

Exception Handling

Chapter 11

Exceptions

- An *exception* is an object that describes and unusual, erroneous, or unexpected situation
 - An event that disrupts the normal flow of the program
- You can think about your program as having two possibly executions:
 1. Normal execution flow
 2. Exception execution flow

Exceptions (cont.)

- When an exception occurs within a method, the method creates an object and hands it to the runtime system
 - This object is the *exception object* and it contains information:
 - Type of the error
 - An error message
 - State of the program when the error occurred
- Creating an exception object is called *throwing an exception*

Exceptions (cont.)

- After an exception is thrown, the runtime system tries to find some way to handle it
 - The runtime system looks through the methods that have been called to get to where you are now, known as the *call stack*
- The runtime system searches the call stack for some method that contains a way to deal with the exception
 - This block of code is called the *exception handler*

Exceptions (cont.)

- When an appropriate handler is found (one that matches the type of the exception), the runtime system passes the exception to the handler
 - This is called *catching the exception*
- If there is no method that has an appropriate handler, the program will terminate (crash)
 - You want to avoid this!

Exception Handling

- Java has a predefined set of exceptions and errors that describe what might occur during program execution
- You can deal with an exception in three ways:
 - Ignore it
 - Handle it where it occurs (catch it)
 - Handle it somewhere else (throw and catch it)
- How you choose to handle exceptions is an important design consideration

Ignoring Exceptions

- If you ignore an exception, your program will terminate abnormally and produce a message describing what happened
- This message is called the *call stack trace* and it:
 - Indicates the line in the program on which the exception occurred
 - Shows the method call trail that lead to the attempted execution of this line
- If you know an exception might occur, you probably should not ignore it.

Practice

- Review a program that reads in two numbers from the user and divides them.
 - Ignore the possible exception that could be thrown.

CATCHING EXCEPTIONS

The `try-catch` Statement

- To handle an exception, the line that throws the exception must be executed within a *try block*
- A `try` block is usually followed by one or more *catch clauses**
 - Each `catch` clause has an associated exception type and is called an *exception handler*
- When an exception occurs, processing continues at the first catch clause that matches the exception type then jumps outside of the complete `try-catch` block

*It is allowed to have a try-finally with no catch.

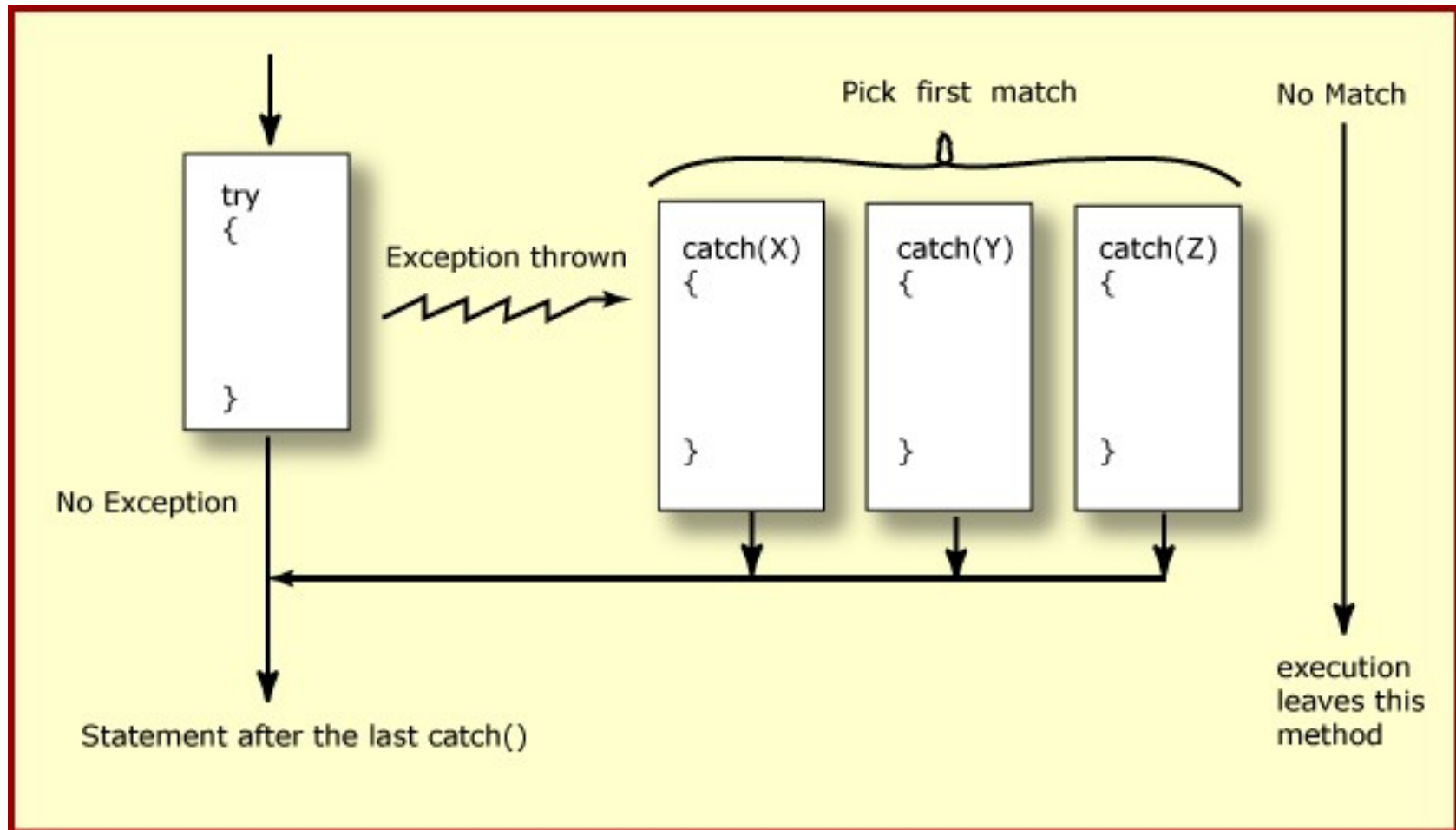
The try-catch Statement (cont.)

... other code

```
try {  
    // code to execute with the normal flow  
    // this code might throw an exception  
} catch (ExceptionA ex) {  
    // code to handle exceptions of type ExceptionA  
} catch (ExceptionB ex) {  
    // code to handle exceptions of type ExceptionB  
}
```

... other code

Flow of Control with try-catch



Practice

- Write a program to process codes entered by the user.
 - Codes contain a zip code and a shipping code (S for standard, E for express)
 - Example: 94117S, 94122E
 - Use exception handling to account for bad input.
 - Count the number of valid and invalid entries.

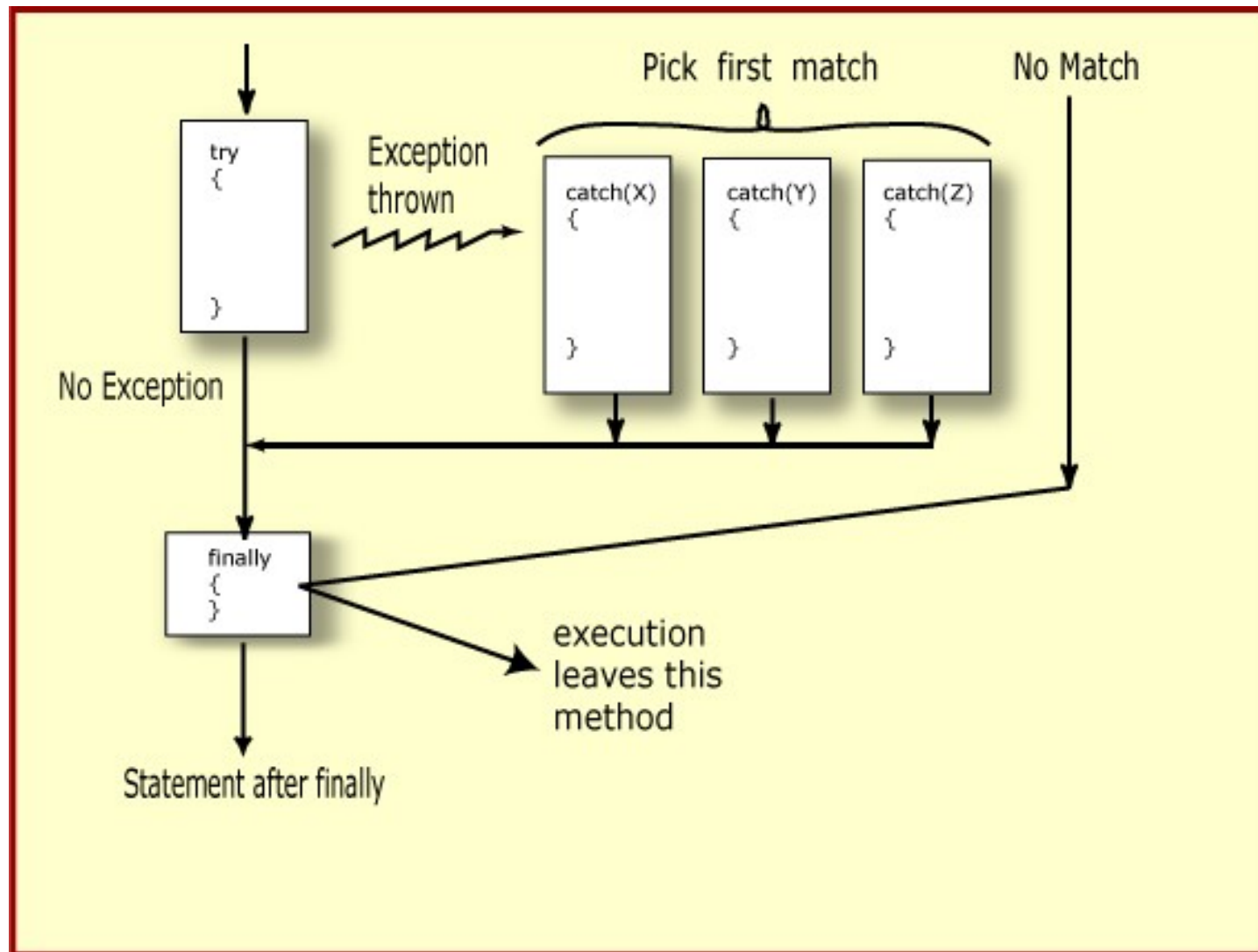
The `finally` Clause

- A `try` statement can have an optional clause following the `catch` clauses, designated by the reserved word `finally`
- The statements in the `finally` clause are *always* executed
 - Example: closing an input/output stream

The `finally` Clause (cont.)

- If no exception is generated:
 - The statements in the `finally` clause are executed after the statements in the `try` block
- If an exception is generated and caught:
 - The statements in the `finally` clause are executed after the statements in the appropriate `catch` clause
- If an exception is generated and propagated:
 - The statements in the `finally` clause are executed before control is handed over to the invoking method

Flow of Control with the `finally` Clause



Practice

- Modify the previous program to print out the number of valid and invalid codes entered each pass through the loop.

PROPAGATING EXCEPTIONS

Exception Propagation

- An exception can be handled at a higher level if it is not appropriate to handle it where it occurs
- Exceptions *propagate* up through the method calling hierarchy until they are caught and handled or until they reach the `main` method
- A `try` block that contains a call to a method in which an exception is thrown can be used to catch that exception

Practice

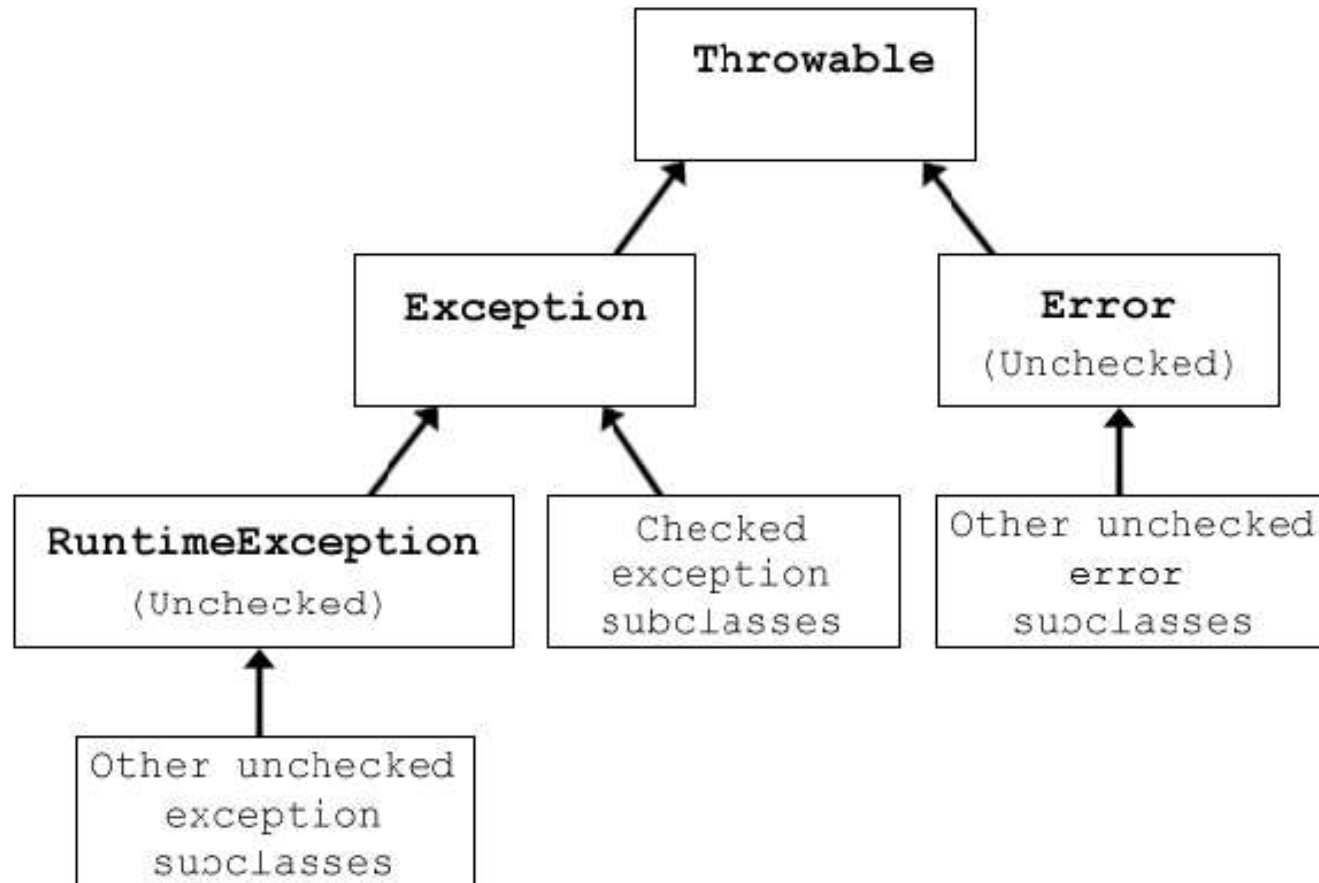
- Review the propagation example.

EXCEPTION CLASS HIERARCHY

The Exception Class Hierarchy

- Classes that define exceptions are related by inheritance, forming an exception class hierarchy
- All error and exception classes are descendents of the `Throwable` class

The Exception Class Hierarchy (cont.)



Errors

- Errors are similar to exceptions.
- Errors represent a serious problem from which your program cannot recover.
- You do *not* need to handle (catch) or plan for errors.
- Errors usually result from external conditions.

Exceptions

- Exceptions come in two types: checked and unchecked.
- Checked exceptions have to be dealt with.
 - The compiler *checks* to make sure you have addressed these possible conditions.
- Unchecked exceptions do not have to be explicitly handled.

Checked Exceptions

- A checked exception must be either:
 - Caught by the method (in a `try-catch` block)
 - Propagated (listed in the `throws` clause of the method)
- The compiler will issue an error if a checked exception is not caught or asserted in a `throws` clause
- Checked exceptions are conditions that a program *should* anticipate and recover from
 - Example: a nonexistent file

Unchecked Exceptions

- The compiler does not require explicit handling of unchecked exceptions.
 - Example: an array that goes out of bounds, divide by zero
 - You do not have to use a `try-catch` or `throws` clause every time you create an array or divide two numbers!
 - That is because these exceptions are *unchecked*.
- Unchecked exceptions are of type `RuntimeException` or any of its descendants

Unchecked Exceptions (cont.)

- Unchecked exceptions often result from logical errors in your code (an internal condition)
 - Example: lack of an if-statement to make sure the denominator is not zero
 - Example: off-by-one error in array processing
- You can use explicit handling for these types of exceptions, but it's often better to write code to avoid the errors being generated in the first place.

INPUT/OUTPUT EXCEPTIONS

The IOException Class

- Operations performed by some I/O classes may throw an `IOException`
- For example:
 - A file might not exist
 - A file might exist but the program can't find it
 - A file might not contain the expected type of data
- `IOException`s are *checked* exceptions
 - You must either `catch` them or declare that your `method` `throws` them

Practice

- Write a string of random numbers to a file.
 - First “ignore” the exception.
 - Then modify the code to handle it.
- Modify the two previous I/O programs so they do not ignore the exception.

WRITING YOUR OWN EXCEPTIONS

Exception Hierarchy

- Java provides many classes that describe unexpected situations.
- You might also have unexpected situations in your own programs that you want to handle.
 - Sometimes an if-else structure is best.
 - Sometimes exception handling is best.
 - This is an important design decision. Often, companies will have a standard approach that defines when to use each approach.

Defining Exceptions

- You can define your own exceptions by extending the `Exception` class or any of its descendents

The `throw` Statement

- When you use a Java-defined exception, Java will throw it when needed.
- When you define your own exceptions, *you* have to throw them when the situation arises.
- Exceptions are thrown using the *throw* statement
- Usually a `throw` statement is executed inside an `if`-statement that evaluates a condition to see if the exception should be thrown

Practice

- Create an exception class using the previous example to represent an invalid shipping status.

EXCEPTIONS IN JAVA 7

Old Method


```
File f = null;
try {
    f = new File(...);
    // use the file
} catch (IOException ex) {
    ex.printStackTrace();
} finally {
    if (f != null)
        f.close();
}
```

Must declare outside
of the try/catch/finally
block so it can be
seen inside the
finally statement

Must make sure not
null and then always
close- sometimes
this line can also
generate an
exception- nested
blocks- yikes!

Try-with-Resources

```
try (File f = new File(...) ) {  
    f = new File(...);  
    // use the file  
} catch (IOException ex) {  
    ex.printStackTrace();  
} finally {  
    // no need to close!}
```



Declare and initialize
resources inside
of () after try

Try-with-Resources

- No matter how the try block exits (with or without exception, with or without finding a matching catch block), the resource is closed.
- You can declare multiple resources inside of the try. (Separate with ;)
- You can only declare objects that implements the `AutoCloseable` interface.
 - This will essentially be any of your standard IO objects. But you can always check the API to be sure!

Handling Multiple Exceptions

- Often you have the same code inside of multiple catch blocks.
 - Example: `ex.printStackTrace();`
- You can now have a catch block catch multiple types of exceptions.
- Use this only if they will all be handled the same way.
- If you are taking different actions for different kinds of exceptions, use the old approach of multiple catch blocks.

Handling Multiple Exceptions

```
try {  
  
    // code to execute with the normal flow  
    // this code might throw an exception  
  
} catch (ExceptionA | ExceptionB ex) {  
  
    // code to handle exceptions of type  
    // ExceptionA or Exception B  
  
} catch (ExceptionC ex) {  
    // code to handle exceptions of type  
    // ExceptionC  
}
```

SUMMING UP

Bottom Line

1. When an exception is thrown, search for the first matching catch block in the current method.
 - A. If you find a match in the current method:
 - i. execute catch
 - ii. execute finally
 - iii. execute first line after the try-catch block (never go back up into the try!)
 - B. If you do not find a match in the current method:
 - A. execute finally
 - B. leave the method and return to the “paused” line in the previous invoking method
 - C. repeat Step 1 starting from that line

Bottom Line

- Java's Exceptions
 - Checked
 - **Must** be handled in order to compile
 - Either a) catch or b) propagate using “throws” in the method header
 - Unchecked
 - Design code to avoid these (e.g., NullPointerException, ArrayIndexOutOfBoundsException)
 - Use them to your advantage (e.g., reading in numeric data)
 - *Can* catch or propagate to handle
- Your Exceptions
 - Use to represent your own error conditions
 - Write the Exception class, throw the exception when the condition occurs, catch and handle the exception or propagate with throws

Tips for Using Exceptions

1. Don't replace simple conditionals with exceptions.
 - If a list is empty, if a number is negative, etc.
2. Wrap full tasks in try-blocks, not individual statements.
3. Throw and catch specific exceptions.
 - Do not catch(Exception ex)!! This is bad design.
4. Handle exceptions in the appropriate place.

Exceptions: Why do we bother?

- There are several advantages to using exceptions.
 1. It separates the error-handling code from the normal execution code.
 - The try block contains all the code we expect to run if everything goes smoothly.
 2. It allows us to propagate errors up the call stack so only the methods that care about the errors have to worry about them.
 3. It allows us to group and categorize our errors.