One more chapter

Handling Exceptions

When bad things happen to good objects

The essentials of Java error handling

One more chapter Exception Handling

Goal

To effectively catch and handle runtime exceptions.

Prerequisites

Basic Java programming skills.

Objectives

At the end of this lesson, you will be able to:

- List several common runtime problems
- Catch an exception
- Distinguish among multiple exceptions types
- Define a new exception class
- Throw an exception

A runtime problem is called an exception

Even though your code compiles, it may not always run smoothly

```
((Widget)array.objectAtIndex(i)).price();
```

- array is null
- index i is out of bounds
- the object at index i is not a Widget
- Widgets don't implement a price method

```
shoppingCart.checkOut();
```

- the shopper's credit card is declined
- the order can't be committed to the database

Such problems *throw* an exception and change the flow of control

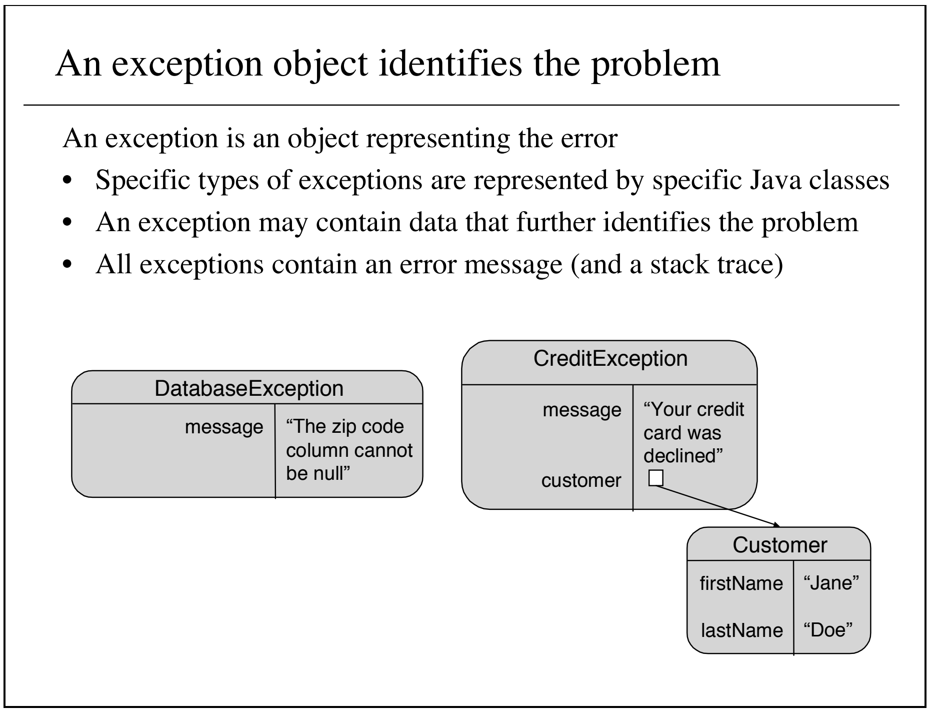
A runtime problem is called an exception

Once your Java code compiles successfully, you can launch your application and run the code. Depending on what happens during the life your application, the code may encounter problems. If your code is well constructed, a runtime problem is not the normal case, but an exceptional one. The Java language and runtime environment define a formal and practical way to deal with exceptions and to create robust production-ready code, you will have to make use of a few important features.

Notice, a runtime problem might be a simple language issue: an object reference is null, an array index is out of bounds, a cast turns out to be a lie, etc. These exceptions usually indicate a fundamental logic or code problem in your implementation.

More realistically, your code is well written, but includes true exceptional cases beyond your control: a customer's credit card is rejected or the database encounters an error when committing a transaction.

In all of these cases, the Java runtime brings the problem to your attention by throwing an exception. By throwing an exception, Java instantly changes the normal flow of control of your code. In the end, the exception requires your attention which you provide through a bit of custom code.



An exception object identifies the problem

An exception is an object representing the error

- Specific types of exceptions are represented by specific Java classes
- An exception may contain data that further identifies the problem
- All exceptions contain an error message (and a stack trace)

DatabaseException
message
"The zip code column cannot be null"
CreditException
message
"Your credit card was declined"
customer
Customer
firstName
"Jane"
lastName
"Doe"

An exception object identifies the problem

The runtime problem will be represented by an object, an instance of a special exception class. Different exceptions—a null pointer, a database failure—are usually represented by instances of different Java classes. The type of exception object begins to explain exactly what happened.

Since each exception type is implemented by a specific, separate Java class, different exceptions can provide different data and behavior specific to a particular runtime problem. A credit exception may include a reference to the customer or credit card that is the focus of the problem.

All exception objects have two things in common: a message string suitable for presenting in your application's user interface, and a Java stack trace, useful for debugging the problem.

What happens when an exception is thrown?

When an exception is thrown

- The code stops executing immediately
- The message sequence “unwinds” until the exception is caught
- An exception object provides the details to the catcher

By default

- The Java/WebObjects infrastructure catches the exception
- Your code that caused the exception loses control

You can explicitly

- Catch the exception to retain control
- Supply custom logic to deal with the problem

What happens when an exception is thrown?

What really happens when an exception is thrown during the life your of application? First, your code stops executing, dead in its tracks. The exceptional condition means that the normal flow of control is somehow impossible. The code that throws the exception manufactures an exception object to record the details of the problem. Finally, the Java runtime “throws” the exception object so that it is “up for grabs” by a special part of your code that “catches” the exception in order to deal with it. In essense, the Java runtime now unwinds backwards through your code until it reaches the first available exception handler.

By default, the Java or WebObjects infrastructure provides the exception handling code. The default behavior typically displays a detailed stack trace showing you where exactly the code was when the exception occurred, and more importantly, how the code got there in the first place. The key fact however is that you lose control of the situation. Your code stops and someone else’s code takes over. Control never returns to you, at least, in the same place that you were when the problem occurred.

You can write your code to explicitly catch the exception yourself. This way, you retain control of the situation, continuing to execute where you have the most information about the context in which the problem has occurred. But now you take on an important burden: you must effectively deal with the problem.

Catching an exception in your own code

To catch an exception, use the **try/catch** keywords

```
try {
  // the code that may throw an exception
  shoppingCart.checkOut();
}
catch (Exception e) {
  // your code to process the actual exception
  System.err.println("Could not check out");
  System.err.println(e.getMessage());
  return; // abort processing
}
// no exception, continue processing
```

Catching an exception in your own code

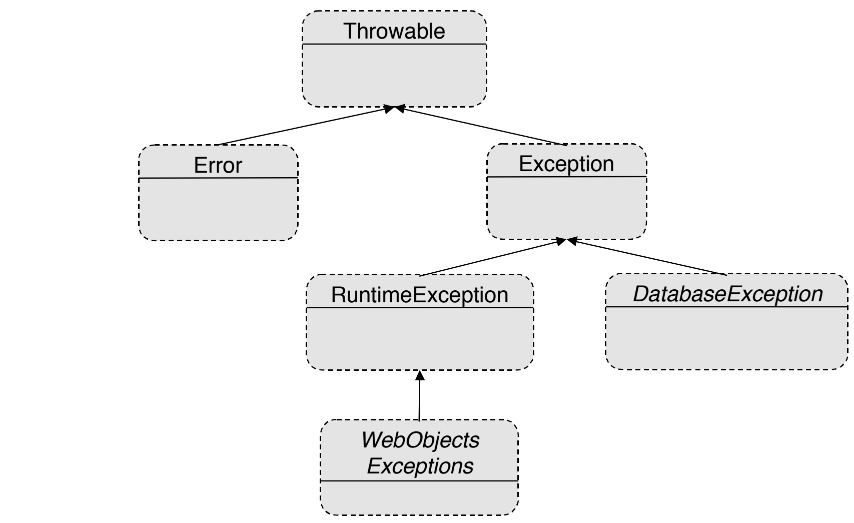
To catch an exception, use the Java keywords *try* and *catch*. First, identify the line or lines of code that may throw an exception. Nest the code inside a try block indicating that you wish to try to do something that may go awry.

Next, supply a block of special code that will be activated only if the code you are trying actually throws an exception. The catch keyword defines the block and takes a single local argument: an exception object. Notice that the exception argument is typed to the specific class of exception you are prepared to handle. To catch just about any exception, you can use the more generic type, java.lang.Exception. The exception argument is in scope only for the duration of the catch block that defines it. In this sense, the catch block is much like a local "in line" method.

What you do inside the catch block depends on the logic of your application. You know the class of exception object from the catch argument. The exception object at minimum contains a message string (and a stack trace). No matter what, your catch code usually takes an alternate path such as returning from the method or avoiding additional processing that is carried out in the normal case.

If the code in the try block executes successfully without an exception, the catch block is ignored and the flow of control resumes with the next Java statement flowing the exception handler.

There are multiple exception classes

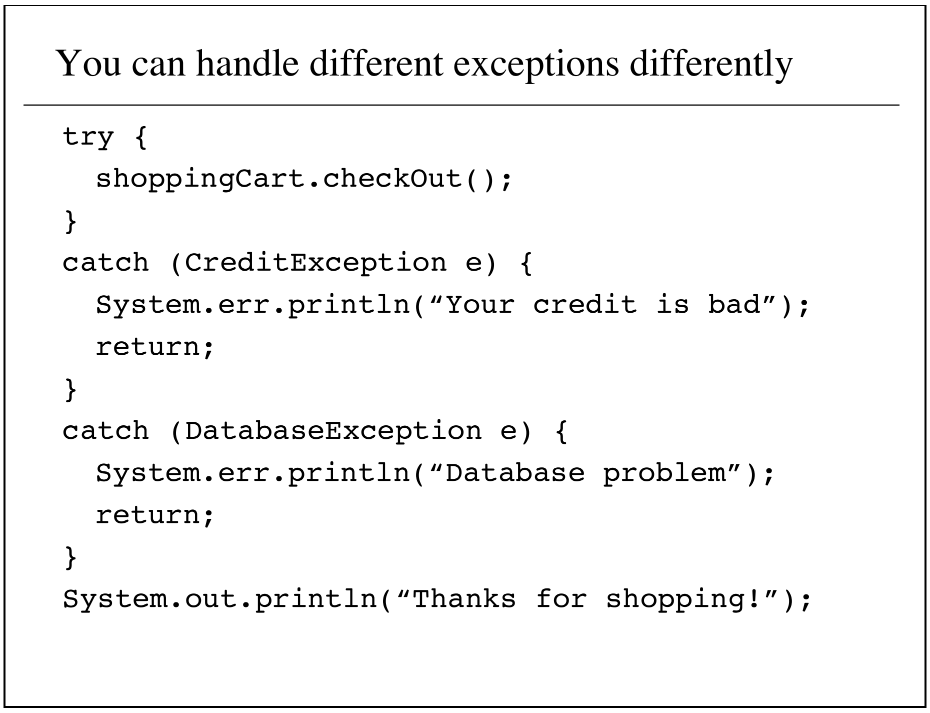


There are multiple exception classes

Each exceptional case in the Java runtime or your custom code can be represented by a specific exception class. Java defines several basic exception types in an inheritance hierarchy rooted at the abstract class java.lang.Throwable. Serious errors are subclasses of java.lang.Error and often cannot be caught at all. Catchable exceptions are subclasses of Exception. Exceptions defined by the WebObjects frameworks are usually subclasses of java.lang.RuntimeException. These have a special property as you will see.

Most other exception classes—defined by 3rd-party packages or new custom classes that you might define—are subclasses of Exception. Our fictional DatabaseException is a good example.

The key point is that specific exception classes can be defined to represent specific runtime problems. You can implement different code for different classes of exceptions. Multiple exception types can be classified into sub-groups that inherit common behavior from a superclass. This enables you to be specific—here is a DatabaseException—or more general—here is some kind of generic runtime problem.



You can handle different exceptions differently

```
try {
  shoppingCart.checkOut();
}
catch (CreditException e) {
  System.err.println("Your credit is bad");
  return;
}
catch (DatabaseException e) {
  System.err.println("Database problem");
  return;
}
System.out.println("Thanks for shopping!");
```

You can handle different exceptions differently

To handle different classes of exceptions differently, you can supply multiple catch blocks in your exception handler. Each catch block declares the specific class of exception it is prepared to deal with. When the try block throws an exception, Java will select the most specific catch block it can find. It will execute at most only one try block. If the exception is not covered by any of your try blocks—the actual exception class is not within any of the sub-groups your declared—it will be handled by the default exception handler from Java or WebObjects.

This makes sense: provide the code for those specific exception cases you can handle, and ignore the others.

Exception or not, some code is always necessary

Often, some code is necessary regardless of what happens

Use the **finally** clause for code that always executes

```
try {
  shoppingCart.checkOut();
}
catch (CreditException e) {
  System.err.println(“Your credit is bad”);
  return;
}
finally {
  shoppingCart.setProcessingComplete(true);
}
```

Exception or not, some code is always necessary

The code within the try block might well succeed without throwing an exception. On the other hand, only one of many different catch blocks will be activated if there is an exception. Often, your logic requires some final processing in all cases—regardless of whether or not there was an exception or regardless of the actual exception type that was thrown.

To implement this requirement, you can provide a finally block. The finally block takes no arguments (it is independent of any exception) and will be activated no matter what happens in the try or catch blocks. It is always the last bit of code to execute before the flow of control resumes.

Exception classes are often inner classes

Exceptions are often specific to a class or interface

For example, credit exceptions might only occur with shopping carts

CreditException could be defined *within* the ShoppingCart class

You refer to inner classes using a dot separated path name, like

```
ShoppingCart.CreditException e = new
  ShoppingCart.CreditException();
```

or

```
catch (ShoppingCart.CreditException("Bad")) {
```

Exception classes are often inner classes

Java includes a feature for defining inner classes, that is, a class definition inside of another class or interface. This provides a useful encapsulation feature. Consider that a credit exception might only be thrown when interacting with a shopping cart object. In this case, it may make sense to define the CreditException class within the ShoppingCart class. It is up to the class implementor.

To use an inner class, you must include the outer class (or interface) at part of the formal type name. Much like a package name, the inner class name is specified by a dot-separated path that moves from outer to inner class.

You can define your own new exception classes

Custom exceptions typically subclass java.lang.Exception

You can add new instance data and behavior

```
class CreditException extends Exception {
  private Customer customer;
  public Customer getCustomer() {
     return customer;
  }
  public setCustomer(Customer customer) {
     this.customer = customer;
  }
}
```

You can define your own new exception classes

What if your own custom code detects a runtime problem, an exception to the normal, successful flow? You may wish to define your own custom exception classes that you can throw.

Most custom exception classes are subclasses of java.lang.Exception. You can define a new subclass and, like any Java subclass, you can add optional data and behavior. Notice, however, that often just the new class type alone may be sufficient for a catch block to be specific about your particular exception.

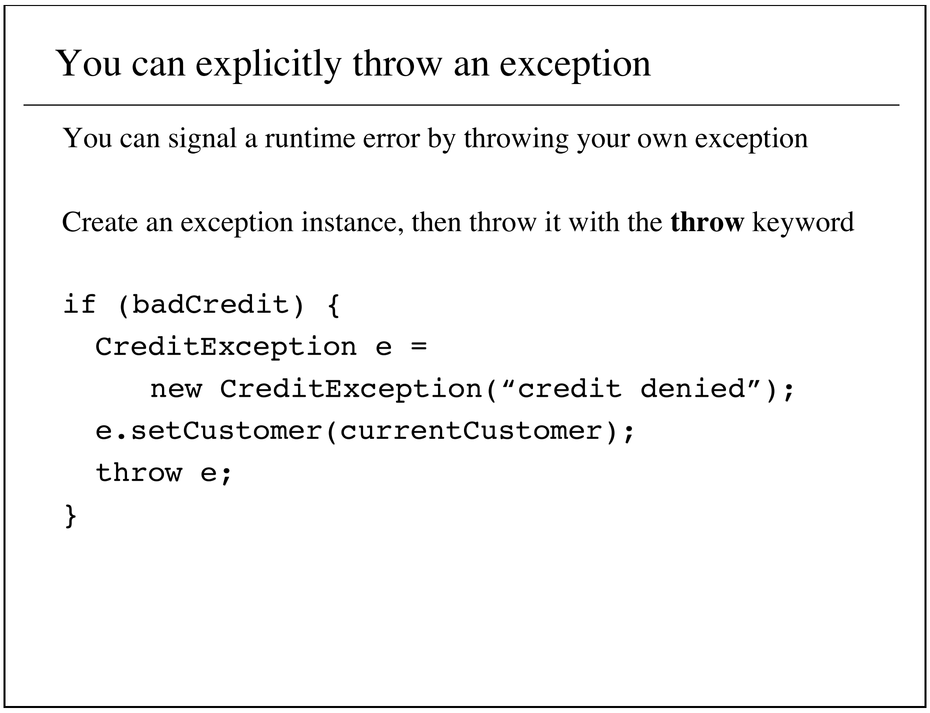
A simple example of custom behavior might be the ability of a credit exception to reference the customer to which it applies. In the CreditException, you might add an instance variable and a pair of accessor methods.

Note that java.lang.Exception defines a one-argument constructor for creating a new exception object along with the message string. To provide the same one-argument constructor in your CreditException subclass, you must include a one-argument constructor such as:

```
public CreditException(String message) {

   super(message);

}
```



You can explicitly throw an exception

You can signal a runtime error by throwing your own exception

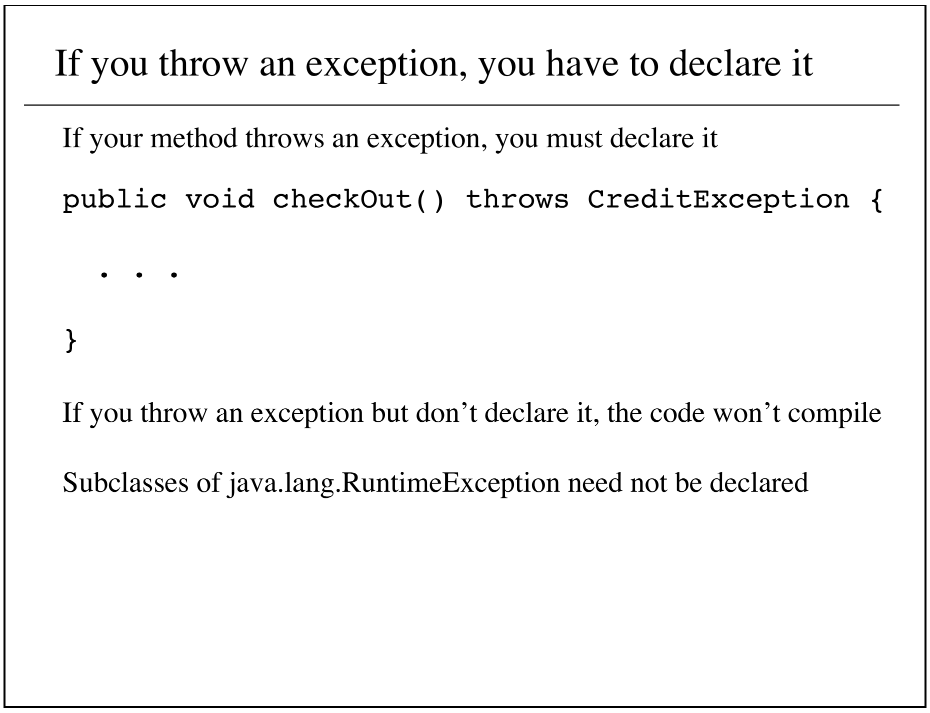
Create an exception instance, then throw it with the **throw** keyword

```
if (badCredit) {
  CreditException e =
      new CreditException(“credit denied”);
  e.setCustomer(currentCustomer);
  throw e;
}
```

You can explicitly throw an exception

When your custom code detects that a specific runtime problem has occurred, it can throw an exception thereby aborting the flow of control and making an exception object available to whichever catch block the Java runtime finds to handle it.

To throw an exception, create a new instance of the specific exception class that is most appropriate, typically a custom exception class you have defined. Initialize the exception object in any additional way the situation requires—in our example, setting the customer reference in the exception. Finally, throw the exception using the Java keyword *throw.*



If you throw an exception, you have to declare it

If your method throws an exception, you must declare it

```
public void checkOut() throws CreditException {

  . . .

}
```

If you throw an exception but don't declare it, the code won't compile

Subclasses of java.lang.RuntimeException need not be declared

If you throw an exception, you have to declare it

Java features *checked exceptions.* That is, if a Java method may possibly throw an exception, it must formally declare it in the method signature. This way, the Java complier can warn you if you call a method but do not make provisions for catching the exceptions that might result.

Use the keyword throws in your method declaration. If you don't but the body of the method contains a throw statement, your code will not compile. If you throw multiple types of exceptions, specify each one in a comma-separated list.

Subclasses of java.lang.RuntimeException need not be declared. You can throw them without advertising it your method declaration. This is usually not a good idea because it withholds important information from the client of your code. You are discouraged from using RuntimeException classes for this reason.