Matt Weisfeld

Fifth Edition

The Object-Oriented Thought Process

The Object-Oriented Thought Process

Fifth Edition

If an exception is thrown within the `try` block, the `catch` block will handle it. When an exception is thrown while the block is executing, the following occurs:

1. The execution of the `try` block is terminated.
2. The `catch` clauses are checked to determine whether an appropriate `catch` block for the offending exception was included. (There might be more than one `catch` clause per `try` block.)
3. If none of the `catch` clauses handles the offending exception, it is passed to the next higher-level `try` block. (If the exception is not caught in the code, the system ultimately catches it, and the results are unpredictable—that is, an application crash.)
4. If a `catch` clause is matched (the first match encountered), the statements in the `catch` clause are executed.
5. Execution then resumes with the statement following the `try` block.

Suffice it to say that exceptions are an important advantage for OO programming languages. Here is an example of how an exception is caught in Java:

```
try {

    // possible nasty code
    count = 0;
    count = 5/count;

} catch(ArithmeticException e) {

    // code to handle the exception
    System.out.println(e.getMessage());
    count = 1;

}
System.out.println("The exception is handled.");
```

> **Exception Granularity**
>
> You can catch exceptions at various levels of granularity. You can catch all exceptions or check for specific exceptions, such as arithmetic exceptions. If your code does not catch an exception, the Java runtime will—and it won't be happy about it!

In this example, the division by zero (because `count` is equal to `0`) within the `try` block will cause an arithmetic exception. If the exception was generated (thrown) outside a `try` block, the program would most likely have been terminated (crashed). However, because the exception was thrown within a `try` block, the `catch` block is checked to see whether the specific exception (in this case, an arithmetic exception) was planned for. Because the `catch` block contains a check for the arithmetic exception, the code within the `catch` block is executed, thus setting `count`

to 1. After the `catch` block executes, the `try/catch` block is exited, and the message `The exception is handled.` appears on the Java console. The logical flow of this process is illustrated in Figure 3.5.

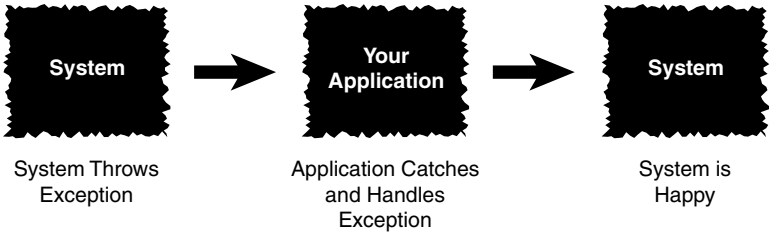

Figure 3.5 Catching an exception.

If you had not put `ArithmeticException` in the `catch` block, the program would likely have crashed. You can catch all exceptions by using the following code:

```
try {

    // possible nasty code

} catch(Exception e) {

    // code to handle the exception
}
```

The `Exception` parameter in the `catch` block is used to catch any exception that might be generated within the scope of a `try` block.

> **Bulletproof Code**
>
> It's a good idea to use a combination of the methods described here to make your program as bulletproof to your user as possible.

The Importance of Scope

Multiple objects can be instantiated from a single class. Each of these objects has a unique identity and state. This is an important point. Each object is constructed separately and is allocated its own separate memory. However, some attributes and methods may, if properly declared, be shared by all the objects instantiated from the same class, thus sharing the memory allocated for these class attributes and methods.

> **A Shared Method**
>
> A constructor is a good example of a method that is shared by all instances of a class.

Methods represent the behaviors of an object; the state of the object is represented by attributes. There are three types of attributes:

- Local attributes
- Object attributes
- Class attributes

Local Attributes

Local attributes are owned by a specific method. Consider the following code:

```
public class Number {

    public method1() {
        int count;

    }

    public method2() {

    }

}
```

The method `method1` contains a local variable called `count`. This integer is accessible only inside `method1`. The method `method2` has no idea that the integer `count` even exists.

At this point, we introduce a very important concept: scope. Attributes (and methods) exist within a particular scope. In this case, the integer count exists within the scope of `method1`. In Java, C#, C++, and Swift, scope is delineated by curly braces ({}). In the `Number` class, there are several possible scopes—just start matching the curly braces.

The class itself has its own scope. Each instance of the class (that is, each object) has its own scope. Both `method1` and `method2` have their own scopes as well. Because `count` lives within `method1`'s curly braces, when `method1` is invoked, a copy of `count` is created. When `method1` terminates, the copy of `count` is removed.

For some more fun, look at this code:

```
public class Number {

    public method1() {
        int count;
    }

    public method2() {
        int count;
    }

}
```

This example has two copies of an integer `count` in this class. Remember that `method1` and `method2` each has its own scope. Thus, the compiler can tell which copy of `count` to access simply by recognizing which method it is in. You can think of it in these terms:

```
method1.count;

method2.count;
```

As far as the compiler is concerned, the two attributes are easily differentiated, even though they have the same name. It is almost like two people having the same last name, but based on the context of their first names, you know that they are two separate individuals.

Object Attributes

In many design situations, an attribute must be shared by several methods within the same object. In Figure 3.6, for example, three objects have been constructed from a single class. Consider the following code:

```
public class Number {

    int count;    // available to both method1 and method2

    public method1() {
        count = 1;
    }

    public method2() {
        count = 2;
    }

}
```

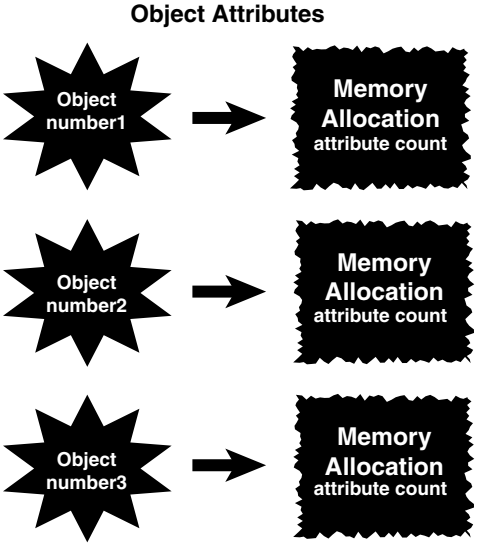

Figure 3.6 Object attributes.

Note here that the class attribute `count` is declared outside the scope of both `method1` and `method2`. However, it is within the scope of the class. Thus, `count` is available to both `method1` and `method2`. (Basically, all methods in the class have access to this attribute.) Notice that the code for both methods is setting `count` to a specific value. There is only one copy of `count` for the entire object, so both assignments operate on the same copy in memory. However, this copy of `count` is not shared between different objects.

To illustrate, let's create three copies of the `Number` class:

```
Number number1 = new Number();
Number number2 = new Number();
Number number3 = new Number();
```

Each of these objects—`number1`, `number2`, and `number3`—is constructed separately and is allocated its own resources. There are three separate instances of the integer `count`. When `number1` changes its attribute `count`, this in no way affects the copy of `count` in object `number2` or object `number3`. In this case, integer `count` is an *object attribute*.

You can play some interesting games with scope. Consider the following code:

```
public class Number {

    int count;

    public method1() {
        int count;
    }

    public method2() {
        int count;
    }

}
```

In this case, three totally separate memory locations have the name of `count` for each object. The object owns one copy, and `method1()` and `method2()` each have their own copy.

To access the object variable from within one of the methods, say `method1()`, you can use a pointer called `this` in the C-based languages:

```
public method1() {
        int count;

        this.count = 1;
}
```

Notice that some code looks a bit curious:

```
this.count = 1;
```

The selection of the word `this` as a keyword is perhaps unfortunate. However, we must live with it. The use of the `this` keyword directs the compiler to access the object variable `count` and not the local variables within the method bodies.

> **Note**
> The keyword `this` is a reference to the current object.

Class Attributes

As mentioned earlier, it is possible for two or more objects of the same class to share attributes. In Java, C#, C++, and Swift, you do this by making the attribute *static*:

```
public class Number {

    static int count;

    public method1() {
    }

}
```

By declaring `count` as static, this attribute is allocated a single piece of memory for all objects instantiated from the class. Thus, all objects of the class use the same memory location for `count`. Essentially, each class has a single copy, which is shared by all objects of that class (see Figure 3.7). This is about as close to global data as we get in OO design.

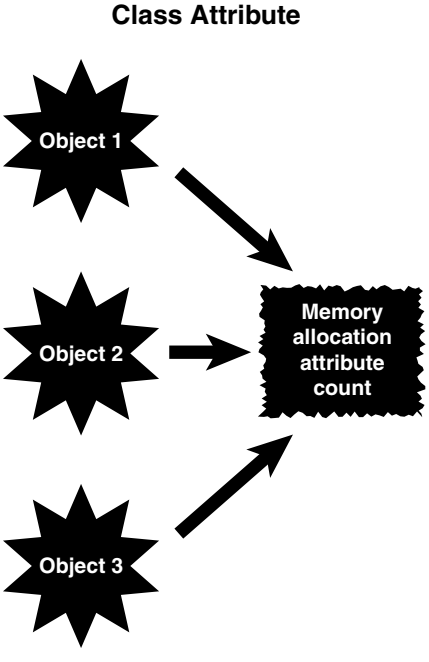

Figure 3.7 Class attributes.