ADVANCED JAVA LANGUAGE FEATURES

Exceptions in Java, Part 1: Exception handling basics

Throwing, trying, catching, and cleaning up after Java exceptions

Java exceptions are library types and language features used to represent and deal with program failure. If you've wanted to understand how failure is represented in source code, you've come to the right place. In addition to an overview of Java exceptions, I'll get you started with Java's language features for throwing objects, trying code that may fail, catching thrown objects, and cleaning up your Java code after an exception has been thrown.

In the first half of this tutorial you'll learn about basic language features and library types that have been around since Java 1.0. In the second half, you'll discover advanced capabilities introduced in more recent Java versions.

Note that code examples in this tutorial are compatible with JDK 12.

Get the code

Download the source code for example applications in this tutorial. *Created by Jeff Friesen for JavaWorld.*

What are Java exceptions?

Failure occurs when a Java program's normal behavior is interrupted by unexpected behavior. This divergence is known as an *exception*. For example, a program tries to open a file to read its contents, but the file doesn't exist. Java classifies exceptions into a few types, so let's consider each one.

Checked exceptions

Java classifies exceptions arising from external factors (such as a missing file) as *checked exceptions*. The Java compiler checks that such exceptions are either *handled* (corrected) where they occur or documented to be handled elsewhere.

Exception handlers

An *exception handler* is a sequence of code that handles an exception. It interrogates the context--meaning that it reads values saved from variables that were in scope at the time the exception occurred--then uses what it learns to restore the Java program to a flow of normal behavior. For example, an exception handler might read a saved filename and promp the user to replace the missing file.

Runtime (unchecked) exceptions

Suppose a program attempts to divide an integer by integer 0. This impossibility illustrates another kind of exception, namely a *runtime exception*. Unlike checked exceptions, runtime exceptions typically arise from poorly written source code, and should thus be fixed by the programmer. Because the compiler doesn't check that runtime exceptions are handled or documented to be handled elsewhere, you can think of a runtime exception as an *unchecked exception*.

About runtime exceptions

You might modify a program to handle a runtime exception, but it's better to fix the source code. Runtime exceptions often arise from passing invalid arguments to a library's methods; the buggy calling code should be fixed.

Errors

Some exceptions are very serious because they jeopardize a program's ability to continue execution. For example, a program tries to allocate memory from the JVM but there isn't enough free memory to satisfy the request. Another serious situation occurs when a program tries to load a classfile via a `Class.forName()` method call, but the classfile is corrupt. This kind of exception is known as an *error*. You should never try to handle errors yourself because the JVM might not be able to recover from it.

Exceptions in source code

An exception may be represented in source code as an *error code* or as an *object*. I'll introduce both and show you why objects are superior.

Error codes versus objects

Programming languages such as C use integer-based *error codes* to represent failure and reasons for failure--i.e., exceptions. Here are a couple of examples:

```
if (chdir("C:\\temp"))
   printf("Unable to change to temp directory: %d\n", errno);

FILE *fp = fopen("C:\\temp\\foo");
if (fp == NULL)
   printf("Unable to open foo: %d\n", errno);
```

C's `chdir()` (change directory) function returns an integer: 0 on success or -1 on failure. Similarly, C's `fopen()` (file open) function returns a nonnull *pointer* (integer address) to a `FILE` structure on success or a null (0) pointer (represented by constant `NULL`) on failure. In either case, to identify the exception that caused the failure, you must read the global `errno` variable's integer-based error code.

Error codes present some problems:

- Integers are meaningless; they don't describe the exceptions they represent. For example, what does 6 mean?
- Associating context with an error code is awkward. For example, you might want to output the name of the file that couldn't be opened, but where are you going to store the file's name?
- Integers are arbitrary, which can lead to confusion when reading source code. For example, specifying `if (!chdir("C:\\temp"))` (`!` signifies NOT) instead of `if (chdir("C:\\temp"))` to test for failure is clearer. However, 0 was chosen to indicate success, and so `if (chdir("C:\\temp"))` must be specified to test for failure.

- Error codes are too easy to ignore, which can lead to buggy code. For example, the programmer could specify `chdir("C:\\temp");` and ignore the `if (fp == NULL)` check. Furthermore, the programmer need not examine `errno`. By not testing for failure, the program behaves erratically when either function returns a failure indicator.

To solve these problems, Java embraced a new approach to exception handling. In Java, we combine objects that describe exceptions with a mechanism based on throwing and catching these objects. Here are some advantages of using objects versus error code to denote exceptions:

- An object can be created from a class with a meaningful name. For example, `FileNotFoundException` (in the `java.io` package) is more meaningful than 6.
- Objects can store context in various fields. For example, you can store a message, the name of the file that could not be opened, the most recent position where a parse operation failed, and/or other items in an object's fields.
- You don't use `if` statements to test for failure. Instead, exception objects are thrown to a handler that's separate from the program code. As a result, the source code is easier to read and less likely to be buggy.

Throwable and its subclasses

Java provides a hierarchy of classes that represent different kinds of exceptions. These classes are rooted in the `java.lang` package's `Throwable` class, along with its `Exception`, `RuntimeException`, and `Error` subclasses.

`Throwable` is the ultimate superclass where exceptions are concerned. Only objects created from `Throwable` and its subclasses can be thrown (and subsequently caught). Such objects are known as *throwables*.

A `Throwable` object is associated with a *detail message* that describes an exception. Several constructors, including the pair described below, are provided to create a `Throwable` object with or without a detail message:

- **Throwable()** creates a `Throwable` with no detail message. This constructor is appropriate for situations where there is no context. For example, you only want to know that a stack is empty or full.
- **Throwable(String message)** creates a `Throwable` with `message` as the detail message. This message can be output to the user and/or logged.

`Throwable` provides the `String getMessage()` method to return the detail message. It also provides additional useful methods, which I'll introduce later.

The Exception class

`Throwable` has two direct subclasses. One of these subclasses is `Exception`, which describes an exception arising from an external factor (such as attempting to read from a nonexistent file). `Exception` declares the same constructors (with identical parameter lists) as `Throwable`, and each constructor invokes its `Throwable` counterpart. `Exception` inherits `Throwable`'s methods; it declares no new methods.

Java provides many exception classes that directly subclass `Exception`. Here are three examples:

- **CloneNotSupportedException** signals an attempt to clone an object whose class doesn't implement the `Cloneable` interface. Both types are in the `java.lang` package.
- **IOException** signals that some kind of I/O failure has occurred. This type is located in the `java.io` package.
- **ParseException** signals that a failure has occurred while parsing text. This type can be found in the `java.text` package.

Notice that each `Exception` subclass name ends with the word `Exception`. This convention makes it easy to identify the class's purpose.

You'll typically subclass `Exception` (or one of its subclasses) with your own exception classes (whose names should end with `Exception`). Here are a couple of custom subclass examples:

```
public class StackFullException extends Exception
{
}

public class EmptyDirectoryException extends Exception
{
   private String directoryName;

   public EmptyDirectoryException(String message, String directoryName)
   {
      super(message);
      this.directoryName = directoryName;
   }

   public String getDirectoryName()
   {
      return directoryName;
   }
}
```

The first example describes an exception class that doesn't require a detail message. It's default noargument constructor invokes `Exception()`, which invokes `Throwable()`.

The second example describes an exception class whose constructor requires a detail message and the name of the empty directory. The constructor invokes `Exception(String message)`, which invokes `Throwable(String message)`.

Objects instantiated from `Exception` or one of its subclasses (except for `RuntimeException` or one of its subclasses) are checked exceptions.

The RuntimeException class

`Exception` is directly subclassed by `RuntimeException`, which describes an exception most likely arising from poorly written code. `RuntimeException` declares the same constructors (with identical parameter lists) as `Exception`, and each constructor invokes its `Exception` counterpart. `RuntimeException` inherits `Throwable`'s methods. It declares no new methods.

Java provides many exception classes that directly subclass `RuntimeException`. The following examples are all members of the `java.lang` package:

- **ArithmeticException** signals an illegal arithmetic operation, such as attempting to divide an integer by 0.
- **IllegalArgumentException** signals that an illegal or inappropriate argument has been passed to a method.
- **NullPointerException** signals an attempt to invoke a method or access an instance field via the null reference.

Objects instantiated from `RuntimeException` or one of its subclasses are *unchecked exceptions*.

The Error class

`Throwable`'s other direct subclass is `Error`, which describes a serious (even abnormal) problem that a reasonable application should not try to handle--such as running out of memory, overflowing the JVM's stack, or attempting to load a class that cannot be found. Like `Exception`, `Error` declares identical constructors to `Throwable`, inherits `Throwable`'s methods, and doesn't declare any of its own methods.

You can identify `Error` subclasses from the convention that their class names end with `Error`. Examples include `OutOfMemoryError`, `LinkageError`, and `StackOverflowError`. All three types belong to the `java.lang` package.

Throwing exceptions

A C library function notifies calling code of an exception by setting the global `errno` variable to an error code and returning a failure code. In contrast, a Java method throws an object. Knowing how and when to throw exceptions is an essential aspect of effective Java programming. Throwing an exception involves two basic steps:

1. Use the `throw` statement to throw an exception object.
2. Use the `throws` clause to inform the compiler.

Later sections will focus on catching exceptions and cleaning up after them, but first let's learn more about throwables.

The throw statement

Java provides the throw statement to throw an object that describes an exception. Here's the syntax of the throw statement :

```
throw throwable;
```

The object identified by *throwable* is an instance of Throwable or any of its subclasses. However, you usually only throw objects instantiated from subclasses of Exception or RuntimeException. Here are a couple of examples:

```
throw new FileNotFoundException("unable to find file " + filename);

throw new IllegalArgumentException("argument passed to count is less than zero");
```

The throwable is thrown from the current method to the JVM, which checks this method for a suitable handler. If not found, the JVM unwinds the method-call stack, looking for the closest calling method that can handle the exception described by the throwable. If it finds this method, it passes the throwable to the method's handler, whose code is executed to handle the exception. If no method is found to handle the exception, the JVM terminates with a suitable message.

The throws clause

You need to inform the compiler when you throw a checked exception out of a method. Do this by appending a throws clause to the method's header. This clause has the following syntax:

```
throws checkedExceptionClassName (, checkedExceptionClassName)*
```

A throws clause consists of keyword throws followed by a comma-separated list of the class names of checked exceptions thrown out of the method. Here is an example:

```
public static void main(String[] args) throws ClassNotFoundException
{
   if (args.length != 1)
   {
      System.err.println("usage: java ... classfile");
      return;
   }
   Class.forName(args[0]);
}
```

This example attempts to load a classfile identified by a command-line argument. If `Class.forName()` cannot find the classfile, it throws a `java.lang.ClassNotFoundException` object, which is a checked exception.

Page 1 of 3