Introduction to Exceptions in Java
Runtime Errors

• What are syntax errors?
• What are runtime errors?
• Java differentiates between runtime errors and exceptions
  – Exceptions can be handled in some way
    • Example: division by zero
  – Errors are unrecoverable situations
    • Example: running out of memory
Exceptions

• *Exception*: an abnormal or erroneous situation at runtime

• Examples:
  • Division by zero
  • Array index out of bounds
  • Illegal input number format
  • Following a null reference
Exceptions

• These erroneous situations *throw an exception*
• Exceptions can be thrown by the *runtime environment* or by the *program* itself
Throwing Exceptions

• You have seen a situation where a program throws an exception
• Example in ArrayStack.java:

```java
public T pop() throws EmptyCollectionException {
    if (isEmpty())
        throw new EmptyCollectionException("Stack");
...
}
```
Java Exceptions

• In Java, an exception is an object
• There are predefined exception classes in the Java API
  • Exception
    • Its subclass RuntimeException
      • Its subclass NullPointerException
  etc.
Java Exceptions

• Examples of Java predefined exception classes (types):
  
  ArithmeticException
  IndexOutOfBoundsException
  IOException
  NullPointerException
Some Java Error and Exception Classes

FIGURE 2.7 Part of the Error and Exception class hierarchy
Java Exceptions

• We can throw an exception object of one of these predefined types, or we can define our own exception classes

• How? We can extend Exception or one of its subclasses
public class EmptyCollectionException extends RuntimeException {
{
/**
 * Sets up this exception with an appropriate message.
 * @param collection String representing name of collection
 */

public EmptyCollectionException (String collection) {
    super ("The " + collection + " is empty.");
}
}
Uncaought Exceptions

• If an exception is not handled at all by the program, a standard error message is printed by the Java runtime system and the program is terminated

• Example:

java.lang.ArrayIndexOutOfBoundsException: 5 at ExceptionExample1.main(ExceptionExample1.java:17)
Exception in thread "main"
Catching Exceptions

• To *handle an exception* in your program:
  – A method, or the runtime environment, that detected an error during execution *throws* an exception
  – The code that deals with the exception is said to *catch* or *handle* it
How to Catch an Exception

• Syntax of try-catch statement:

  try
  {
    // try block: statements(s) that might cause
    // an exception to be thrown
  }
  catch (possible-exception-type e)
  {
    // catch clause: statements to handle the
    // problem, referring to the exception object e
  }
Try block

You can either

• put each line of code that might throw an exception within its own try block and provide separate exception handlers for each, or

• put all the code within a single try block and associate multiple handlers with it.
Catching Exceptions: Example

```java
public static void main (String[] args) {

    BufferedReader keyboard = new BufferedReader( 
    (new InputStreamReader(System.in), 1);

    System.out.print("Enter an integer:");

    String userTyped = keyboard.readLine();

    //continued
```
Catching Exceptions: Example

try {
    int value = Integer.parseInt(userTyped);
} catch (NumberFormatException e) {
    System.out.println(userTyped + " is not an integer!");
    System.out.println(e.getMessage());
}
Catching Exceptions

• How try-catch works:
  – When the try-catch statement is executed, the statements in the try block are executed
  – If no exception is thrown:
    • Processing continues with the statement following the try-catch statement
  – If an exception is thrown:
    • Control is immediately passed to the first catch clause whose specified exception corresponds to the class of the exception that was thrown
Catching Exceptions: Example

Example: try to create a file (in a non-existent directory)

```java
String filename = "/nosuchdir/myfilename";

try {
    new File(filename).createNewFile();
}

catch (IOException e) {
    System.out.println("Unable to create" + filename + ":" + e.getMessage());
}

// execution continues here
```
Catching Exceptions: Example

• Here is the output:

Unable to create /nosuchdir/myfilename: The system cannot find the path specified
Catching Exceptions

• If an exception is *not* caught and handled where it occurs:
  – Control is immediately returned to the *method that invoked the method* that produced the exception
  – If that method does not handle the exception (via a try statement with an appropriate catch clause) then control returns to the method that called it
• This process is called *propagating the exception*
Catching Exceptions

• Exception propagation continues until
  – The exception is caught and handled
  – Or until it is propagated out of the main method, resulting in the termination of the program
A single catch block can handle more than one type of exception.
This can reduce code duplication and lessen the temptation to catch an overly broad exception.
In the catch clause, specify the types of exceptions that the block can handle and separate each exception type with a vertical bar (|).
A Try-Catch-Finally Example

The try-catch-finally syntax:

```
try {
    code
}
catch(exception1) {statements}
catch(exception2) {statements}
catch(exception3|exception4){statements}
...
finally {statements}
```
Try example

As an example [see http://docs.oracle.com/javase/tutorial/essential/exceptions/try.html], consider exception handling for a `writeList` method using a `ListOfNumbers` class. We enclose the exception-throwing statements of the `writeList` method within a try block.
Try example, continued

```java
private List<Integer> list;
private static final int SIZE = 10;
PrintWriter out = null;
try {
    System.out.println("Entered try statement");
    out = new PrintWriter(new FileWriter("OutFile.txt"));
    for (int i = 0; i < SIZE; i++)
        System.out.println("Value at: "+i+" = "+list.get(i));
}
```
Try example continued

If an exception occurs within the try block, that exception is handled by an exception handler associated with it. To associate an exception handler with a try block, you must put a catch block after it.
Catch Block

Two exception handlers for the writeList method: one for the two checked exceptions that can be thrown within the try statement:

catch(FileNotFoundException e) {
    System.err.println("FileNotFoundException: " +
    e.getMessage());
    // Ask for new filename
    throw new SampleException(e); }
catch(IOException e) {
    System.err.println("Caught IOException: " +
    e.getMessage());
}
Catch Block

Both handlers print error messages. The first handler throws a user-defined exception. You might want to do this if you want your program to handle an exception in this situation in a specific way (exception handlers can do more than just print error messages or halt the program).
Finally Block

The finally block always executes when the try block exits. This ensures that the finally block is executed even if an unexpected exception occurs. Useful for more than just exception handling: it allows the programmer to avoid having cleanup code accidentally bypassed by a return, continue, or break.

Putting cleanup code in a finally block is always a good practice, even when no exceptions are anticipated.
Finally Block

The try block of the `writeList` method that we have been using opens a `PrintWriter`. The program should close that stream before exiting the `writeList` method. `writeList`'s try block can exit in one of the following ways:

1. The new FileWriter statement fails and throws an IOException.
2. An FileNotFoundException is thrown
3. The list.get(i) statement fails and throws an ArrayIndexOutOfBoundsException
4. Everything succeeds and the try block exits normally.
Finally Block

The runtime system always executes the statements within the finally block, so it's the perfect place to perform cleanup. The following finally block for the writeList method cleans up and then closes the PrintWriter.

```java
finally {
    if (out != null) {
        System.out.println("Closing PrintWriter");
        out.close(); }
    else
        System.out.println("PrintWriter already closed");
}
```