Chapter 4: Specifying and Throwing Exceptions

Specifying and Throwing Exceptions

Methods in Java can tell the compiler that they may throw an exception if they contain code that can possibly throw one. In fact, if they can throw a checked exception, they must tell the compiler. They do that by using the *throws clause* in the method declaration.

Here's what that looks like. Let's say I wanted to do some file processing as in the example at the end of Chapter 3, but I didn't want to have the try-catch logic in my method. I could pass any of the above exceptions to the next level like this, assuming my method has the name processFile():

```
public void processFile() throws FileNotFoundException,
                                 InputMismatchException,
                                 IOException {
    // code to process file(s)
}
```

That throws clause tells the compiler (and anyone else who wants to use the method) that I know my code may throw those exceptions, but I want the program to handle those at another level. So it won't check my method for try-catch logic to process them. And any programmer who wants to use the method can see that she or he will need to either handle the exception or pass it along to the next level in the *call stack*.

Let me explain a bit about the call stack. It's the system's list of active methods. So if method A calls method B, and then B calls C, there are three methods in the call stack.

If an exception occurs in C, and C doesn't handle it, Java moves on to B to see if there's a handler there. If not, it moves on to A. If there's no handler there, then the program will blow up. So Java searches every level for an exception handler before giving up and blowing up the program. Take a look at the following flow chart to see what I mean:

Text equivalent start.

Method C throws exception -> Method B does not catch it -> Method A’s exception handler catches the exception.

Text equivalent stop.

When method C throws its exception, Java moves to the next level in the stack and checks method B for an exception handler. Since B doesn't have one, Java continues to the next level and checks method A for a handler. In this case, it finds one, and Java handles the exception there. Because of the exception, methods B and C don't return normally, so the exception handler will have to manage that detail too.

Let's discuss about a real example. The Scanner class, which we've used quite a bit, throws several checked exceptions. Some of its constructors that use file names, for example, may throw a FileNotFoundException, and some of its methods may throw an IOException. When we use those methods, should we handle the exception, or does it make more sense to pass it on to the next level?

Deciding whether to handle an exception or pass it on to another method in the call stack is a design decision that you should base on the type of class you're designing and how people will use it. Since Scanner's constructors depend on their callers for file names, if a file isn't there, it makes sense to pass the error back to the caller through the throws clause.

Now let's look at specifics in the Temperature class. As I mentioned, it's possible for a user of my Temperature class to be inconsiderate and call my constructor with a type or temperature that the user hasn't verified. What should I do about that?

I can't just ignore it, and since this lesson is about exceptions, it's logical to assume that I might want to throw an exception. Now Java has lots of exceptions already defined, but where would the fun be if we just used one of those? Instead we're going to create new exception classes for our errors.

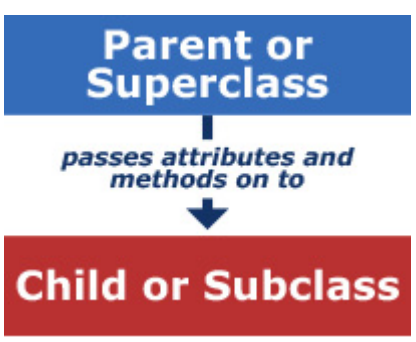
The first error we'll consider is an invalid type, so we're going to set up a new exception class with a name that indicates exactly what the problem is. In your Temperature project, use BlueJ's **New Class** button to create a new class, and name it InvalidTemperatureTypeException. Once you've created the class, set it up like this:

```
/**
 * InvalidTemperatureTypeException is an exception that gets thrown if
 * a user tried to create a Temperature object using an invalid temperature
 * type.
 *
 * @author Merrill Hall
 * @version 1.0
 */
public class InvalidTemperatureTypeException extends Exception {
    /**
     * Default constructor for InvalidTemperatureTypeException objects. It
     * creates an InvalidTemperatureTypeException with a null detail
     * message.
     */
    public InvalidTemperatureTypeException() {
        super();
    }

    /**
     * Constructor for InvalidTemperatureTypeException objects. It creates
     * an InvalidTemperatureTypeException object with the specified detail
     * message.
     */
    public InvalidTemperatureTypeException(String message) {
        super(message);
    }
}
```

I'll mention several new things about this class, but we won't get into much detail about them as that would take too long and be too far off our main topic. I'll cover some of them in more detail in the *Intermediate Java Programming* course.

This class has a new keyword in its first line: extends. That keyword tells Java we want this class to be a *subclass*, or *child*, of another class (in this case, the class is Exception). So we can also say that Exception is the *superclass*, or *parent*, of this one. What that means is that our class will *inherit*, or have, all the attributes and methods that Exception has.



We won't use any of Exception's attributes or methods in this case except for Exception's constructor, which we call with the call to super(). The *super keyword* tells Java we want to refer to the superclass and call its constructor from this class's constructor. It's common for a constructor in a subclass to call the constructor of its superclass in this way. That's how we can initialize the superclass attributes properly. (As I mentioned in Lesson 3, when you initialize something, you give it a value to start off with, even though you may change that value later on.)

In this case our default constructor creates an exception object with a null message. The other constructor allows its user to include an error message. In either case the exception name will display to let the user know what type of problem we ran into. In our Temperature class, we'll include an error message to specify more details. You'll see how we do that in just a minute.

There's only one other thing I need to mention right now, and then we'll get on with using our exception class.

You may have noticed that the only methods I defined for our InvalidTemperatureTypeException class are its constructors. That's because the class inherits from the Exception class all the other methods that it needs for an exception. If you'd like to see those methods, the list is available in the Java API documentation for Exception. The Exception class in turn inherits its methods from the Throwable class, so if you're looking into the details, you may need to look there too.

Now back to our main discussion. How can we use this new exception class we've created? Let me show you in a new revision to Temperature's constructor:

```
/**
     * This constructor for Temperature sets the temperature
     * values to the value from degrees, based on the type
     *
     * @param  type      temperature scale to use
     * @param  degrees   degrees Fahrenheit
     * @throws InvalidTemperatureTypeException if type is not C, F, or K
     */
    public Temperature(String type, double degrees)
            throws InvalidTemperatureTypeException {
        if ( ! isTypeValid(type) )
            throw new InvalidTemperatureTypeException(type + " is not a valid
type.");
        if (type.equalsIgnoreCase("C"))
            setDegreesCelsius(degrees);
        else if (type.equalsIgnoreCase("F"))
            setDegreesFahrenheit(degrees);
        else if (type.equalsIgnoreCase("K"))
            setDegreesKelvin(degrees);
    }
```

I've made two changes to the constructor. The first is to add this clause to the method declaration so that the compiler will know that this method may throw the exception:

```
throws InvalidTemperatureTypeException {
```

Then I added this code to check the type and throw the exception if need be:

```
if ( ! isTypeValid(type) )
    throw new InvalidTemperatureTypeException(type + " is not a valid type.");
```

This code creates a new InvalidTemperatureTypeException object with an appropriate message, and then it throws it to the calling code.

If you've tried compiling your project at this point, you may have noticed that the driver program won't compile any longer! The compiler is now aware that our Temperature constructor can throw a checked exception, and our driver doesn't handle it or pass it on. Since the calling method is our main() method, I'm going to add a try-catch handler for the exception. Java allows the main() method to pass on a checked exception, but programmers consider that bad coding.

We'll need to handle our exception when we call our constructor, so I'm going to put the constructor call and all the code after it into the try block and then catch our new exception. Here's what the new try-catch block in my driver program looks like:

```
try {
    t1 = new Temperature(temperatureType, inputTemperature);
    if (temperatureType.equalsIgnoreCase("F")) {
        System.out.println("You entered " + inputTemperature +
                           " degrees Fahrenheit");
        System.out.println("which is " + t1.getDegreesCelsius() +
                           " degrees Celsius");
        System.out.println("and " + t1.getDegreesKelvin() +
                           " degrees Kelvin.");
    }
    else if (temperatureType.equalsIgnoreCase("C")) {
        System.out.println("You entered " + inputTemperature +
                           " degrees Celsius");
        System.out.println("which is " + t1.getDegreesFahrenheit() +
                           " degrees Fahrenheit");
        System.out.println("and " + t1.getDegreesKelvin() +
                           " degrees Kelvin.");
    }
    else if (temperatureType.equalsIgnoreCase("K")) {
        System.out.println("You entered " + inputTemperature +
                           " degrees Kelvin");
        System.out.println("which is " + t1.getDegreesCelsius() +
                           " degrees Celsius");
        System.out.println("and " + t1.getDegreesFahrenheit() +
                           " degrees Fahrenheit.");
    }
}
catch (InvalidTemperatureTypeException e) {
    System.err.println(e);
    System.exit(1);
}
```

I needed to move all the code into the try block since it uses the variable t1. The try block initializes the t1 variable, so the code using it needs to be in the try block too. If it isn't, the compiler will tell us that it might not be initialized.

There's one more exception to add to our Temperature class, but I'll leave that for this lesson's assignment. Please join me in Lesson 5 for a summary.