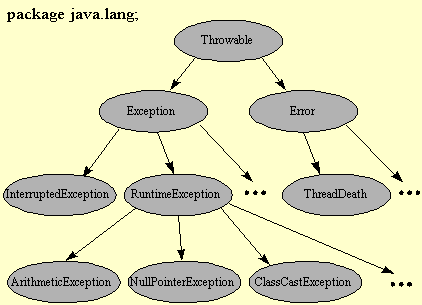
Objects and Java by Bill Venners  
Chapter 9:  
Exceptions

As mentioned in the previous chapter, exceptions are the customary way in Java to indicate to a calling method that an abnormal condition occurred. This chapter explains how exceptions work and gives advice on their use.

**Exception Classes**

When a method encounters an abnormal condition (an *exception condition*) that it can't handle itself, it may *throw* an exception. Throwing an exception is like throwing a beeping, flashing red ball to indicate there is a problem that can't be handled where it occurred. Somewhere, you hope, this ball will be caught and the problem handled. Exceptions are *caught* by handlers positioned along the thread's method invocation stack. If the calling method isn't prepared to catch the exception, then it throws the exception up to its calling method, and so on. If one of the threads of your program throws an exception that isn't caught by any method along the method invocation stack, that thread will expire. When you program in Java, you must position catchers strategically, so your program will catch and handle all exceptions that shouldn't result in a dead thread.

In Java, exceptions are objects. When you throw an exception, you throw an object. You can't throw just any object as an exception, however, only objects whose class descends from Throwable. Throwable serves as the base class for an entire family of classes, declared in java.lang, that your program can instantiate and throw. A small part of this family is shown in Figure 9-1.

  
Figure 9-1. A partial view of the Throwable family.

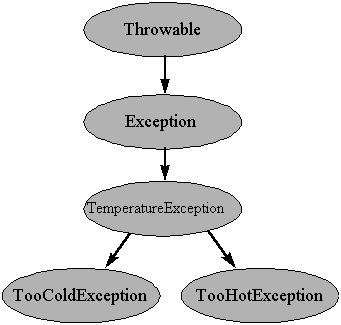
As you can see in Figure 9-1, Throwable has two direct subclasses, Exception and Error. Exceptions (members of the Exception family) are thrown for abnormal conditions that can often be handled by some catcher, though they may not be caught and could therefore result in a dead thread. Errors (members of the Error family) are usually thrown for more serious problems, such as OutOfMemoryError, that may not be so easy to handle. In general, code you write should throw only exceptions, not errors. Errors are usually thrown by the methods of the Java API, or by the Java Virtual Machine itself.

In addition to throwing objects whose class is declared in java.lang, you can throw objects of your own design. To create your own class of throwable objects, you need only declare it as a subclass of some member of the Throwable family. In general, however, the throwable classes you define should extend class Exception. They should be "exceptions." The reasoning behind this rule will be explained later in this chapter.

Whether you use an existing exception class from java.lang or create one of your own depends upon the situation. In some cases, a class from java.lang will do just fine. For example, if one of your methods is invoked with an invalid argument, you could throw IllegalArgumentException, a subclass of RuntimeException in java.lang.

Other times, however, you will want to convey more information about the abnormal condition than a class from java.lang will allow. Usually, the class of the exception object itself indicates the type of abnormal condition that was encountered. For example, if a thrown exception object has class IllegalArgumentException, that indicates someone passed an illegal argument to a method. Sometimes you will want to indicate a method encountered an abnormal condition that isn't represented by a class in the Throwable family of java.lang.

As an example, imagine you are writing a Java program that simulates a customer of a virtual cafe drinking a cup of coffee. Consider the exceptional conditions that might occur while the customer sips. The class hierarchy of exceptions shown in Figure 9-2 represents a few possibilities.

  
Figure 9-2. Exception hierarchy for coffee sipping.

If the customer discovers, with dismay, that the coffee is cold, your program could throw a TooColdException. On the other hand, if the customer discovers that the coffee is overly hot, your program could throw a TooHotException. These conditions could be exceptions because they are (hopefully) not the *normal* situation in your cafe. (Exceptional conditions are not necessarily rare, just outside the normal flow of events.) The code for your new exception classes might look like this:

// In Source Packet in file except/ex1/TemperatureException.java

class TemperatureException extends Exception {

}

// In Source Packet in file except/ex1/TooColdException.java

class TooColdException extends TemperatureException {

}

// In Source Packet in file except/ex1/TooHotException.java

class TooHotException extends TemperatureException {

}

This family of classes, the TemperatureException family, declares three new types of exceptions for your program to throw. Note that each exception indicates by its class the kind of abnormal condition that would cause it to be thrown. TemperatureException indicates some kind of problem with temperature. TooColdException indicates something was too cold.TooHotException indicates something was too hot. Note also that TemperatureException extends Exception--not Throwable, Error, or any other class declared in java.lang.

**Throwing Exceptions**

To throw an exception, you simply use the throw keyword with an object reference, as in:

throw new TooColdException();

The type of the reference must be Throwable or one of its subclasses.

The following code shows how a class that represents the customer, class VirtualPerson, might throw exceptions if the coffee didn't meet the customer's temperature preferences. Note that Java also has a throws keyword in addition to the throw keyword. Only throw can be used to throw an exception. The meaning of throws will be explained later in this chapter.

// In Source Packet in file except/ex1/VirtualPerson.java

class VirtualPerson {

private static final int tooCold = 65;

private static final int tooHot = 85;

public void drinkCoffee(CoffeeCup cup) throws

TooColdException, TooHotException {

int temperature = cup.getTemperature();

if (temperature <= tooCold) {

throw new TooColdException();

}

else if (temperature >= tooHot) {

throw new TooHotException();

}

//...

}

//...

}

// In Source Packet in file except/ex1/CoffeeCup.java

class CoffeeCup {

// 75 degrees Celsius: the best temperature for coffee

private int temperature = 75;

public void setTemperature(int val) {

temperature = val;

}

public int getTemperature() {

return temperature;

}

//...

}

**Catching Exceptions**

To catch an exception in Java, you write a try block with one or more catch clauses. Each catch clause specifies one exception type that it is prepared to handle. The try block places a fence around a bit of code that is under the watchful eye of the associated catchers. If the bit of code delimited by the try block throws an exception, the associated catch clauses will be examined by the Java Virtual Machine. If the virtual machine finds a catch clause that is prepared to handle the thrown exception, the program continues execution starting with the first statement of that catch clause.

As an example, consider a program that requires one argument on the command line, a string that can be parsed into an integer. When you have a String and want an int, you can invoke theparseInt() method of the Integer class. If the string you pass represents an integer, parseInt() will return the value. If the string doesn't represent an integer, parseInt() throwsNumberFormatException. Here is how you might parse an int from a command-line argument:

// In Source Packet in file except/ex1/Example1.java

class Example1 {

public static void main(String[] args) {

int temperature = 0;

if (args.length > 0) {

try {

temperature = Integer.parseInt(args[0]);

}

catch(NumberFormatException e) {

System.out.println(

"Must enter integer as first argument.");

return;

}

}

else {

System.out.println(

"Must enter temperature as first argument.");

return;

}

// Create a new coffee cup and set the temperature of

// its coffee.

CoffeeCup cup = new CoffeeCup();

cup.setTemperature(temperature);

// Create and serve a virtual customer.

VirtualPerson cust = new VirtualPerson();

VirtualCafe.serveCustomer(cust, cup);

}

}

Here, the invocation of parseInt() sits inside a try block. Attached to the try block is a catch clause that catches NumberFormatException:

catch(NumberFormatException e) {

System.out.println(

"Must enter integer as first argument.");

return;

}

The lower case character e is a reference to the thrown (and caught) NumberFormatException object. This reference could have been used inside the catch clause, although in this case it isn't. (Examples of catch clauses that use the reference are shown later in this chapter.)

If the user types "Harumph" as the first argument to the Example1 program, parseInt() will throw a NumberFormatException exception and the catch clause will catch it. The program will print:

Must enter integer as first argument.

Although the above example had only one catch clause, you can have many catch clauses associated with a single try block. Here's an example:

// In Source Packet in file except/ex1/VirtualCafe.java

class VirtualCafe {

public static void serveCustomer(VirtualPerson cust,

CoffeeCup cup) {

try {

cust.drinkCoffee(cup);

System.out.println("Coffee is just right.");

}

catch (TooColdException e) {

System.out.println("Coffee is too cold.");

// Deal with an irate customer...

}

catch (TooHotException e) {

System.out.println("Coffee is too hot.");

// Deal with an irate customer...

}

}

}

If any code inside a try block throws an exception, its catch clauses are examined in their order of appearance in the source file. For example, if the try block in the above example throws an exception, the catch clause for TooColdException will be examined first, then the catch clause for TooHotException. During this examination process, the first catch clause encountered that handles the class of object thrown gets to "catch" the exception. The ordering of catch clause examination matters because it is possible that multiple catch clauses of a try block could handle the same exception.

Catch clauses indicate the type of abnormal condition they handle by the type of exception reference they declare. In the example above, the catch clauses declare exception typeTooColdException and TooHotException. Had a single catch clause declared a TemperatureException, a thrown TooColdException or TooHotException would have still been caught, because TemperatureException is their superclass. In the object- oriented way of thinking, a TooColdException *is-a* TemperatureException, therefore, a catch clause forTemperatureException will also catch a thrown TooColdException. An example of this variety is shown below:

// In Source Packet in file except/ex2/VirtualCafe.java

class VirtualCafe {

public static void serveCustomer(VirtualPerson cust,

CoffeeCup cup) {

try {

cust.drinkCoffee(cup);

System.out.println("Coffee is just right.");

}

catch (TemperatureException e) {

// This catches TooColdException, TooHotException,

// as well as TemperatureException.

System.out.println("Coffee is too cold or too hot.");

// Deal with an irate customer...

}

}

}

Multiple catch clauses could handle the same exception because you could, for example, declare two catch clauses, one for TooColdException and another for TemperatureException. In this case, however, you must place the catch clause for TooColdException above the one for TemperatureException, or the source file won't compile. If a catch clause forTemperatureException could be declared before a catch clause for TooColdException, the first catch clause would catch all TooColdExceptions, leaving nothing for the second catch clause to do. The second catch clause would never be reached. The general rule is: subclass catch clauses must precede superclass catch clauses. Here's an example of both orders, only one of which compiles:

// In Source Packet in file except/ex3/VirtualCafe.java

class VirtualCafe {

public static void serveCustomer(VirtualPerson cust,

CoffeeCup cup) {

try {

cust.drinkCoffee(cup);

System.out.println("Coffee is just right.");

}

catch (TemperatureException e) {

// This catches TooColdException, TooHotException,

// as well as TemperatureException.

System.out.println("Coffee is too cold or too hot.");

// Deal with an irate customer...

}

// THIS WON'T COMPILE, BECAUSE THIS CATCH CLAUSE

// WILL NEVER BE REACHED.

catch (TooColdException e) {

System.out.println("Coffee is too cold.");

}

}

}

// In Source Packet in file except/ex4/VirtualCafe.java

// This class compiles fine.

class VirtualCafe {

public static void serveCustomer(VirtualPerson cust,

CoffeeCup cup) {

try {

cust.drinkCoffee(cup);

System.out.println("Coffee is just right.");

}

catch (TooColdException e) {

System.out.println("Coffee is too cold.");

// Deal with an irate customer...

}

catch (TemperatureException e) {

// This catches TooHotException as well

// as TemperatureException.

System.out.println(

"There's temperature trouble in this coffee.");

// Deal with an irate customer...

}

}

}

**Embedding Information in an Exception Object**

When you throw an exception, you are performing a kind of structured "go to" from the place in your program where an abnormal condition was detected to a place where it can be handled. The Java Virtual Machine uses the class of the exception object you throw to decide which catch clause, if any, should be allowed to handle the exception. But an exception doesn't just transfer control from one part of your program to another, it also transmits information. Because the exception is a full-fledged object that you can define yourself, you can embed information about the abnormal condition in the object before you throw it. The catch clause can then get the information by querying the exception object directly.

The Exception class allows you to specify a String detail message that can be retrieved by invoking getMessage() on the exception object. When you define an exception class of your own, you can give client programmers the option of specifying a detail message like this:

// In Source Packet in file except/ex5/UnusualTasteException.java

class UnusualTasteException extends Exception {

UnusualTasteException() {

}

UnusualTasteException(String msg) {

super(msg);

}

}

Given the above declaration of UnusualTasteException, client programmers could create an instance in either of two ways:

1. new UnusualTasteException()
2. new UnusualTasteException("This coffee tastes like tea.")

A catch clause can then query the object for a detail string, like this:

// In Source Packet in file except/ex5/VirtualCafe.java

class VirtualCafe {

public static void serveCustomer(VirtualPerson cust,

CoffeeCup cup) {

try {

cust.drinkCoffee(cup);

System.out.println("Coffee tastes just right.");

}

catch (UnusualTasteException e) {

System.out.println(

"Customer is complaining of an unusual taste.");

String s = e.getMessage();

if (s != null) {

System.out.println(s);

}

// Deal with an unhappy customer...

}

}

}

When you need to embed more information into an exception object than you can represent with a String, you can add data and access methods to your exception class. For example, you could define the temperature exception classes like this:

// In Source Packet in file except/ex6/TemperatureException.java

abstract class TemperatureException extends Exception {

private int temperature; // in Celsius

public TemperatureException(int temperature) {

this.temperature = temperature;

}

public int getTemperature() {

return temperature;

}

}

// In Source Packet in file except/ex6/TooColdException.java

class TooColdException extends TemperatureException {

public TooColdException(int temperature) {

super(temperature);

}

}

// In Source Packet in file except/ex6/TooHotException.java

class TooHotException extends TemperatureException {

public TooHotException(int temperature) {

super(temperature);

}

}

Note that this version of the TemperatureException family uses constructors in more elaborate ways than you've seen in previous chapters. Constructors are explained in detail in the next chapter. At this point, the important thing to understand is that this design *forces* client programmers to specify a temperature when they instantiate a TooColdException or TooHotException.

Note also that this version of the TemperatureException family illustrates a case in which an abstract base class makes sense. Because the TemperatureException is abstract, client programmers can't instantiate it. If they want to indicate a temperature problem, they'll have to be more specific than just TemperatureException. They'll have to instantiate eitherTooColdException or TooHotException. Still, all members of the TemperatureException family inherit the implementation and interface of the base class. A client programmer can invokegetTemperature() on any member of this family. Note that the TemperatureException family also illustrates a situation in which it makes sense to have an abstract class in the middle of an inheritance hierarchy. TemperatureException's superclasses-- Exception, Throwable, and Object--are not abstract.

Given a TemperatureException family as defined above, catch clauses can query the exception object to find out the precise temperature that caused the problem. The temperature field of the exception object must be set when the object is created, as in:

// In Source Packet in file except/ex6/VirtualPerson.java

class VirtualPerson {

private static final int tooCold = 65;

private static final int tooHot = 85;

public void drinkCoffee(CoffeeCup cup) throws

TooColdException, TooHotException {

int temperature = cup.getTemperature();

if (temperature <= tooCold) {

throw new TooColdException(temperature);

}

else if (temperature >= tooHot) {

throw new TooHotException(temperature);

}

//...

}

//...

}

Wherever the exception is caught, the catch clause can easily determine the actual temperature of the coffee and act accordingly, as in:

// In Source Packet in file except/ex6/VirtualCafe.java

class VirtualCafe {

public static void serveCustomer(VirtualPerson cust,

CoffeeCup cup) {

try {

cust.drinkCoffee(cup);

System.out.println("Coffee is just right.");

}

catch (TooColdException e) {

int temperature = e.getTemperature();

System.out.println("Coffee temperature is "

+ temperature + " degrees Celsius.");

if (temperature > 55 && temperature <= 65) {

System.out.println("Coffee is cooling off.");

// Add more hot coffee...

}

else if (temperature > 0 && temperature <= 55) {

System.out.println("Coffee is too cold.");

// Give customer a new cup of coffee with the

// proper temperature...

}

else if (temperature <= 0) {

System.out.println("Coffee is frozen.");

// Deal with an irate customer...

}

}

catch (TooHotException e) {

int temperature = e.getTemperature();

System.out.println("Coffee temperature is "

+ temperature + " degrees Celsius.");

if (temperature >= 85 && temperature < 100) {

System.out.println("Coffee is too hot.");

// Ask customer to let it cool a few minutes...

}

else if (temperature >= 100 && temperature < 2000) {

System.out.println(

"Both coffee and customer are steamed.");

// Deal with an irate customer...

}

else if (temperature >= 2000) {

System.out.println(

"The coffee is plasma.");

// Deal with a very irate customer...

}

}

}

}

The program could deal with the temperature problem differently depending upon the coffee's actual temperature. If the coffee is just a little cold, the program could add more hot coffee to the cup. If the coffee is so cold that the customer's lips were instantly frozen to the cup, alternative measures could be taken.

**Exceptions and the Method Invocation Stack**

Code inside a try block is in a sense surrounded by the catch clauses associated with the try block. When an exception is thrown, the surrounding catch clauses are examined in inside-out order. You can nest try blocks inside try blocks, in effect building up more and more layers of catch clauses that surround the code. When a method is invoked from within a try block, the catch clauses associated with that try block surround the code in the *invoked*method as well. If that method has try blocks and catch clauses, they are added as inner surrounding layers. What this means is that an exception may be thrown far up the method invocation stack before landing in a catch clause that can handle it.

As an example, consider the following exception classes, which are simpler versions of exceptions introduced in examples above:

// In Source Packet in file except/ex7/TemperatureException.java

class TemperatureException extends Exception {

}

// In Source Packet in file except/ex7/TooColdException.java

class TooColdException extends TemperatureException {

}

// In Source Packet in file except/ex7/TooHotException.java

class TooHotException extends TemperatureException {

}

// In Source Packet in file except/ex7/UnusualTasteException.java

class UnusualTasteException extends Exception {

}

When the drinkCoffee() method of class VirtualPerson is invoked, it throws one of these four exceptions, chosen at random:

// In Source Packet in file except/ex7/VirtualPerson.java

class VirtualPerson {

public void drinkCoffee(CoffeeCup cup) throws TooColdException,

TemperatureException, UnusualTasteException {

try {

int i = (int) (Math.random() \* 4.0);

switch (i) {

case 0:

throw new TooHotException();

case 1:

throw new TooColdException();

case 2:

throw new UnusualTasteException();

default:

throw new TemperatureException();

}

}

catch (TooHotException e) {

System.out.println("This coffee is too hot.");

// Customer will wait until it cools to an

// acceptable temperature.

}

}

//...

}

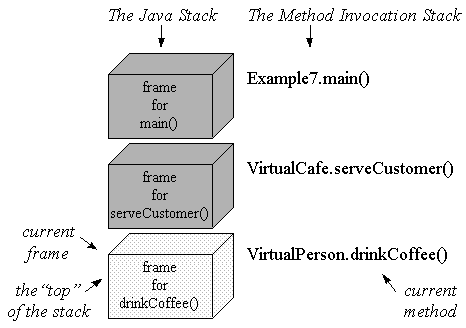
If variable i in the drinkCoffee()method above happens to be set to a value of zero, the switch statement will instantiate and throw a TooHotException. Because the switch statement itself is enclosed within a try block that has a catch clause for TooHotException, execution continues at that catch clause. The program prints out:

This coffee is too hot.

If variable i in the drinkCoffee()method above happens to be set to the value of one, the switch statement will instantiate and throw a TooColdException. When this exception is thrown, the Java Virtual Machine will first check the catch clauses of the try block that surrounds the switch statement. In this case, however, no catch clause matches the thrown exception.

Because the TooColdException is not caught by the drinkCoffee() method, the Java Virtual Machine throws the exception up the method invocation stack to the method that invokeddrinkCoffee(). As used here, a *method invocation stack* (or *call stack*) is a list of the methods that have been invoked by a thread, starting with the first method the thread invoked and ending with the current method. A method invocation stack shows the path of method invocations a thread took to arrive at the current method.

A graphical representation of the method invocation stack for drinkCoffee() is shown in Figure 9-3. In this figure, the method invocation stack is shown on the right and the corresponding Java Stack is shown on the left. The *Java Stack* is where methods keep their state inside the Java Virtual Machine. Each method gets a *stack frame* (or *frame*), which is pushed onto the stack when the method is invoked and popped from the stack when the method completes. The frame is an area in memory that contains the method's local variables, parameters, return value, and other information needed by the Java Virtual Machine to execute the method. (The full details of stack frames are described in Chapter 21.) In Figure 9-3, as in all graphical depictions of the Java Stack in this book, the stack is shown growing downwards. The top of the stack is at the bottom of the picture.

  
Figure 9-3. The method invocation stack for drinkCoffee().

When a method completes by executing a return statement, or by successfully executing the last statement in a method declared as void, it is said to *complete normally*. The Java Virtual Machine pops the returning method's stack frame, and continues executing just after the method invocation in the calling method. The calling method becomes the *current method* and its stack frame becomes the *current frame*.

When a method throws an exception that it doesn't catch itself, it is said to *complete abruptly*. Methods do not return a value when they complete abruptly, though they do pass along an exception object.

For example, when the drinkCoffee() method throws a TooColdException, it completes abruptly. Because the exception isn't caught by drinkCoffee(), the Java Virtual Machine popsdrinkCoffee()'s stack frame. It then examines the next method up the invocation stack, in this case the serveCustomer() method of VirtualCafe, to see if it has a catch clause prepared to handle the exception.

Here's the code for VirtualCafe:

// In Source Packet in file except/ex7/VirtualCafe.java

class VirtualCafe {

public static void serveCustomer(VirtualPerson cust,

CoffeeCup cup)throws TemperatureException,

UnusualTasteException {

try {

cust.drinkCoffee(cup);

}

catch (TooColdException e) {

System.out.println("This coffee is too cold.");

// Add more hot coffee...

}

}

}

The serveCustomer() method above does indeed surround its invocation of drinkCoffee() with a try block that has an attached catch clause for TooColdException. So the exception stops here. The Java Virtual Machine makes the serveCustomer() method's stack frame current and continues execution at the first statement inside the catch clause. The program prints out:

This coffee is too cold.

If variable i in the drinkCoffee()method above happens to be set to the value of two, the switch statement will instantiate and throw an UnusualTasteException. When this exception is thrown, the Java Virtual Machine will first check the catch clauses of the try block that surrounds the switch statement. In this case, no catch clause matches the thrown exception. The virtual machine will then pop drinkCoffee()'s stack frame and examine the serveCustomer() method. But in serveCustomer(), no catch clause attached to the try block matches the thrown exception either. The virtual machine will therefore pop serveCustomer()'s stack frame and examine the next method up the invocation stack: the main() method of class Example7.

Here's the code for Example7:

// In Source Packet in file except/ex7/Example7.java

class Example7 {

public static void main(String[] args)

throws TemperatureException {

// Create a new coffee cup.

CoffeeCup cup = new CoffeeCup();

// Create and serve a virtual customer.

try {

VirtualPerson cust = new VirtualPerson();

VirtualCafe.serveCustomer(cust, cup);

}

catch (UnusualTasteException e) {

System.out.println("This coffee has an unusual taste.");

}

}

}

This main() method was farsighted enough to surround its invocation of serveCustomer() with a try block that includes a catch clause for UnusualTasteException. Thus, the Java Virtual Machine will make the main() method's stack frame current and will continue execution at the first statement in the catch clause. The program will print:

This coffee has an unusual taste.

In the UnusualTasteException case, both drinkCoffee() and serveCoffee() methods completed abruptly. The Java Virtual Machine popped two frames from the Java stack, stopping its popping only when it reached the main() method.

The last case in this example occurs if the variable i in the drinkCoffee()method gets set to a value greater than two. In this case, the switch statement will instantiate and throw aTemperatureException. When this exception is thrown, the Java Virtual Machine will go through its usual procedure of examining methods for catch clauses and popping frames for methods that can't handle the exception. The virtual machine will examine drinkCoffee(), pop its frame, examine serveCustomer(), pop its frame, examine main(), and pop its frame. At this point, however, the virtual machine has run out of frames. It can't go any further up the method invocation stack because main() was the first method invoked by the thread.

Because none of the methods on the invocation stack is prepared to handle the TemperatureException, the exception is "uncaught." It will be handled by a default handler and result in the death of the thread. Because this thread is the main one of the Example7 application, and the application didn't fire off any other threads that are still running when the main thread dies, the application terminates. (A dead thread doesn't always cause the death of its application, only when a dying thread is the last "non-daemon" thread running inside the application. The details of threads are described in Chapter 17.) In most Java runtime environments, the default handler for an uncaught exception will print out a stack trace when a thread dies. For example, the javaprogram from JDK 1.1.1 prints the following when the main thread of Example7 dies because of an uncaught TemperatureException:

TemperatureException

at VirtualPerson.drinkCoffee(VirtualPerson.java:20)

at VirtualCafe.serveCustomer(VirtualCafe.java:9)

at Example7.main(Example7.java:12)

**The throws Clause**

As you may have guessed from the examples above, the Java language requires that a method declare in a throws clause the exceptions that it may throw. A method's throws clause indicates to client programmers what exceptions they may have to deal with when they invoke the method.

For example, the drinkCoffee() method of class VirtualPerson, shown below, declares three exceptions in its throws clause: TooColdException, TemperatureException, andUnusualTasteException. These are the three exceptions that the method throws but doesn't catch. The method may also throw TooHotException, but this exception doesn't appear in thethrows clause because drinkCoffee() catches and handles it internally. Only exceptions that will cause a method to complete abruptly should appear in its throws clause.

// In Source Packet in file except/ex7/VirtualPerson.java

class VirtualPerson {

public void drinkCoffee(CoffeeCup cup) throws TooColdException,

TemperatureException, UnusualTasteException {

try {

int i = (int) (Math.random() \* 4.0);

switch (i) {

case 0:

throw new TooHotException();

case 1:

throw new TooColdException();

case 2:

throw new UnusualTasteException();

default:

throw new TemperatureException();

}

}

catch (TooHotException e) {

System.out.println("This coffee is too hot.");

// Customer will wait until it cools to an

// acceptable temperature.

}

}

//...

}

In the drinkCoffee() method above, each exception declared in the throws clause is explicitly thrown by the method via a throw statement. This is one of two ways a method can complete abruptly. The other way is by invoking another method that completes abruptly.

An example of this is VirtualCafe's serveCustomer() method, shown below, that invokes VirtualPerson's drinkCoffee() method. The serveCustomer() method contains no throw statements, but it does declare two exceptions in its throws clause: TemperatureException and UnusualTasteException. These are two of three exceptions that may be thrown bydrinkCoffee(), which serveCustomer() invokes. The third exception, TooColdException, doesn't appear in the throws clause because serveCustomer() catches and handles it internally. Only those exceptions that will cause the serveCustomer() method to complete abruptly appear in its throws clause.

// In Source Packet in file except/ex7/VirtualCafe.java

class VirtualCafe {

public static void serveCustomer(VirtualPerson cust,

CoffeeCup cup)throws TemperatureException,

UnusualTasteException {

try {

cust.drinkCoffee(cup);

}

catch (TooColdException e) {

System.out.println("This coffee is too cold.");

// Add more hot coffee...

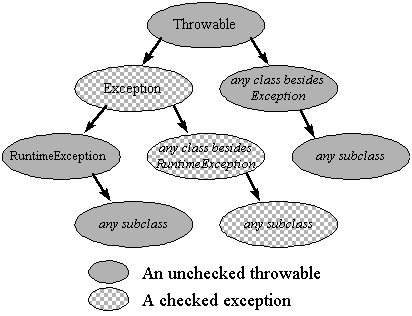
}

}

}

Although a throws clause lists exceptions that may cause a method to complete abruptly, the list is not necessarily complete. Not everything that can be thrown by a method need be put in athrows clause.

There are two kinds of exceptions in Java, *checked* and *unchecked*, and only checked exceptions need appear in throws clauses. Whether or not an exception is "checked" is determined by its position in the hierarchy of throwable classes. Figure 9-4 shows that some parts of the Throwable family tree contain checked exceptions while other parts contain unchecked exceptions. To create a new checked exception, you simply extend another checked exception. All throwables that are subclasses of Exception, but not subclasses of RuntimeException, are checked exceptions.

  
Figure 9-4. Checked and unchecked throwables.

The conceptual difference between checked and unchecked exceptions is that checked exceptions signal abnormal conditions that you want client programmers to deal with. For instance, because the drinkCoffee() method allocates memory with the new operator, it could potentially complete abruptly by throwing an OutOfMemoryError. This is not a checked exception, because it's not a subclass of Exception. It's a subclass of Error. Conceptually, OutOfMemoryError isn't a checked exception because you don't want client programmers to have to deal directly with the fact that drinkCoffee() could complete abruptly because of low memory.

When you place an exception in a throws clause, it *forces* client programmers who invoke your method to deal with the exception, either by catching it or declaring it in their own throwsclause. If they don't, their classes won't compile. For example, because the drinkCoffee() method declares three exceptions in its throws clause, the serveCustomer() method, which invokes drinkCoffee(), has to deal with those three exceptions. In this case, serveCustomer()catches one exception, TooColdException, but not the other two. If serveCustomer()hadn't declared in its throws clause the other two exceptions, TemperatureException and UnusualTasteException, the VirtualCafe class would not have compiled.

The general rule is: any checked exceptions that may be thrown in a method must either be caught or declared in the method's throws clause. Checked exceptions are called "checked" because both the Java compiler and the Java Virtual Machine check to make sure this rule is obeyed.

Most unchecked throwables declared in java.lang (subclasses of Error and RuntimeException) are problems that would be detected by the Java Virtual Machine. Errors usually signal abnormal conditions that you wouldn't want a program to handle. Problems with linking, such as NoClassDefFoundError, or memory, such as StackOverflowError, could happen just about anywhere in a program. In the rare cases in which they happen, it is usually reasonable that the thread terminate.

Although most runtime exceptions (members of the RuntimeException family) are also thrown by the Java Virtual Machine itself, they are usually more an indication of software bugs. Problems with arrays, such as ArrayIndexOutOfBoundsException, or passed parameters, such as IllegalArgumentException, could also happen just about anywhere in a program. When exceptions like these are thrown, you'll want to fix the bugs that caused them to be thrown. You won't, however, want to force client programmers to wrap every invocation of a method that uses arrays with a catch clause for ArrayIndexOutOfBoundsException.

You can throw and catch unchecked exceptions just like checked exceptions; however, the Java Language Specification advises against throwing errors. It is intended that errors be thrown only by the Java runtime. You may, however, reasonably throw runtime exceptions. You can throw a runtime exception declared in java.lang, or declare your own subclasses ofRuntimeException.

To decide whether to throw a checked exception or an unchecked runtime exception, you must look at the abnormal condition you are signaling. If you are throwing an exception to indicate an improper use of your class, you are signaling a software bug. The class of exception you throw should probably descend from RuntimeException, which will make it unchecked. Otherwise, if you are throwing an exception to indicate not a software bug, but an abnormal condition that client programmers should deal with every time they use your method, your exception should be checked.

**The finally Clause**

Once a Java Virtual Machine has begun to execute a block of code--the statements between two matching curly braces--it can exit that block in any of several ways. It could, for example, simply execute past the closing curly brace. It could encounter a break, continue, or return statement that causes it to jump out of the block from somewhere in the middle. Or, if an exception is thrown that isn't caught inside the block, it could exit the block while searching for a catch clause.

Given that a block can be exited in many ways, it is important to be able to ensure that something happens upon exiting a block, no matter how the block is exited. For example, if you open a file in a method, you may want to ensure the file gets closed no matter how the method completes. In Java, you express such a desire with a finally clause.To use a finally clause, you simply:

* enclose in a try block the code that has multiple exit points, and
* put in a finally clause the code that must get executed when the try block is exited.

Here's an example:

try {

// Block of code with multiple exit points

}

finally {

// Block of code that must always be executed when the try block

// is exited, no matter how the try block is exited

}

At least one clause, either catch or finally, must be associated with each try block. If you have both catch clauses and a finally clause with the same try block, you must put the finally clause after all the catch clauses, as in:

// In Source Packet in file except/ex8/VirtualPerson.java

class VirtualPerson {

public void drinkCoffee(CoffeeCup cup) {

try {

int i = (int) (Math.random() \* 4.0);

switch (i) {

case 0:

throw new TooHotException();

case 1:

throw new TooColdException();

case 2:

throw new UnusualTasteException();

default:

System.out.println("This coffee is great!");

}

}

catch (TooHotException e) {

System.out.println("This coffee is too hot.");

}

catch (TooColdException e) {

System.out.println("This coffee is too cold.");

}

catch (UnusualTasteException e) {

System.out.println("This coffee is too strong.");

}

finally {

System.out.println("Can I please have another cup?");

}

}

//...

}

If during execution of the code within a try block, an exception is thrown that is handled by a catch clause associated with the try block, the finally clause will be executed after the catch clause. For example, if a TooColdException exception is thrown during execution of the try block above, the program would print the following:

This coffee is too cold.

Can I please have another cup?

If an exception is thrown that is *not* handled by a catch clause associated with the try block, the finally clause is still executed. The Java Virtual Machine will execute the code of the finally clause before it continues searching elsewhere for an appropriate catch clause. There is no way to leave a try block without executing the code of its finally clause.

You can do anything inside a finally clause that you can do elsewhere, including executing break, continue, or return statements, or throwing exceptions. Such actions inside a finally clause, however, can have some surprising effects. For example, consider a finally clause that is entered because of an uncaught exception. If the finally clause executes a return, the method would complete normally via the return, not abruptly by throwing the exception. The exception would have in effect been handled by the finally clause instead of a catch clause.

As another example, consider a finally clause that is entered because a "return true;" statement was executed inside the try block. If the finally clause executes a "return false;" statement, the method will return false. For more detailed information about why finally clauses behave as they do, see Chapter 25.

**The Meaning of "Abnormal Condition"**

As described above, if your method can encounter an abnormal condition that you feel the caller of your method should deal with, you should throw a checked exception. This will force any client programmer to deal with the abnormal condition, either by catching it or declaring it in a throws clause. Deciding whether or not a condition is "abnormal," however, is a subjective process. The general rule of thumb is: avoid using exceptions to indicate conditions that can reasonably be expected as part of the normal functioning of the method.

As an illustration, consider the FileInputStream and DataInputStream classes from the java.io package. An application that uses FileInputStream to print the text of a file to the standard output is shown below:

// In Source Packet in file except/ex9/Example9a.java

import java.io.\*;

class Example9a {

public static void main(String[] args)

throws IOException {

if (args.length == 0) {

System.out.println("Must give filename as first arg.");

return;

}

FileInputStream in;

try {

in = new FileInputStream(args[0]);

}

catch (FileNotFoundException e) {

System.out.println("Can't find file: " + args[0]);

return;

}

int ch;

while ((ch = in.read()) != -1) {

System.out.print((char) ch);

}

System.out.println();

in.close();

}

}

This example shows that the read() method of FileInputStream reports an "end of file has been reached" condition not by throwing an exception, but by returning a special value: -1. In this method, reaching end of file is considered a normal part of using the method. It is not considered an "abnormal" condition. The normal way to read bytes is to keep on reading them until you hit the end.

The DataInputStream class, on the other hand, takes a different approach when reporting end of file:

// In Source Packet in file except/ex9b/Example9b.java

import java.io.\*;

class Example9b {

public static void main(String[] args)

throws IOException {

if (args.length == 0) {

System.out.println("Must give filename as first arg.");

return;

}

FileInputStream fin;

try {

fin = new FileInputStream(args[0]);

}

catch (FileNotFoundException e) {

System.out.println("Can't find file: " + args[0]);

return;

}

DataInputStream din = new DataInputStream(fin);

try {

int i;

for (;;) {

i = din.readInt();

System.out.println(i);

}

}

catch (EOFException e) {

}

fin.close();

}

}

Each time the readInt() method of DataInputStream is invoked it reads four bytes from the stream and interprets them as an int. When readInt() encounters end of file, it throwsEOFException. There are two reasons that throwing an exception is a reasonable approach for this method. First, readInt() can't return a special value to indicate end of file, because all possible return values are valid ints. (It can't return -1 on end of file, for example, because it may read a -1 from the stream and need to return it as a valid int value.) Second, if readInt()encounters end of file after reading only one, two, or three bytes, that probably qualifies as an "abnormal condition." The method is supposed to read four bytes, but only one to three are available. Given that this exception is an integral part of using this class, it is a checked exception (a subclass of Exception). Client programmers are forced to deal with it.

A third approach to signaling an "end has been reached" condition is illustrated by the StringTokenizer and Stack classes in the following example:

// In Source Packet in file except/ex9b/Example9c.java

// This program prints the white-space separated tokens of an

// ASCII file in reverse order of their appearance in the file.

import java.io.\*;

import java.util.\*;

class Example9c {

public static void main(String[] args)

throws IOException {

if (args.length == 0) {

System.out.println("Must give filename as first arg.");

return;

}

FileInputStream in = null;

try {

in = new FileInputStream(args[0]);

}

catch (FileNotFoundException e) {

System.out.println("Can't find file: " + args[0]);

return;

}

// Read file into a StringBuffer

StringBuffer buf = new StringBuffer();

try {

int ch;

while ((ch = in.read()) != -1) {

buf.append((char) ch);

}

}

finally {

in.close();

}

// Separate StringBuffer into tokens and

// push each token into a Stack

StringTokenizer tok = new StringTokenizer(buf.toString());

Stack stack = new Stack();

while (tok.hasMoreTokens()) {

stack.push(tok.nextToken());

}

// Print out tokens in reverse order.

while (!stack.empty()) {

System.out.println((String) stack.pop());

}

}

}

This example reads in the bytes of a file, converts them to chars, and places the chars into a StringBuffer. It then uses a StringTokenizer to extract one white-space separated token (aString) at a time and push it onto a Stack. It then pops all tokens from the Stack and prints them out one per line. Because Stack implements a LIFO (Last In First Out) stack, the tokens are printed in reverse order from their appearance in the file.

Both the StringTokenizer and the Stack must signal an "end has been reached" condition. The StringTokenizer constructor takes as a parameter the source String to tokenize. Each invocation of nextToken() returns a String that represents the next token of the source String. Eventually, all the tokens in the source String will be consumed and StringTokenizermust somehow indicate that the end of tokens has been reached. In this case, there is a special return value, null, that could have been used to indicate the end of tokens. But the designer of this class took a different approach. A separate method, hasMoreTokens(), returns a boolean value indicating whether or not the end of tokens has been reached. You must invokehasMoreTokens() each time you invoke nextToken().

This approach shows that the designer did not consider reaching the end of tokens an abnormal condition. It is a normal way to use the class. If you don't check hasMoreTokens() and go ahead and call nextToken() after the end has been reached, however, you will be rewarded with the NoSuchElementException. Although this exception is thrown on an end of tokens condition, it is an unchecked exception (a subclass of RuntimeException). It is thrown more to indicate a software bug--that you are not using the class correctly--than to indicate the end of tokens condition.

Similarly, the Stack class has a method, empty(), that returns a boolean to indicate that the last object has been popped from the stack. You must invoke empty() each time you invokepop(). If you neglect to invoke empty()and invoke pop() on an empty stack, you get an EmptyStackException. Although this exception is thrown when an "end of objects on the stack" condition is encountered, it is another unchecked runtime exception. It is intended to be more an indication of a software bug (the improper use of the Stack class) than the normal way to detect an empty stack.

[MAY WANT TO MENTION ABILITY TO RETHROW AN EXCEPTION FROM WITHIN THE CATCH CLAUSE]

MAY ALSO WANT TO MENTION THAT EXCEPTIONS USUALLY HAVE LONG, DESCRIPTIVE NAMES AND OFTEN HAVE NOTHING ELSE INSIDE OF THEM.]

OVERRIDDEN METHODS CAN ONLY THROW EXCEPTIONS DECLARED IN THE THROWS CLAUSE OF THE SUPERCLASS'S IMPLEMENTATION OF THE METHOD OR SUBCLASSES OF THOSE EXCEPTIONS.

**Extras**

Except in the case of anonymous inner classes, an instance initializer may throw checked exceptions only if the checked exceptions are explicitly declared in the throws clause of every constructor in the class. Instance initializers in anonymous inner classes, on the other hand, can throw any exception.