Return to Module Overview Page (https://onlinelearning.berkeley.edu/courses/665040/pages/module-three)

3.7: Type Safety Revisited

In the previous module we talked about type safety. To you it might have sounded more like a pain than anything else. Why is type safety such a big issue? One important reason is that in a type-safe language, you can get the compiler to help you find bugs even before your code runs the first time.

Consider a program that asks the classic question: "Can you put a square peg in a round hole?" Somewhere in this program, you might encounter a function declared like this:

```
int insertPeg(int p, int h) { ... }
```

What does this function do? Well, it looks like p should be the index of a peg, perhaps in some global peg array, and h must be a hole index. What is the return value? It could be an error code, or perhaps the index of the previous peg from the given hole, or . . . we'll have to read the manual.

Now imagine we're maintaining this program. We see the following code:

```
int i,j;
for (i=FIRST_HOLE, j=FIRST_PEG;
    i<N_HOLES && j<N_PEGS;
    i++, j++)
    insertPeg(i,j);
```

We know the program compiles fine, and runs fine. Does this mean that the code above is correct? No! Can you see why?

If FIRST_HOLE and N_HOLES equal FIRST_PEG and N_PEGS, this program will run perfectly, but it's really doing the wrong thing. We've switched the order of the arguments, yet the compiler is perfectly happy to accept what we've written. Furthermore, if we change the final variables N_HOLES, N_PEGS, etc., and the program starts failing at runtime, we might not even notice, because the programmer is not checking the return value for an error status (if indeed the return value is an error status). How can we get the compiler to find this mistake at compile time, when the programmer is sitting at the computer, ready to fix the problem, rather than deferring the error until runtime, when the hapless user is trying to test her IQ?

Custom data types to the rescue!

What if the insertPeg() function were defined like this:

```
class Peg { int pegID; }
class Hole { int holeID; }
class Error { int errorCode; }
// the following code has to be inside
//some other class...
Error insertPeg(Peg p, Hole h) { ... }
```

Now it is impossible to call the function incorrectly!

```
Peg[] pegs = new Peg[10];
Hole[] holes = new Hole[10];
// ... initialize peg and hole arrays somehow

int i,j;
for (i=FIRST_HOLE, j=FIRST_PEG;
     i<N_HOLES && j<N_PEGS;

     i++, j++)
     insertPeg(holes[i],pegs[i]);
```

Compiler says:

```
test.java:18: Incompatible type for method. Can't
              convert Hole to Peg.
        insertPeg(holes[i],pegs[j]);
                      ^
```

This notion (using type safety as a means to force the compiler to find errors it might otherwise miss) is very important, and is central to the concept of object-oriented programming, as we will see in the next module (after we've learned a bit more about classes and objects).