else` S_2, we first check that *typeOf*(b) is `boolean` — if not, we report a type error ("condition of a conditional is not a Boolean"). Otherwise, we check S_1 and S_2: *check*(S_1) and *check*(S_2).

Similarly for other kind of statements.

The algorithm is conservative—it deems that some programs may have a type error even though no error occurs during execution. For instance, it is easy to see, by looking at how `check` works, that the code snippet:

```
if (x==x)
  return 10 + 10;
else
  return 10 + true;
```

would fail to type check: the algorithm reports a type error, namely that the "else" branch of the conditional does not type check, since expression `10+true` does not type check. But there will be no error during execution because the "else" branch cannot be taken — the test `x==x` is always true. The reason that the type checker reports a type error is that it does not take advantage of possible information it may have about whether there is any guarantee that only one of the branches will ever be taken. It always checks that both branches type check, even if one will never be taken when the code executes. This is a choice that the designers of the algorithm made: such choices depend on complexity of the resulting type checking algorithm, uniform behavior that is more easily predicted by the user, etc.

It is possible to prove that the Java type system basically guarantees that if there are no type error reported during compilation of a set *CLS* of classes, then executing any method in any class in *CLS* with arguments of the right type does not cause a primitive operation to be applied to values of the wrong type, or an attempt to invoke a method that is not defined.

The above is the starting point for our examination of what can be done to help programming software, focusing on how we can use the type system to help us with encapsulation, modularity, code reuse, and design patterns. We next examine a first example.

Code Reuse: Subclassing

Consider a simple Point ADT:

```
public static Point create (int, int);
public int xPos ();
public int yPos ();
public Point move (int, int);

create(x,y).xPos() = x
create(x,y).yPos() = y
create(x,y).move(dx,dy) = create(x+dx,y+dy)
```

Suppose that we extended the Point ADT into a Colored Point ADT, with the following interface:

```
public static CPoint create (int, int, Color);
```

```
public int xPos ();
public int yPos ();
public CPoint move (int, int);
public int getColor ();
public CPoint newColor (Color);

create(x,y,c).xPos() = x
create(x,y,c).yPos() = y
create(x,y,c).move(dx,dy) = create(x+dx,y+dy,c)
create(x,y,c).getColor() = c
create(x,y,c).newColor(d) = create(x,y,d)
```

I claim that this is an extension of the Point ADT simply because every colored point is really a point: everything you can do to a point, you can in fact do to a colored point. This is reflected by the algebraic specification of the two classes, which in some sense agree on the operations that the ADT have in common. **Thought exercise:** can you make formal this relationship? More precisely, how can you get, from the specification of Point and CPoint, that every colored point in fact behaves just like a point? For now, let us rely on our intuition that colored points are points.

What does this mean? It means, among other things, that if we implement the CPoint ADT as a class, we should really make it so that it is a *subclass* of the Point class. This means that an object of class of CPoint can be passed to any method or stored in any variable expecting an object of class Point. The main requirement is that every public method of Point should also be a public method of CPoint. Of course, the implementation of the method can be quite different, but the method should exist, and have the same types. (Similarly, every public field of Point should be a public field of CPoint. But because we will never use public fields in this course —immutability!—we only have to worry about methods.)

Definition: *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.*

In Java, subclassing is not automatic, in the sense that it needs to be pointed out to the compiler.[1] Subclassing is expressed in Java using an `extends` annotation on the class. Let `Point` be an immutable implementation of the Point ADT. For instance:

```
public class Point {
  private int xpos;
  private int ypos;

  // this is needed - Java specific incantation
  // we'll examine this later.
```

[1]This is called *nominal subtyping*. When we don't need to point out that C is a subclass of D and the language can figure it out for itself, the language uses what is called *structural subtyping*.

```
  // think of this as: Point can now have subclasses
  protected Point () { }

  private Point (int x, int y) {
    xpos = x;
    ypos = y;
  }

  public static Point create (int x, int y) {
    return new Point(x,y,c);
  }

  public int xPos () { return this.xpos; }

  public int yPos () { return this.ypos; }

  public Point move (int dx, int dy) {
    return create(this.xpos + dx, this.ypos + dy);
  }
}
```

Assume that we have a class `Color` available, the details of which are completely irrelevant. Here is a possible definition of the CPoint class:

```
public class CPoint extends Point {
  private int xpos;
  private int ypos;
  private Color color;

  private CPoint (int x, int y, Color c) {
    xpos = x;
    ypos = y;
    color = c;
  }

  public static CPoint create (int x, int y, Color c) {
    return new CPoint (x,y,c);
  }

  public int xPos () { return this.xpos; }

  public int yPos () { return this.ypos; }
```

```
  public CPoint move (int dx, int dy) {
    return create(this.xpos + dx, this.ypos + dy, this.color);
  }

  public Color getColor () { return this.color; }

  public CPoint newColor (Color c) {
    return create(this.xpos, this.ypos, c);
  }
}
```

This works perfectly fine, and Java will work with this quite happily.

This seems like awfully redundant code. And it is. And some of you are screaming, what about inheritance? We'll see inheritance next week. Inheritance is something different. **Inheritance is not subclassing.** The two are somewhat related, of course, but it is possible to have subclassing without inheritance. The above code uses subclassing, and does not rely on inheritance. It is also possible to have inheritance without subclassing, but that's less common. Inheritance brings its own bag of problems with it, that subclassing by itself does not have.

Subclassing enables code reuse. How? **Subclassing lets you reuse client code**. For instance, suppose that you have a method that takes two points p, q and computes their distance, such as:

```
public static double distance (Point p, Point q) {
   return Math.sqrt(Math.pow(p.xPos()-q.xPos(),2.0) +
                    Math.pow(p.yPos()-q.yPos(),2.0));
}
```

Now, this *same piece of code* can be used to work on colored points, because the class `CPoint` is a subclass of `Point`, and because every colored point is guaranteed to have methods `xPos` and `yPos`.