Unit 20: Run-Time Class Mismatch

After taking this unit, students should:

- Understand the need for narrowing type conversion and type casting when writing code that depends on higher-level abstraction
- Understand the possibility of encountering run-time errors if typecasting is not done properly.

We have seen in Unit 18 how we can write code that is reusable and general by making our code dependent on types at a higher-level of abstraction. Our main example is the following `findLargest` method, which takes in an array of objects that support the `getArea` method, and returns the largest area among these objects.

```
1  // version 0.3
2  double findLargest(GetAreable[] array) {
3    double maxArea = 0;
4    for (GetAreable curr : array) {
5      double area = curr.getArea();
6      if (area > maxArea) {
7        maxArea = area;
8      }
9    }
10   return maxArea;
11 }
```

The method served our purpose well, but it is NOT a very well-designed method. Just returning the value of the largest area is not as useful as returning the *object* with the largest area. Once the caller has a reference of the object, the caller can call `getArea` to find the value of the largest area.

Let's write our `findLargest` method to find which object has the largest area instead.

```
1  // version 0.4
2  GetAreable findLargest(GetAreable[] array) {
3    double maxArea = 0;
4    GetAreable maxObj = null;
5    for (GetAreable curr : array) {
6      double area = curr.getArea();
7      if (area > maxArea) {
8        maxArea = area;
9        maxObj = curr;
10     }
11   }
     return maxObj;
```

```
12   }
13
```

Let's see how `findLargest` can be used:

```
1  GetAreable[] circles = new GetAreable[] {
2    new Circle(new Point(1, 1), 2),
3    new Circle(new Point(0, 0), 5)
4  };
5
6  GetAreable ga = findLargest(circles);  // ok
7  Circle c1 = findLargest(circles); // error
8  Circle c2 = (Circle) findLargest(circles); // ok
```

The return type of `findLargest` (version 0.4) is now `GetAreable`. On Line 6 above, we assign the return object with a compile-time type of `GetAreable` to `ga`, which also has `GetAreable` as its compile-time type. Since the variable `ga` is of type `GetAreable`, however, it is not very useful. Recall that `GetAreable` is an interface with only one method `getArea`. We cannot use it as a circle.

On Line 7, we try to return the return object to a variable with compile-time type `Circle`. This line, however, causes a compile-time error. Since `Circle` <: `GetAreable`, this is a narrowing type conversion and thus is not allowed (See Unit 14). We will have to make an explicit cast of the result to `Circle` (on Line 8). Only with casting, our code can compile and we get a reference with a compile-time type of `Circle`.

Cast Carefully

Typecasting, as we did in Line 8 above, is basically is a way for programmers to ask the compiler to trust that the object returned by `findLargest` has a run-time type of `Circle` (or its subtype).

In the snippet above, we can be sure (even *prove*) that the returned object from `findLargest` must have a run-time type of `Circle` since the input variable `circles` contains only `Circle` objects.

The need to cast our returned object, however, leads to fragile code. Since the correctness of Line 8 depends on the run-time type, the compiler cannot help us. It is then up to the programmers to not make mistakes.

Consider the following two snippets, which will compile perfectly, but will lead to the program crashing at run-time.

```
1  GetAreable[] circles = new GetAreable[] {
2    new Circle(new Point(1, 1), 2),
```

```
    new Square(new Point(1, 1), 5)
};

Circle c2 = (Circle) findLargest(circles);
```

Or

```
GetAreable[] circles = new GetAreable[] {
    new Circle(new Point(1, 1), 2),
    new Circle(new Point(1, 1), 5)
};

Square sq = (Square) findLargest(circles);
```

We will see how to resolve this problem in later units.