Hierarchy of Exception Classes

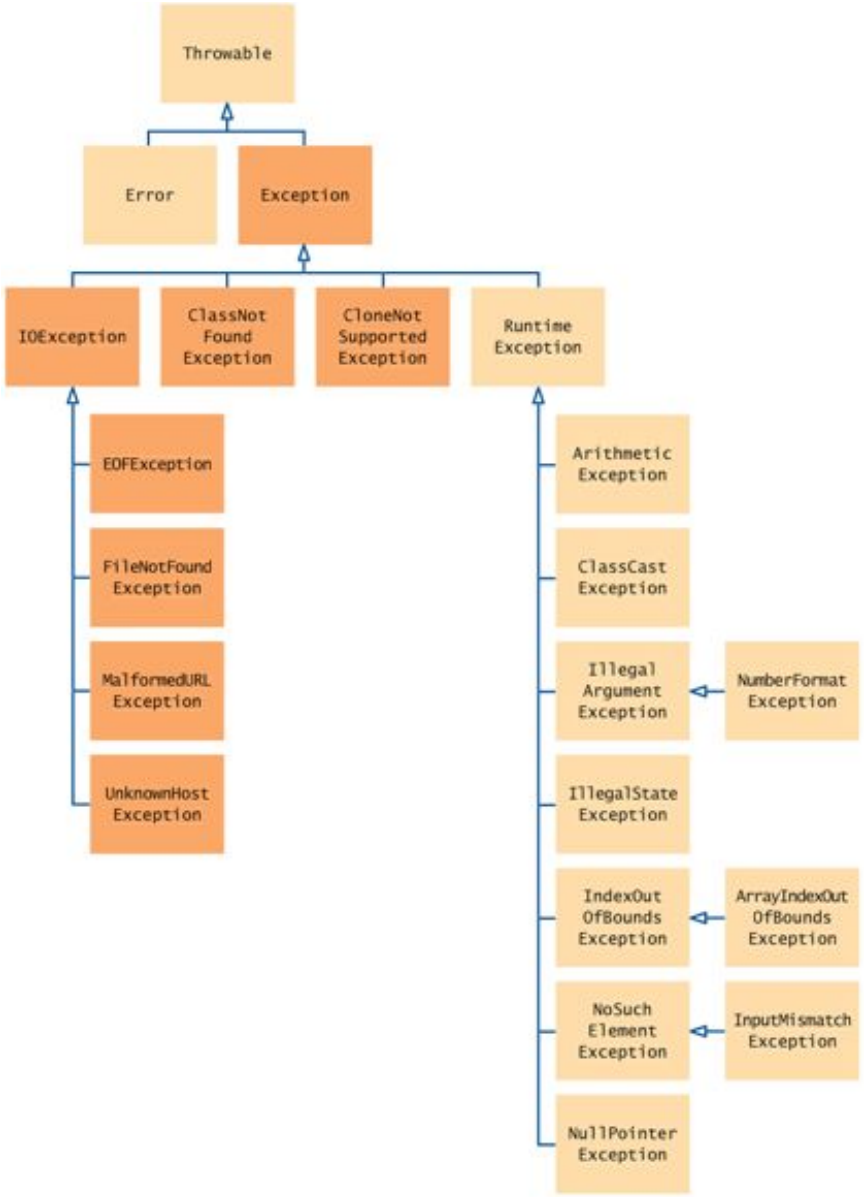

Figure 1 The Hierarchy of Exception Classes

Checked and Unchecked Exceptions

- Two types of exceptions:
  - *Checked*
    - *The compiler checks that you don't ignore them*
    - *Due to external circumstances that the programmer cannot prevent*
    - *Majority occur when dealing with input and output*
    - *For example,* **`IOException`**
  - *Unchecked*
    - *Extend the class* **`RuntimeException`** *or* **`Error`**
    - *They are the programmer's fault*
    - *Examples of runtime exceptions:*

      `NumberFormatException`
      `IllegalArgumentException`
      `NullPointerException`
    - *Example of error:*

      `OutOfMemoryError`

Checked and Unchecked Exceptions

- Categories aren't perfect:
  - ***`Scanner.nextInt`*** *throws unchecked* ***`InputMismatchException`***
  - *Programmer cannot prevent users from entering incorrect input*
  - *This choice makes the class easy to use for beginning programmers*
- Deal with checked exceptions principally when programming with files and streams
- For example, use a **`Scanner`** to read a file:

```
String filename = ...;
File reader = new File(filename);
Scanner in = new Scanner(reader);
```

- But, **`File`** constructor can throw a **`FileNotFoundException`**

Checked and Unchecked Exceptions

- Two choices:
  1. *Handle the exception*
  2. *Tell compiler that you want method to be terminated when the exception occurs*
     - *Use* **`throws`** *specifier so method can throw a checked exception*

```
public void read(String filename) throws
   FileNotFoundException
{
   File reader = new File(filename);
   Scanner in = new Scanner(reader);
   ...
}
```

     - *For multiple exceptions:*

```
public void read(String filename)
   throws IOException, ClassNotFoundException
```

Continued

Syntax 11.2 `throws` Clause

Syntax

accessSpecifier returnType methodName(parameterType parameterName, . . .)
`throws` *ExceptionClass, ExceptionClass, . . .*

Example

```
public void read(String filename)
      throws FileNotFoundException, NoSuchElementException
```

You must specify all checked exceptions that this method may throw.

You may also list unchecked exceptions.

Checkpoint

Suppose a method calls the `Scanner` constructor, which can throw a `FileNotFoundException`, and the `nextInt` method of the `Scanner` class, which can cause a `NoSuchElementException` or `InputMismatchException`.

Which exceptions should be included in the `throws` clause?

Answer:

You must include the `FileNotFoundException` and you may include the `NoSuchElementException` if you consider it important for documentation purposes.

`InputMismatchException` is a subclass of `NoSuchElementException`. It is your choice whether to include it.

Catching Exceptions

- Install an exception handler with `try/catch` statement
- `try` block contains statements that may cause an exception
- `catch` clause contains handler for an exception type

Continued

Syntax 11.3 Catching Exceptions

Syntax

```
try
{
   statement
   statement
   . . .
}
catch (ExceptionClass exceptionObject)
{
   statement
   statement
   . . .
}
```

Example

```
try
{
   Scanner in = new Scanner(new File("input.txt"));
   String input = in.next();
   process(input);
}
catch (IOException exception)
{
   System.out.println("Could not open input file");
}
```

This constructor can throw a FileNotFoundException.

This is the exception that was thrown.

When an IOException is thrown, execution resumes here.

Additional catch clauses can appear here.

A FileNotFoundException is a special case of an IOException.

Catching Exceptions

- Statements in **`try`** block are executed
- If no exceptions occur, **`catch`** clauses are skipped
- If exception of matching type occurs, execution jumps to **`catch`** clause
- If exception of another type occurs, it is thrown until it is caught by another **`try`** block
- **`catch (IOException exception) block`**
  - ***`exception`*** *contains reference to the exception object that was thrown*
  - ***`catch`*** *clause can analyze object to find out more details*
  - ***`exception.printStackTrace()`*** *Printout of chain of method calls that lead to exception*

Catching Exceptions Example

```
try
{
   String filename = ...;
   File reader = new File(filename);
   Scanner in = new Scanner(reader);
   String input = in.next();
   int value = Integer.parseInt(input);
   ...
}
catch (IOException exception)
{
   exception.printStackTrace();
}
catch (NumberFormatException exception)
{
System.out.println("Input was not a number");
}
```

Checkpoint

Is there a difference between catching checked and unchecked exceptions?

Answer:

No — you catch both exception types in the same way, as you can see from the above code example. Recall that `IOException` is a checked exception and `NumberFormatException` is an unchecked exception.

The `finally` Clause

- Executed when **`try`** block is exited in any of three ways:
  1. *After last statement of **`try`** block*
  2. *After last statement of catch clause, if this **`try`** block caught an exception*
  3. *When an exception was thrown in **`try`** block and not caught*

The `finally` Clause

- Exception terminates current method
- Danger: Can skip over essential code
- Example:

```
reader = new File(filename);
Scanner in = new Scanner(reader);
readData(in);
reader.close(); // May never get here
```

- Must execute `reader.close()` even if exception happens
- Use `finally` clause for code that must be executed “no matter what”

The `finally` Clause

```
File reader = new File(filename);
try
{
   Scanner in = new Scanner(reader);
   readData(in);
}
finally
{
   reader.close();
   // if an exception occurs, finally clause
   // is also executed before exception
   // is passed to its handler
}
```

Example

```
public class BankAccount
{
   public void withdraw(double amount)
   {
      if (amount > balance)
      {
         IllegalArgumentException exception
            = new IllegalArgumentException("Amount
            exceeds balance");
         throw exception;
      }
      balance = balance - amount;
   }
   ...
}
```

Syntax 11.4 `finally` Clause

Syntax

```
try
{
   statement
   statement
   . . .
}
finally
{
   statement
   statement
   . . .
}
```

Example

This variable must be declared outside the `try` block so that the `finally` clause can access it.

```
PrintWriter out = new PrintWriter(filename);
try
{
   writeData(out);
}
finally
{
   out.close();
}
```

This code may throw exceptions.

This code is always executed, even if an exception occurs.

Checkpoint

Why was the `out` variable declared outside the `try` block (slide 18)?

Answer:

If it had been declared inside the `try` block, its scope would only have extended to the end of the `try` block, and the `finally` clause could not have closed it.

Throwing Exceptions

- Throw an exception object to signal an exceptional condition
- Example: `IllegalArgumentException`: Illegal parameter value:

```
IllegalArgumentException exception
   = new IllegalArgumentException("Amount exceeds
   balance");
throw exception;
```

- No need to store exception object in a variable:

```
throw new IllegalArgumentException("Amount exceeds
   balance");
```

- When an exception is thrown, method terminates immediately
  - ***Execution continues with an exception handler***

Syntax 11.1 Throwing an Exception

Syntax `throw` *exceptionObject*;

Example

```
if (amount > balance)
{
   throw new IllegalArgumentException("Amount exceeds balance");
}
balance = balance - amount;
```

Most exception objects can be constructed with an error message.

A new exception object is constructed, then thrown.

This line is not executed when the exception is thrown.

Designing Your Own Exception Types

- You can design your own exception types — subclasses of **`Exception`** or **`RuntimeException`**
-
```
if (amount > balance)
{
   throw new InsufficientFundsException(
      "withdrawal of " + amount + " exceeds balance of "
      + balance);
}
```
- Make it an unchecked exception — programmer could have avoided it by calling **`getBalance`** first
- Extend **`RuntimeException`** or one of its subclasses
- Supply two constructors
  1. *Default constructor*
  2. *A constructor that accepts a message string describing reason for exception*

Designing Your Own Exception Types

```
public class InsufficientFundsException
      extends RuntimeException
{
   public InsufficientFundsException() {}

   public InsufficientFundsException(String message)
   {
      super(message);
   }
}
```

Checkpoint

What is the purpose of the call `super(message)` in the second `InsufficientFundsException` constructor?

Answer:

To pass the exception message string to the `RuntimeException` superclass.

Checkpoint

Suppose you read bank account data from a file. Contrary to your expectation, the next input value is not of type `double`. You decide to implement a `BadDataException`.

Which exception class should you extend?

Answer:

Because file corruption is beyond the control of the programmer, this should be a checked exception, so it would be wrong to extend `RuntimeException` or `IllegalArgumentException`. Because the error is related to input, `IOException` would be a good choice.

Case Study: A Complete Example

- Program
  - Asks user for name of file
  - File expected to contain data values
  - First line of file contains total number of values
  - Remaining lines contain the data
  - Typical input file:

```
  3
1.45
-2.1
0.05
```

Case Study: A Complete Example

- What can go wrong?
  - *File might not exist*
  - *File might have data in wrong format*
- Who can detect the faults?
  - ***`File`*** *constructor will throw an exception when file does not exist*
  - *Methods that process input need to throw exception if they find error in data format*
- What exceptions can be thrown?
  - ***`FileNotFoundException`*** *can be thrown by* ***`File`*** *constructor*
  - ***`IOException`*** *can be thrown by* ***`close`*** *method of* ***`File`***
  - ***`BadDataException`****, a custom checked exception class*

Case Study: A Complete Example

- Who can remedy the faults that the exceptions report?
  - *Only the* `main` *method of* `DataSetTester` *program interacts with user*
  - *Catches exceptions*
  - *Prints appropriate error messages*
  - *Gives user another chance to enter a correct file*

DataAnalyzer.java

```
 1  import java.io.FileNotFoundException;
 2  import java.io.IOException;
 3  import java.util.Scanner;
 4
 5  /**
 6     This program reads a file containing numbers and analyzes its contents.
 7     If the file doesn&apos;t exist or contains strings that are not numbers, an
 8     error message is displayed.
 9  */
10  public class DataAnalyzer
11  {
12     public static void main(String[] args)
13     {
14        Scanner in = new Scanner(System.in);
15        DataSetReader reader = new DataSetReader();  // DataSetReader is class
16                                 // we define - shown later
17        boolean done = false;
18        while (!done)
19        {
```

Continued

DataAnalyzer.java (cont.)

```
20            try
21            {
22               System.out.println("Please enter the file name: ");
23               String filename = in.next();
24
25               double[] data = reader.readFile(filename);
26               double sum = 0;
27               for (double d : data) sum = sum + d;
28               System.out.println("The sum is " + sum);
29               done = true;
30            }
31            catch (FileNotFoundException exception)
32            {
33               System.out.println("File not found.");
34            }
35            catch (BadDataException exception)
36            {
37               System.out.println("Bad data: " + exception.getMessage());
38            }
39            catch (IOException exception)
40            {
41               exception.printStackTrace();
42            }
43         }
44      }
45   }
```

The `readFile` Method of the `DataSetReader` Class

- Constructs `Scanner` object
- Calls `readData` method
- Completely unconcerned with any exceptions

The `readFile` Method of the `DataSetReader` Class

- If there is a problem with input file, it simply passes the exception to caller:

```
public double[] readFile(String filename)
   throws IOException, BadDataException
   // FileNotFoundException is an IOException
{
   File reader = new File(filename);
   try
   {
      Scanner in = new Scanner(reader);
      readData(in);
   }
   finally
   {
      reader.close();
   }
   return data;
}
```

The `readFile` Method of the `DataSetReader` Class

- Reads the number of values
- Constructs an array
- Calls `readValue` for each data value:

```
  private void readData(Scanner in) throws BadDataExceptio
{
   if (!in.hasNextInt())
      throw new BadDataException("Length expected");
   int numberOfValues = in.nextInt();
   data = new double[numberOfValues];

   for (int i = 0; i < numberOfValues; i++)
      readValue(in, i);

   if (in.hasNext())
      throw new BadDataException("End of file expected");
}
```

The `readFile` Method of the `DataSetReader` Class

- Checks for two potential errors
  1. *File might not start with an integer*
  2. *File might have additional data after reading all values*
- Makes no attempt to catch any exceptions

The `readFile` Method of the `DataSetReader` Class

```
private void readValue(Scanner in, int i) throws
   BadDataException
{
   if (!in.hasNextDouble())
      throw new BadDataException("Data value expected");
   data[i] = in.nextDouble();
}
```

Scenario

1. `DataSetTester.main` calls `DataSetReader.readFile`
2. `readFile` calls `readData`
3. `readData` calls `readValue`
4. `readValue` doesn't find expected value and throws `BadDataException`
5. `readValue` has no handler for exception and terminates
6. `readData` has no handler for exception and terminates
7. `readFile` has no handler for exception and terminates after executing `finally` clause
8. `DataSetTester.main` has handler for `BadDataException;` handler prints a message, and user is given another chance to enter file name

DataSetReader.java

```
 1  import java.io.File;
 2  import java.io.IOException;
 3  import java.util.Scanner;
 4
 5  /**
 6     Reads a data set from a file. The file must have the format
 7     numberOfValues
 8     value1
 9     value2
10     . . .
11  */
12  public class DataSetReader
13  {
14     private double[] data;
15
```

Continued

DataSetReader.java (cont.)

```
16       /**
17          Reads a data set.
18          @param filename the name of the file holding the data
19          @return the data in the file
20       */
21       public double[] readFile(String filename) throws IOException
22       {
23          File inFile = new File(filename);
24          Scanner in = new Scanner(inFile);
25          try
26          {
27             readData(in);
28             return data;
29          }
30          finally
31          {
32             in.close();
33          }
34       }
35
```

Continued

DataSetReader.java (cont.)

```
36      /**
37         Reads all data.
38         @param in the scanner that scans the data
39      */
40      private void readData(Scanner in) throws BadDataException
41      {
42         if (!in.hasNextInt())
43            throw new BadDataException("Length expected");
44         int numberOfValues = in.nextInt();
45         data = new double[numberOfValues];
46
47         for (int i = 0; i < numberOfValues; i++)
48            readValue(in, i);
49
50         if (in.hasNext())
51            throw new BadDataException("End of file expected");
52      }
53
```

Continued

DataSetReader.java (cont.)

```
54      /**
55         Reads one data value.
56         @param in the scanner that scans the data
57         @param i the position of the value to read
58      */
59      private void readValue(Scanner in, int i) throws BadDataException
60      {
61         if (!in.hasNextDouble())
62            throw new BadDataException("Data value expected");
63         data[i] = in.nextDouble();
64      }
65   }
```

BadDataException.java

```
1  import java.io.IOException;
2
3  /**
4     This class reports bad input data.
5  */
6  public class BadDataException extends IOException
7  {
8     public BadDataException() {}
9     public BadDataException(String message)
10    {
11       super(message);
12    }
13 }
```

Checkpoint

Why doesn't the `DataSetReader.readFile` method catch any exceptions?

Answer: It would not be able to do much with them. The `DataSetReader` class is a reusable class that may be used for systems with different languages and different user interfaces. Thus, it cannot engage in a dialog with the program user.

Checkpoint

Suppose the user specifies a file that exists and is empty. Trace the flow of execution.

Answer: `DataSetAnalyzer.main` calls `DataSetReader.readFile`, which calls `readData`. The call `in.hasNextInt()` returns `false`, and `readData` throws a `BadDataException`. The `readFile` method doesn't catch it, so it propagates back to `main`, where it is caught.

Questions?