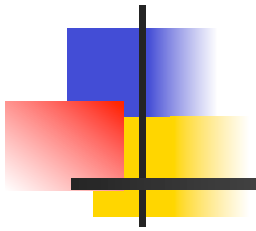
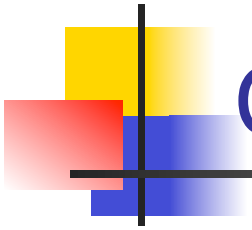


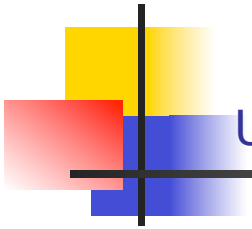
Exception Handling in Java

Rahul Deodhar
rahuldeodhar@gmail.com
www.rahuldeodhar.com
+91 9820213813



Chapter 14 - Exception Handling

- Using `try` and `catch` Blocks to Handle "Dangerous" Method Calls
- `NumberFormatException`
- Line Plot Example
- `try` and `catch` Blocks - More Details
- Two Types of Exceptions - Checked and Unchecked
- Unchecked Exceptions
- Checked Exceptions
- Using API Documentation when Writing Exception-Handling Code
- When a `try` Block Throws Different Types of Exceptions
- The `Exception` Class and its `getMessage` Method
- Multiple `catch` blocks
- Understanding Exception Messages



Using `try` and `catch` Blocks to Handle "Dangerous" Method Calls

- Some API method calls are "dangerous" in that they might possibly lead to a runtime error.
- Example of a "safe" API method call (no runtime error possible):

  `System.out.println(`<*expression*>`)`

- Example of an API method call that might lead to a runtime error:

  `Integer.parseInt(`<*string*>`)`

- Technique for handling such runtime errors:
  - Use *exception handling*. More specifically, surround the method call with a `try` block and insert a `catch` block immediately after the `try` block.

Using `try` and `catch` Blocks to Handle "Dangerous" Method Calls

- Syntax for `try` and `catch` blocks:

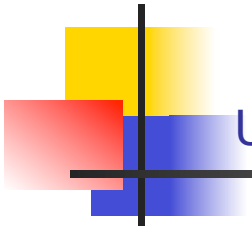
```
try
{
   <statement(s)>
}
catch (<exception-class> <parameter>)
{
   <error-handling-code>
}
```

Normally, one or more of these statements will be a "dangerous" API method call or constructor call.

The exception class should match the type of exception that the `try` block might throw.

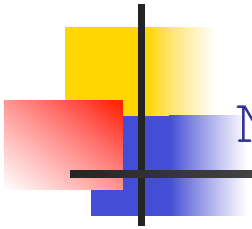
- Example `try` and `catch` code fragment:

```
try
{
   quantity = Integer.parseInt(userEntry);
}
catch (NumberFormatException nfe)
{
   System.out.println("Invalid quantity entered." +
     " Must be a whole number.");
}
```



Using `try` and `catch` Blocks to Handle "Dangerous" Method Calls

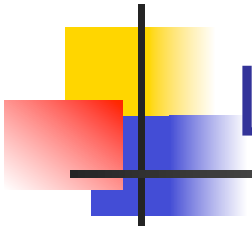
- Semantics for previous slide's `try` and `catch` code fragment:
  - If the `userEntry` string contains all digits, then:
    - The `int` version of `userEntry` is assigned into `quantity`.
    - The JVM skips the `catch` block and continues below it.
  - If the `userEntry` string does not contain all digits, then:
    - The `parseInt` method *throws* a `NumberFormatException` object.
    - The JVM looks for a `catch` block that will catch the thrown exception object; that is, it looks for a matching `catch` block. If it finds one, it executes it and continues below the `catch` block. If there's no matching `catch` block, the program crashes.



NumberFormatException

- The `NumberFormatException` is well named because it's thrown when a number's format is inappropriate.
- More specifically, it's thrown by one of the parse methods (`Integer.parseInt`, `Long.parseLong`, `Double.parseDouble`, etc.) when there's an attempt to convert a string to a number and the string's characters don't form a valid number.
- These code fragments throw `NumberFormatException`s:

```
int numOfPages = Integer.parseInt("962.0");
double height = Double.parseDouble("1.76m");
```



Line Plot Example

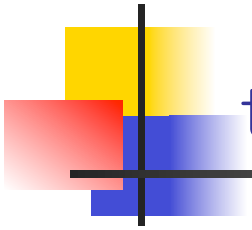
- This program plots a line by reading in a series of point coordinate positions. It works fine as long as the user enters valid input. But with invalid input, the program crashes. Add code so as to avoid those crashes.

```
import java.util.Scanner;

public class LinePlot
{
  private int oldX = 0; // oldX, oldY save the previous point
  private int oldY = 0; // The starting point is the origin (0,0)

  //*********************************************************

  // This method prints a line segment from the previous point
  // to the current point.

  public void plotSegment(int x, int y)
  {
    System.out.println("New segment = (" + oldX + "," + oldY +
      ")-(" + x + "," + y + ")");
    oldX = x;
    oldY = y;
  } // end plotSegment
```

Line Plot Example

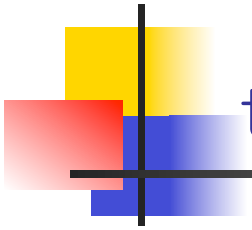
```
//*************************************************************

public static void main(String[] args)
{
  Scanner stdIn = new Scanner(System.in);
  LinePlot line = new LinePlot();
  String xStr, yStr;   // coordinates for a point (String form)
  int x, y;            // coordinates for a point

  System.out.print("Enter x & y coordinates (q to quit): ");
  xStr = stdIn.next();
  while (!xStr.equalsIgnoreCase("q"))
  {
    yStr = stdIn.next();
    x = Integer.parseInt(xStr);
    y = Integer.parseInt(yStr);
    line.plotSegment(x, y);
    System.out.print("Enter x & y coordinates (q to quit): ");
    xStr = stdIn.next();
  } // end while
} // end main
} // end class LinePlot
```

reads a group of characters and stops at whitespace



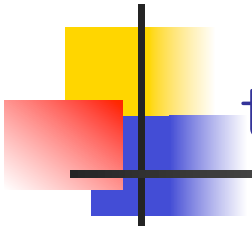
`try` and `catch` Blocks - More Details

- Deciding on the size of your `try` blocks is a bit of an art. Sometimes it's better to use small `try` blocks and sometimes it's better to use larger `try` blocks.
- Note that it's legal to surround an entire method body with a `try` block, but that's usually counterproductive because it makes it harder to identify the "dangerous" code.
- In general, you should make your `try` blocks small so that your "dangerous" code is more obvious.
- However, if a chunk of code has several "dangerous" method/ constructor calls:
  - Adding a separate `try-catch` structure for each such call might result in cluttered code.
  - To improve program readability, consider using a single `try` block that surrounds the calls.



`try` and `catch` Blocks - More Details

- In our LinePlot program solution, we surrounded the two `parseInt` statements with a single `try` block because they were conceptually related and physically close together. We also included the `line.plotSegment()` call within that same `try` block. Why?
- Our single `try` block solution is perfectly acceptable, but wouldn't it be nice to have a more specific message that identified which entry was invalid (x, y, or both)?.
- To have that sort of message, you'd have to have a separate `try-catch` structure for each `parseInt` statement.



`try` and `catch` Blocks - More Details

- If an exception is thrown, the JVM immediately jumps out of the current `try` block and looks for a matching `catch` block. The immediacy of the jump means that if there are statements in the `try` block after the exception-throwing statement, those statements are skipped.
- The compiler is a pessimist. It knows that statements inside a `try` block might possibly be skipped, and it assumes the worst. It assumes that all statements inside a `try` block get skipped.
- Consequently, if there's a `try` block that contains an assignment for x, the compiler assumes that the assignment is skipped. If there's no assignment for x outside of the `try` block and x's value is needed outside of the try block, you'd get this compilation error:

```
variable x might not have been initialized
```

- If you get that error, you can usually fix it by initializing the variable prior to the try block.

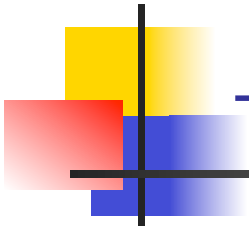
try and catch Blocks - More Details

- This method reads a value from a user, makes sure it's an integer, and returns it. Note the compilation errors. What are the fixes?

```
public static int getIntFromUser()
{
  Scanner stdIn = new Scanner(System.in);
  String xStr;   // user entry
  boolean valid; // is user entry a valid integer?
  int x;         // integer form of user entry

  System.out.print("Enter an integer: ");
  xStr = stdIn.next();
  do
  {
    try
    {
      valid = false;
      x = Integer.parseInt(xStr);
      valid = true;
    }
    catch (NumberFormatException nfe)
    {
      System.out.print("Invalid entry. Enter an integer: ");
      xStr = stdIn.next();
    }
  } while (!valid);

  return x;
} // end getIntFromUser
```

compilation error: variable valid might not have been initialized

compilation error: variable x might not have been initialized

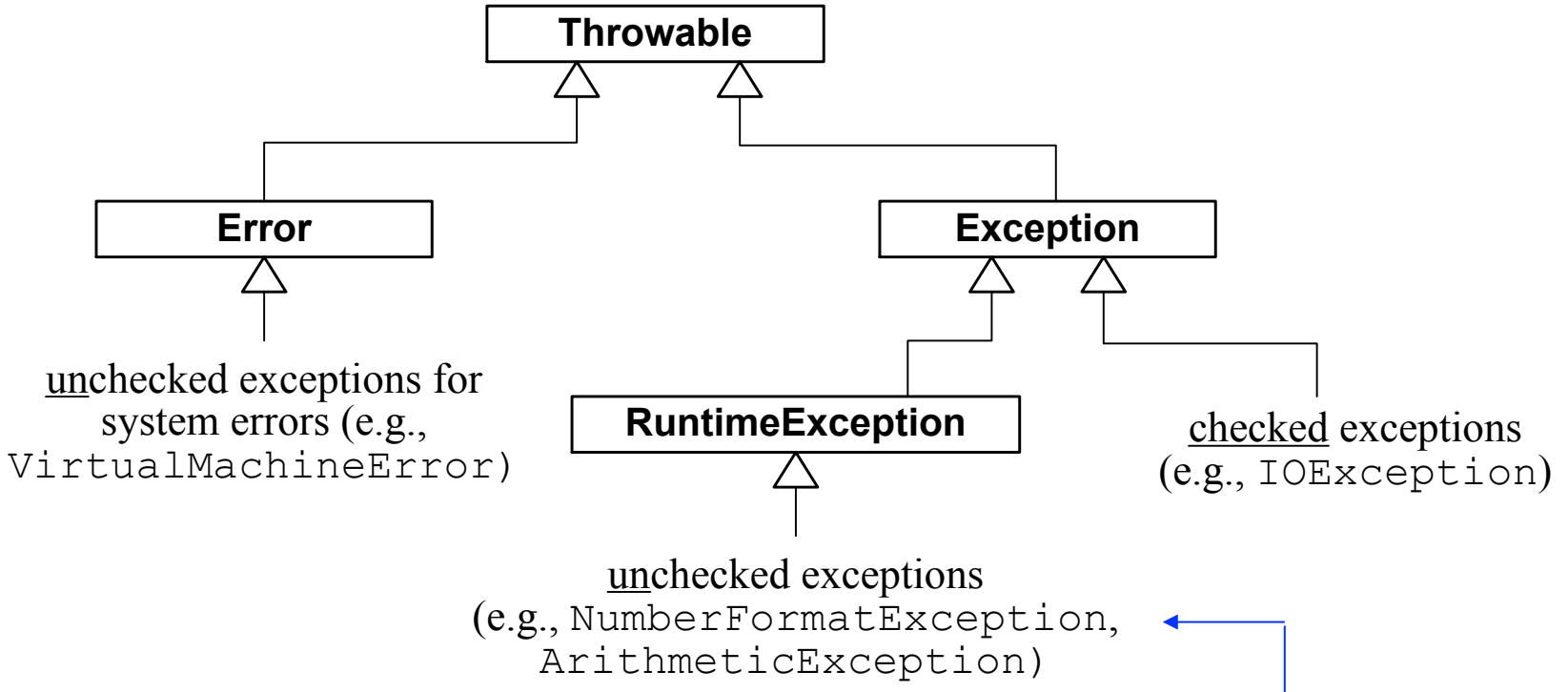


Two Types of Exceptions - Checked and Unchecked

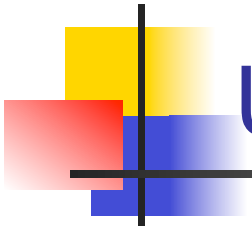
- There are two types of exceptions – checked and unchecked.
  - Checked exceptions are required to be checked with a `try-catch` mechanism.
  - Unchecked exceptions are not required to be checked with a `try-catch` mechanism (but, as an option, unchecked exceptions may be checked with a `try-catch` mechanism).
- How can you tell whether a particular exception is classified as checked or unchecked?
  - To find out if a particular exception is checked or unchecked, look up its associated class in the API documentation.
  - On the class's API page, look at its class hierarchy tree. If you find that the class is derived from the `RuntimeExeption` class or from the `Error` exception class, then it's an unchecked exception. Otherwise, it's a checked exception.

Two Types of Exceptions - Checked and Unchecked

Class Hierarchy For Exception Classes



The `parseInt`, `parseLong`, `parseDouble`, etc. methods all throw a `NumberFormatException` object.

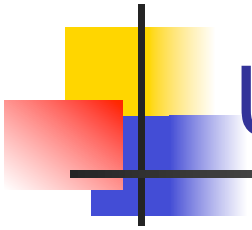


Unchecked Exceptions

- As you know, unchecked exceptions are not required to be checked with a `try-catch` mechanism. However, at runtime, if an unchecked exception is thrown and not caught, then the program will *crash* (terminate ungracefully).

- How to handle code that might throw an unchecked exception:
  - Use a `try-catch` mechanism (see prior `GetIntFromUser` example).

    or
  - Don't attempt to catch the exception, but write the code carefully so as to avoid the possibility of the exception being thrown (see upcoming example).

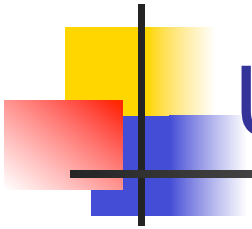


Unchecked Exceptions

- The following method attempts to remove a specified student from a list of student names. The list of student names is stored in an `ArrayList` instance variable named `students`.

```
public void removeStudent(int index)
{
  students.remove(index);
} // end removeStudent
```

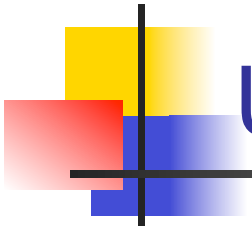
- The `students.remove` method call is dangerous because it throws an unchecked exception, `IndexOutOfBoundsException`, if its argument holds an invalid index.
- On the upcoming slides, we address that problem by providing improved versions of the `removeStudent` method.



Unchecked Exceptions

- Improved `removeStudent` method using a `try-catch` mechanism:

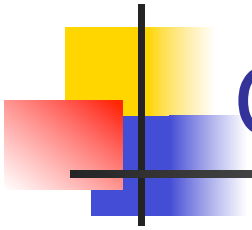
```
public void removeStudent(int index)
{
  try
  {
    students.remove(index);
  }
  catch (IndexOutOfBoundsException e)
  {
    System.out.println(
      "Can't remove student because " + index +
      " is an invalid index position.");
  }
} // end removeStudent
```



Unchecked Exceptions

- Improved `removeStudent` method, using careful code:

```
public void removeStudent(int index)
{
  if (index >= 0 && index < students.size())
  {
    students.remove(index);
  }
  else
  {
    System.out.println(
      "Can't remove student because " + index +
      " is an invalid index position.");
  }
} // end removeStudent
```



Checked Exceptions

- If a code fragment has the potential of throwing a checked exception, then the compiler requires that the code fragment has an associated `try-catch mechanism`. If there is no associated `try-catch mechanism`, then the compiler generates an error.

- The program on the next slide contains code that might possibly throw a checked exception and there's no `try-catch mechanism`. Thus, it generates a compilation error. What code should be added to fix the program?

Checked Exceptions

Program synopsis:

Prompt the user for the name of a file that is to be created.

If the file exists, print a "Sorry" message.

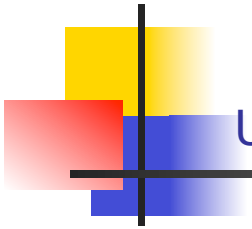
If the file does not exist, create the file.

```
import java.util.Scanner;
import java.io.File;
import java.io.IOException;

public class CreateNewFile
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String fileName; // user-specified file name
    File file;
    System.out.print("Enter file to be created: ");
    fileName = stdIn.nextLine();
    file = new File(fileName);
    if (file.exists())
    {
      System.out.println("Sorry, can't create that file. It already exists.");
    }
    else
    {
      file.createNewFile();
      System.out.println(fileName + " created.");
    }
  } // end main
} // end CreateNewFile class
```

API constructor call

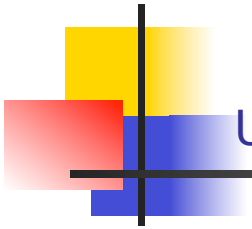
API method call

API method call



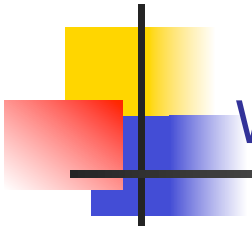
Using API Documentation when Writing Exception-Handling Code

- Whenever you want to use a method or constructor from one of the API classes and you're not sure about it, you should look it up in the API documentation so you know whether to add exception-handling code.
- More specifically, use the API documentation to figure out these things:
  - Can the method/constructor call possibly throw an exception?
    - On the API documentation page for the method/constructor, look for a `throws` section. If there's a `throws` section, that means the method/constructor can possibly throw an exception.
  - If the method/constructor call throws an exception, is it checked or unchecked?
    - On the API documentation page for the method/constructor, drill down on the exception class's name.
    - On the API documentation page for the exception class, look at the exception class's class hierarchy.
    - If you find `RuntimeException` is an ancestor of the exception, then the exception is an unchecked exception. Otherwise, it's a checked exception.



Using API Documentation when Writing Exception-Handling Code

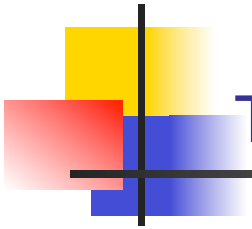
- If the method/constructor call can possibly throw a checked exception, you must add a `try-catch` mechanism to handle it.
- If the method/constructor call can possibly throw an unchecked exception, you should read the API documentation to figure out the nature of the exception. And then, depending on the situation, 1) use a `try-catch` mechanism or 2) use careful code so that the exception won't be thrown.



When a `try` Block Throws Different Types of Exceptions

- If several statements within a `try` block can possibly throw an exception and the exceptions are of different types, you should:
  - Provide a generic `catch` block that handles every type of exception that might be thrown.

    or
  - Provide a sequence of specific `catch` blocks, one for each type of exception that might be thrown.

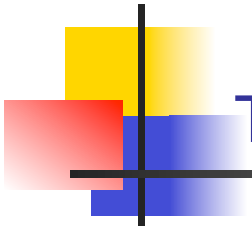


The `Exception` Class and its `getMessage` Method

- How to provide a generic `catch` block:
  - Define a `catch` block with an `Exception` parameter.
  - Inside the `catch` block, call the `Exception` class's `getMessage` method.
- For example:

```
catch (Exception e)
{
   System.out.println(e.getMessage());
}
```

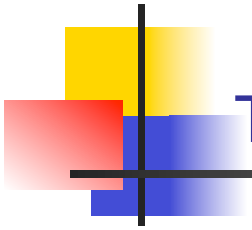
- Why do all thrown exceptions match up with an `Exception` parameter?
  - A thrown exception will be caught by a `catch` block if the thrown exception equals the `catch` heading's parameter or the thrown exception is a subclass of the `catch` heading's parameter.
  - Since every thrown exception is a subclass of the `Exception` class, all thrown exceptions will match up with a generic `Exception` parameter.



The `Exception` Class and its `getMessage` Method

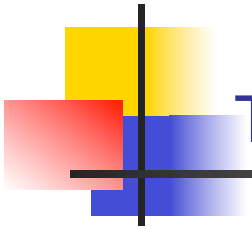
- The `Exception` class's `getMessage` method returns a description of the thrown exception. For example, if you attempt to open a file using the `FileReader` constructor call and you pass in a file name for a file that doesn't exist, the `getMessage` call returns this:

  <*filename*> `(The system cannot find the file specified)`



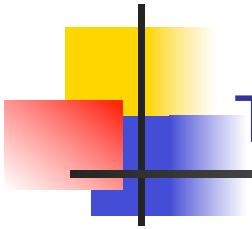
The `Exception` Class and its `getMessage` Method

- The program on the next slide opens a user-specified file and prints the file's first character.
  - The `FileReader` constructor is in charge of opening the file. In your constructor call, if you pass in a file name for a file that doesn't exist, the JVM throws a `FileNotFoundException`.
  - The `read` method is in charge of reading a single character from the opened file. If the file is corrupted and unreadable, the JVM throws an `IOException`.
- Note the generic `catch` block. It handles the different types of exceptions that might be thrown from within the `try` block.



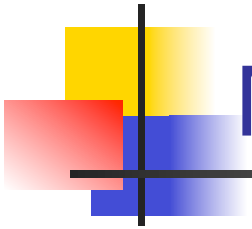
The `Exception` Class and its `getMessage` Method

```
/*************************************************************
* PrintCharFromFile.java
* Dean & Dean
*
* Open an existing text file and print a character from it.
*************************************************************/

import java.util.Scanner;
import java.io.BufferedReader;
import java.io.FileReader;

public class PrintCharFromFile
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String fileName;         // name of target file
    BufferedReader fileIn;   // target file
    char ch;                 // first character from fileIn
```



The `Exception` Class and its `getMessage` Method

```
      System.out.print("Enter a filename: ");
      fileName = stdIn.nextLine();

      try
      {
        fileIn = new BufferedReader(new FileReader(fileName));
        ch = (char) fileIn.read();
        System.out.println("First character: " + ch);
      } // end try

      catch (Exception e)
      {
        System.out.println(e.getMessage());
      }
    } // end main
} // end PrintCharFromFile class
```

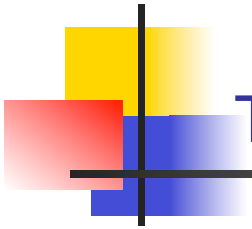


Multiple `catch` Blocks

- If several statements within a `try` block can possibly throw an exception and the exceptions are of different types, and you don't want to use a generic `catch` block, you should:
  - Provide a sequence of specific `catch` blocks, one for each type of exception that might be thrown.
- For example:

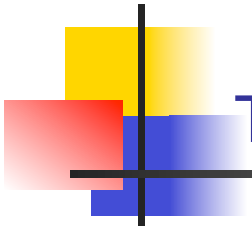
```
catch (FileNotFoundException e)
{
  System.out.println("Invalid filename: " + fileName);
}
catch (IOException e)
{
  System.out.println("Error reading from file: " + fileName);
}
```

- What's a benefit of using specific `catch` blocks rather than a generic `catch` block?



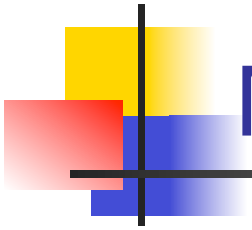
The `Exception` Class and its `getMessage` Method

```
import java.util.Scanner;
import java.io.BufferedReader;
import java.io.FileReader;
import java.io.FileNotFoundException;
import java.io.IOException;

public class PrintCharFromFile2
{
  public static void main(String[] args)
  {
    Scanner stdIn = new Scanner(System.in);
    String fileName;         // name of target file
    BufferedReader fileIn;   // target file
    char ch;                 // first character from fileIn

    System.out.print("Enter a filename: ");
    fileName = stdIn.nextLine();
```



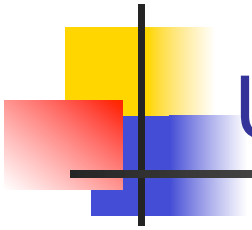
The `Exception` Class and its `getMessage` Method

```
    try
    {
      fileIn = new BufferedReader(new FileReader(fileName));
      ch = (char) fileIn.read();
      System.out.println("First character: " + ch);
    } // end try

    catch (FileNotFoundException e)
    {
      System.out.println("Invalid filename: " + fileName);
    }
    catch (IOException e)
    {
      System.out.println("Error reading from file: " + fileName);
    }
  } // end main
} // end PrintCharFromFile2 class
```



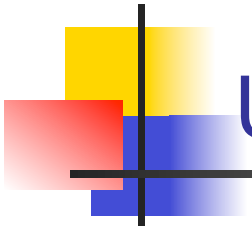
Multiple `catch` Blocks

- If multiple `catch` blocks are used, the first `catch` block that matches the type of the exception thrown is the one that is executed; the other `catch` blocks are then skipped.
- Whenever you use more than one `catch` block after a given `try` block, and one catch block's exception class is derived from another catch block's exception class, to avoid a compilation error, you must arrange the catch blocks with the more general exception classes at the bottom (the superclasses go at the bottom).
- For example, in the prior PrintCharFromFile2 program, you must put the `IOException catch` block at the bottom because the `IOException` class is a superclass of the `FileNotFoundException` class.



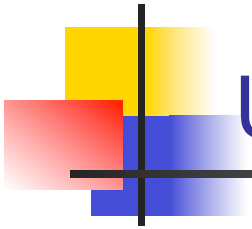
Understanding Exception Messages

- As you know, if your code involves a checked exception being thrown, you must include a `try`/`catch` for that code. Without the `try`/`catch`, your program won't compile successfully.
- On the other hand, if your code involves an unchecked exception being thrown, it's optional whether you include a `try`/`catch` for that code. Without the `try`/`catch`, your program will compile successfully, but if an exception is thrown, your program will crash.
- If such a crash occurs, the JVM prints a runtime error message that describes the thrown exception.



Understanding Exception Messages

```
import java.util.Scanner;

public class NumberList
{
  private int[] numList = new int[100]; // array of numbers
  private int size = 0;                 // number of numbers

  //***************************************

  public void readNumbers()
  {
    Scanner stdIn = new Scanner(System.in);
    String xStr;   // user-entered number (String form)
    int x;         // user-entered number

    System.out.print("Enter a whole number (q to quit): ");
    xStr = stdIn.next();

    while (!xStr.equalsIgnoreCase("q"))
    {
      x = Integer.parseInt(xStr);
      numList[size] = x;
      size++;
      System.out.print("Enter a whole number (q to quit): ");
```

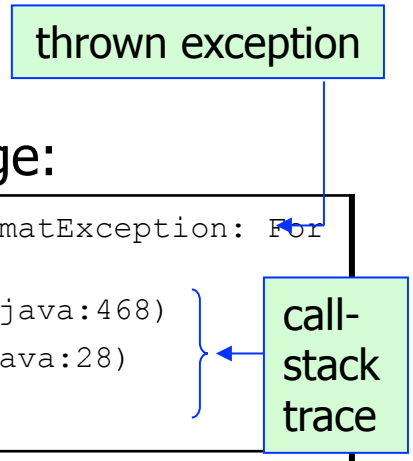


Understanding Exception Messages

```
      xStr = stdIn.next();
    } // end while
  } // end readNumbers

  //***************************************

  public double getMean()
  {
    int sum = 0;
    for (int i=0; i<size; i++)
    {
      sum += numList[i];
    }
    return sum / size;
  } // end getMean

  //***************************************

  public static void main(String[] args)
  {
    NumberList list = new NumberList();
    list.readNumbers();
    System.out.println("Mean = " + list.getMean());
  } // end main
} // end class NumberList
```

Understanding Exception Messages

- The `NumberList` program compiles and runs, but it's not very robust. See below:

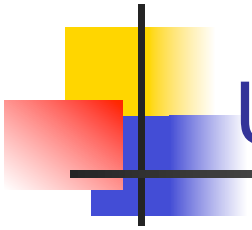


If this happens:	Approximate error message:

User enters a non-integer (e.g., hi).	`Exception in thread "main" java.lang.NumberFormatException: For input string: "hi"`
	`at java.lang.Integer.parseInt(Integer.java:468)`
	`at NumberList.readNumbers(NumberList.java:28)`
	`at NumberList.main(NumberList.java:73)`

User immediately enters q to quit.	`Exception in thread "main" java.lang.ArithmeticException: / by zero`
	`at NumberList.getMean(NumberList.java:49)`
	`at NumberList.main(NumberList.java:58)`

User enters more than 100 numbers.	`Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 100`
	`at NumberList.readNumbers(NumberList.java:29)`
	`at NumberList.main(NumberList.java:73)`



Understanding Exception Messages

- As part of a runtime error, the JVM prints the exception that was thrown and then prints the *call-stack trace*. The call-stack trace shows the methods that were called prior to the crash.
- If you perform integer division with a denominator of zero, the JVM throws an `ArithmeticException` object.
- If you access an array element with an array index that's < 0 or >= the array's size, the JVM throws an `ArrayIndexOutOfBoundsException` object.