Exception Handling

Reading: Chapter 11.4 / 11.5 (BigJava)

CS 2110

Prof. Nada Basit
University of Virginia

Spring 2020

Announcements

- Welcome to the new online format!
  - Every section is conducted in a slightly different way
  - (We'll talk about **section 003** specifics today)
  - View Piazza centralized post for full details: https://piazza.com/class/k58tapw9v4j2d8?cid=398

- Chat Etiquette
  - Chat should be used for questions/answers relevant to the course material only
  - No off-topic chatter, spam, or inappropriate posts
  - Please be kind to each other!

Immediate Schedule Changes

- Exam 2
  - Moved to April 3
- Homework 4 (GUI)
  - Due by 11:30pm on April 1 – *not* an April Fool's joke!
  - (To account for the days of office hours that were lost)
- Office Hours
  - Instructor and TA office hours resume next Mon., March 23
- Weekly Quiz
  - No quiz **this** week – There will be a quiz next week
- Weekly Labs
  - Due by 11:30pm Tuesday Night (seek help: Discord for lab)
  - See Piazza post for Lab Discord link

Lecture: Live/Recording

- **Live Streaming** on **Twitch** (+ recording)
  - Section 001 – McBurney
  - **Section 003 – Basit**
- **Live Streaming** using **Zoom** (+recording)
  - Section 001 – Apostolellis
- **Recording** (only) using **Panopto**
  - Section 004 – Stone

Please be sure to **join** my Twitch channel! Link in announcement that I sent out. Thank you!

(See pinned post on Piazza)

Instructor Office Hours

- **Discord**
  - Section 001 – McBurney
  - **Section 003 – Basit**
- **Slack**
  - Section 004 – Stone

[Section 002 (Apostolellis) – ***TBD***]

Please be sure to **join** my Discord server! Link in announcement that I sent out. Thank you!

(See pinned post on Piazza for more details)

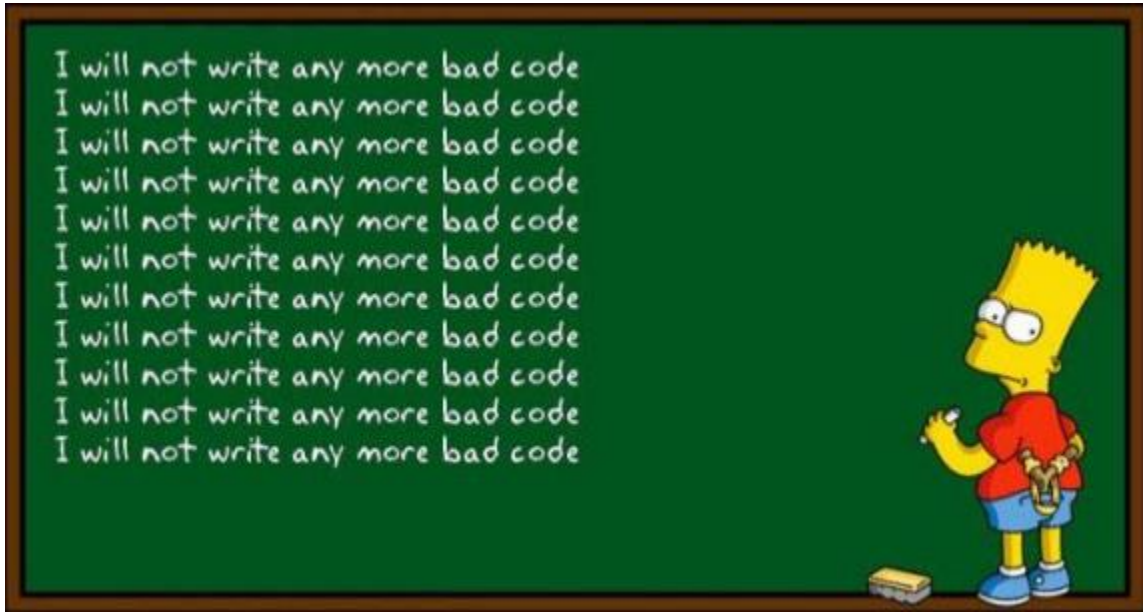
Goals : Exceptions

- To throw and catch exceptions
- To implement programs that propagate checked exceptions

Exceptions

- Exceptions are events that disrupt the intended program flow
- **Exception** is short for exceptional Events
- All exceptions must be:
  - Detected
  - Handled

  during coding to prevent halting the program



Exception Handling – Throwing Exceptions

- Exception handling provides a flexible mechanism for passing control from the point of error detection to a handler that can deal with the error.
- When you detect an error condition, throw an exception object to signal an ***exceptional condition***

Example:

- If someone tries to withdraw too much money from a bank account:
  - Throw an **IllegalArgumentException**

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

This is an object from class Exception

Syntax Throwing an Exception

Syntax `throw` *exceptionObject*;

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

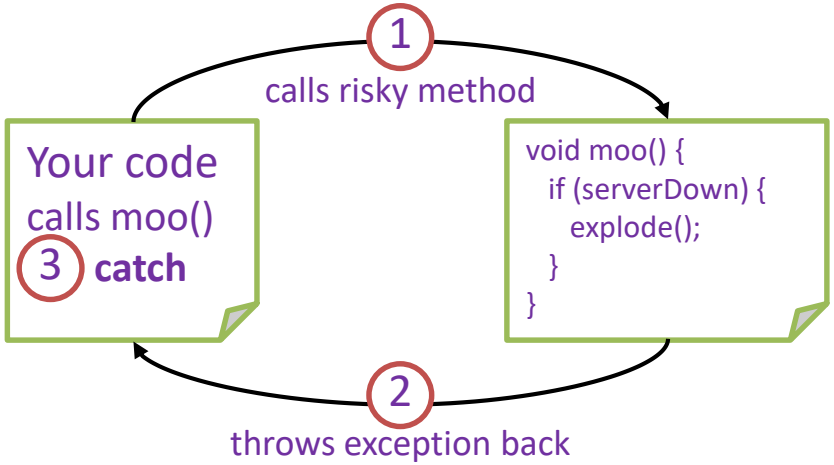
A new exception object is constructed, then thrown.

Most exception objects can be constructed with an error message.

This line is not executed when the exception is thrown.

Exception Handling (overview): Declaring, throwing, & catching exceptions

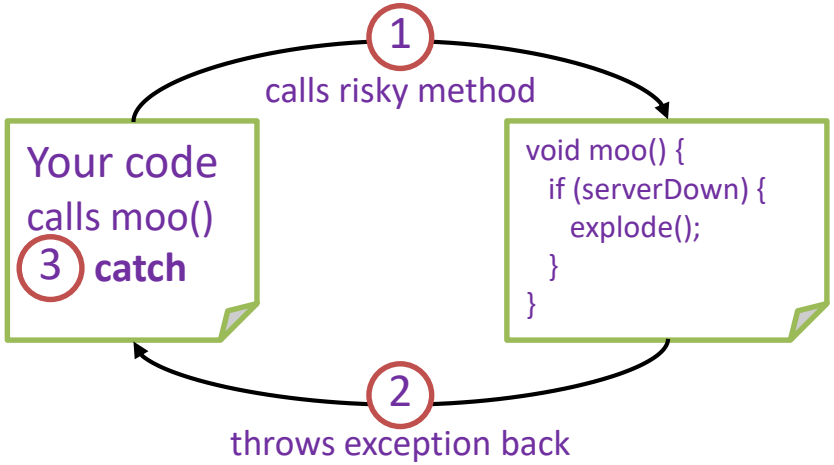
Remember: all exceptions must be detected and handled during coding to prevent halting the program



- An exception is always thrown back to the caller
- One method will catch what another method throws
- The method that throws has to declare that it will throw the exception

Exception Handling (overview): Declaring, throwing, & catching exceptions

In Java, you will implement a try-catch block




```
public void CrossFingers() {
   try {
1     anObject.takeRisk();
3  } catch (BadException ex) {
      System.out.println("Arghhh!");
   }
}
```

```
2 public void takeRisk() throws
                                BadException {
      if (abandonAllHope) {
         throw new BadException();
      }
   }
```

JVM Exceptions: How the JVM handles errors during run-time

- When the JVM detects an **exception**, Java
  - Creates an **Exception object** that has all the known information
  - Looks for and passes that object to the **best** known **exception handler**
    - Java "**throws**" the error/exception and the handler "**catches**" it
    - Execution continues with an exception handler)
  - If a handler is found, then it **terminates** execution immediately
    - Java "throws" the error/exception (but nobody "catches" it!)
- The JVM can detect *many* exceptions
  - We need to manually "catch" and handle them!
  - The JVM at runtime will find the best error handler to pass the error to

© Lisa F. Young/iStockphoto.

- When you throw an exception, the normal control flow is terminated. This is similar to a **circuit breaker** that cuts off the flow of electricity in a dangerous situation.

Hierarchy of Exception Classes

Figure: A Part of the Hierarchy of Exception Classes

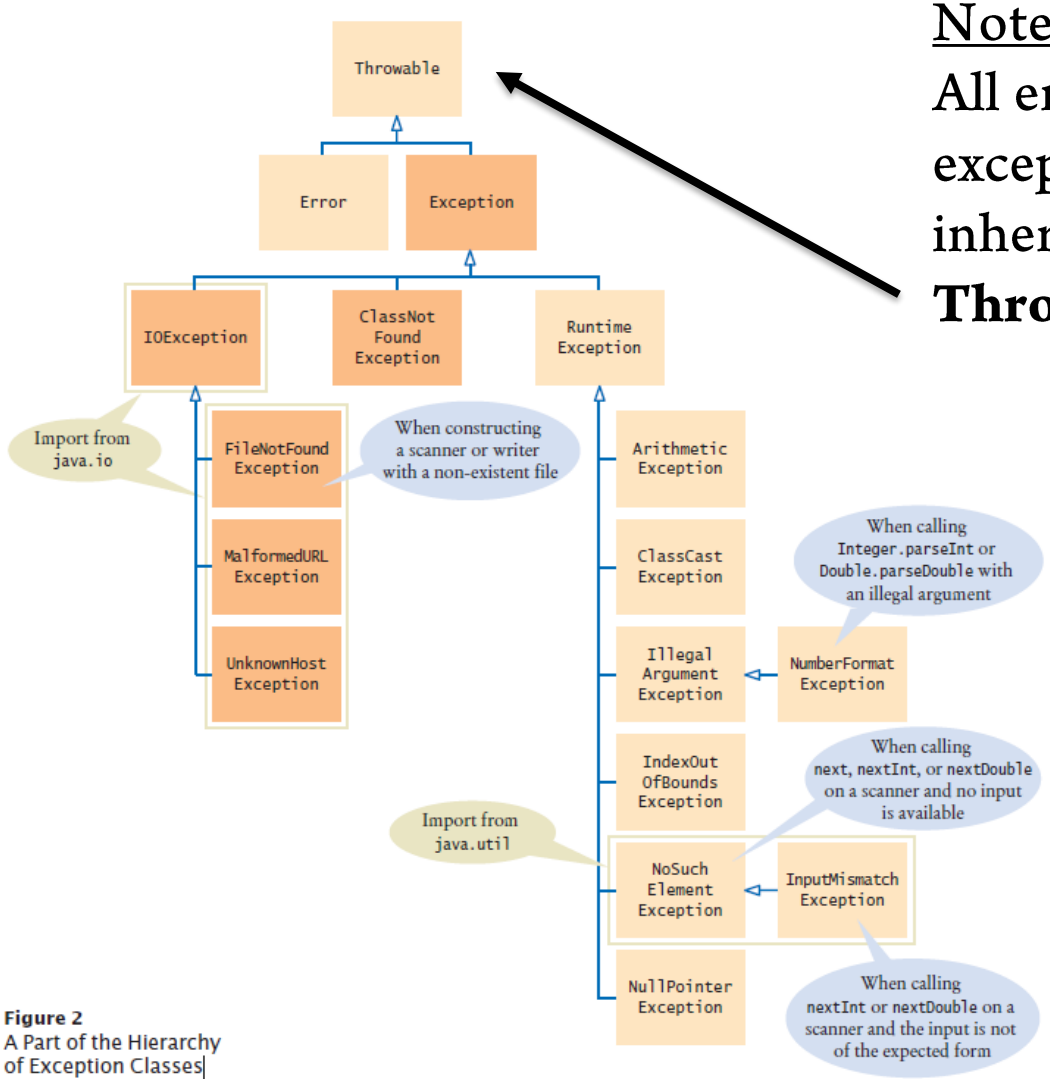
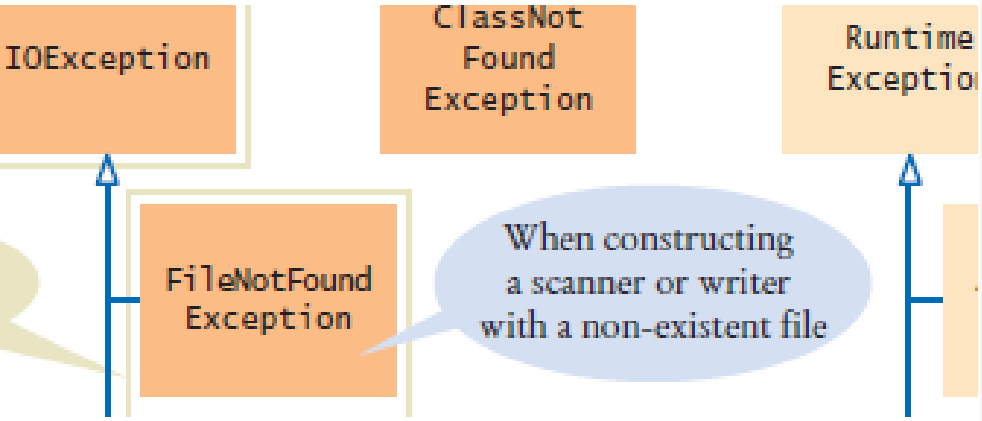


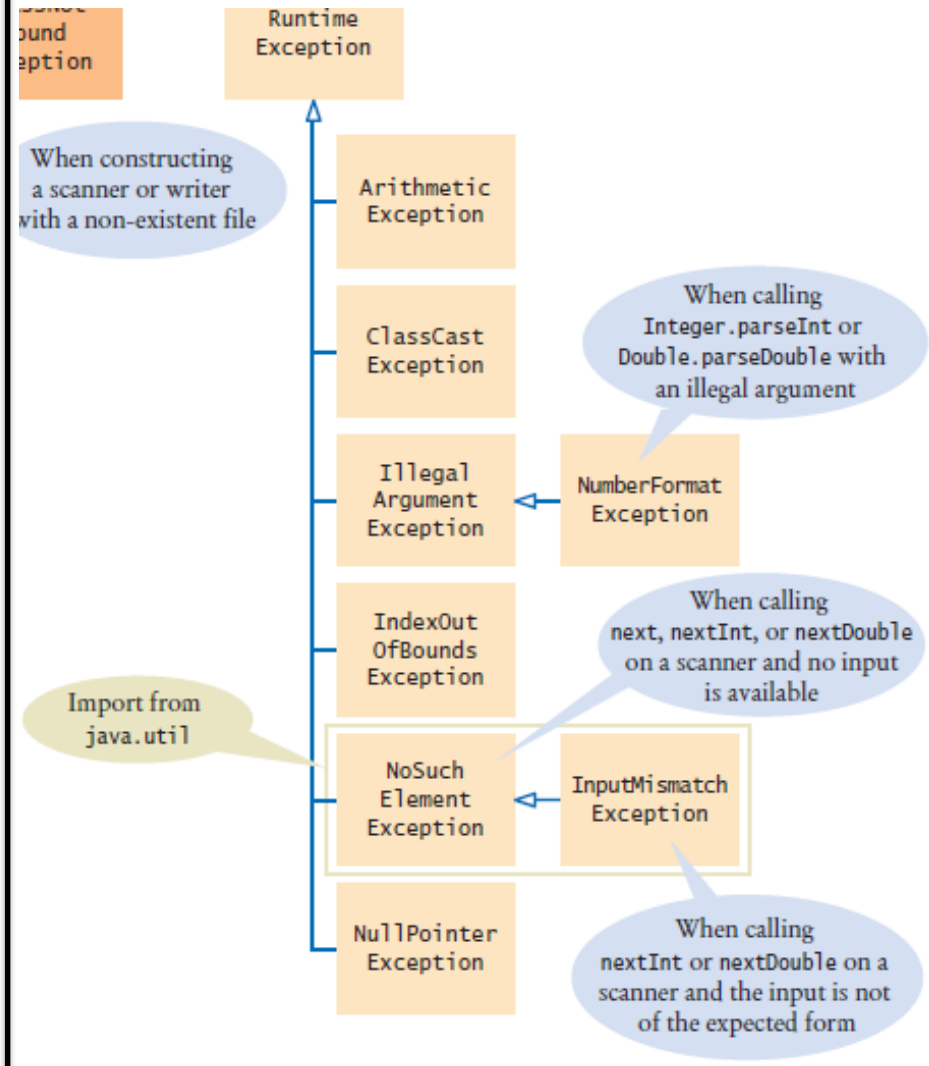
Figure 2
A Part of the Hierarchy of Exception Classes

<u>Note</u>:
All errors/ exceptions must inherit from **Throwable!**

Which means, the Throwable class is the superclass of all errors and exceptions in the Java language

Hierarchy of Exception Classes (closer look)





Common Exceptions

A few common exceptions that are encountered:

- IOException
  - FileNotFoundException
- RuntimeException
  - ArithmeticException
  - IllegalArgumentException
  - IndexOutOfBoundsException
  - NullPointerException
- Errors
  - OutOfMemoryError
  - AssertErrors (JUnit)

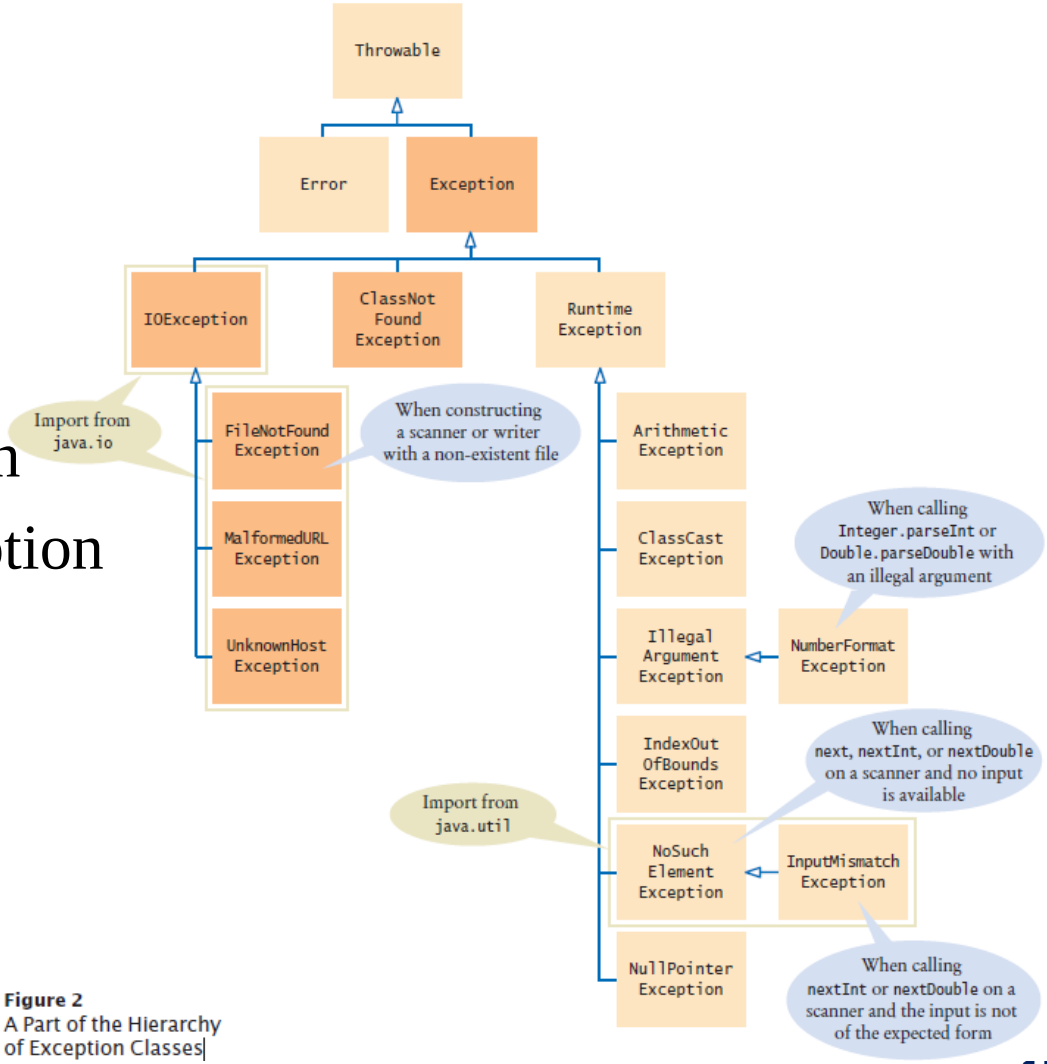


Figure 2
A Part of the Hierarchy of Exception Classes

Checked vs Unchecked Exceptions (& Errors)

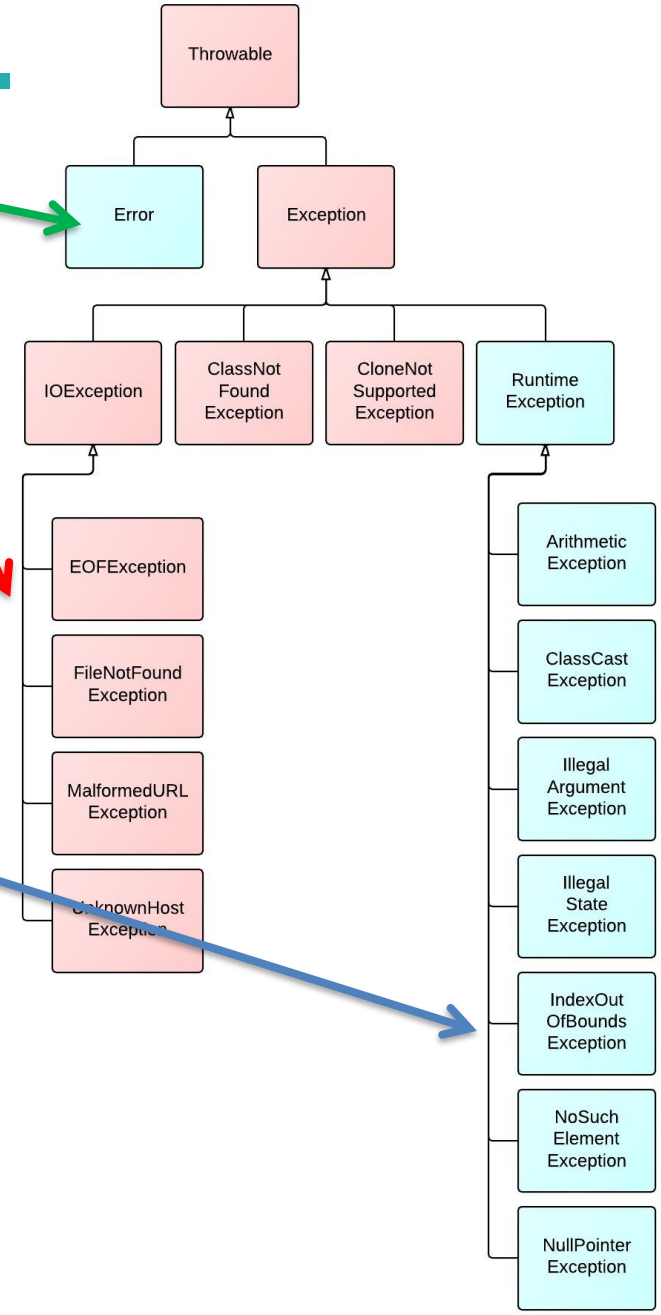
- Exceptions fall into **three categories**
- **Internal errors** are reported by descendants of the type `Error`.
  - Example: `OutOfMemoryError`
- **Unchecked exceptions**
  - Descendants of `RuntimeException`,
  - Example: `IndexOutOfBoundsException` or `IllegalArgumentException`
  - Indicate errors in your code.
- All other exceptions are **checked exceptions**.
  - (Any subclass of Throwable that is not also a subclass of either RuntimeException or Error)
  - Indicate that something has gone wrong for some ***external reason*** beyond your control
  - Example: `IOException`

Checked vs Unchecked Exceptions

- **Checked** exceptions are due to **external** circumstances that the programmer cannot prevent.
  - The compiler checks that your program handles these exceptions.

- The **unchecked** exceptions ***are your fault!*** ☺
  - The compiler does not check whether you handle an unchecked exception.
  - You are responsible for these exceptions!

- *Internal errors* - descendants of type error
- Red/pink colored are *checked* exceptions. Any checked exceptions that may be thrown in a method must either be **caught** or **declared** in the method's **throws** clause.
  - Checked exceptions must be caught at **compile time**. Checked exceptions are so called because both the Java compiler and the Java virtual machine check to make sure this rule is obeyed.
- Light blue colored are *uncheck* exceptions. They are exceptions that are not expected to be recovered, such as null pointer, divide by 0, etc.
  - You can still throw, catch, and declare them, but you don't have to and the compiler won't check!



Catching Exceptions (*try-catch*)

- Every exception should be handled somewhere in your program
- Place the statements that can cause an exception inside a **try** block, and the handler *(way you are handling the failure)* inside a **catch** clause.

```
try
{
   String filename = . . .;
   Scanner in = new Scanner(new File(filename));
   String input = in.next();
   int value = Integer.parseInt(input);
   . . .
}
catch (IOException exception)
{
   exception.printStackTrace();
}
catch (NumberFormatException exception)
{
   System.out.println(exception.getMessage());
}
```

Throwable methods

Catching Exceptions

- Three exceptions may be thrown in the **`try`** block:
  1. The `Scanner` constructor can throw a `FileNotFoundException`
  2. `Scanner.next` can throw a `NoSuchElementException`.
  3. `Integer.parseInt` can throw a `NumberFormatException`.

- If any of these exceptions is actually thrown, then the **rest** of the instructions in the **`try`** block are skipped.
  - A result of a thrown exception is like a "return" statement in a method (once the statement is reached, execution is transferred away and the rest of the statements are skipped)

```
try {
    String filename = . . .;
1   Scanner in = new Scanner(new File(filename));
2   String input = in.next();
3   int value = Integer.parseInt(input);
```

Finding the Best Match

- Exceptions are ***polymorphic***!
- Java requires that **catch blocks** are arranged in ***decreasing precision***:
  - Most specific ErrorType to least specific.
- This works because of the ***Substitution Principle of inheritance***.
  - A subclass (child) exception can be assigned to an Exception (parent) reference (*child can go anywhere it's parent is accepted*)
- Just because you can catch everything with one big polymorphic catch (e.g. the throwable "warehouse", doesn't mean that you should!

IOException


Exception


Throwable

Catching Exceptions – Example

```
try {
//code that should run
}

catch ( ExceptionType e) {
//specific error handling code
}

…

catch (Exception e) {
// default error handling code
}
```

From ***most*** specific to ***least*** specific e.g., **FileNotFoundException** is a sub type of **IOException**

```
try {
 Scanner scannerfile = new Scanner( new
                                File("file.txt"));
}


catch (FileNotFoundException e) {
 System.out.println("File not found");
}


catch (IOException e) {
 System.out.println("Error reading the file");
}


catch (Exception e) {
 System.out.println("An error occurred");
}
```

Catching Exceptions

- What happens when each exception is thrown:
- If a `FileNotFoundException` is thrown,
  - then the `catch` clause for the `IOException` is executed because `FileNotFoundException` is a descendant of `IOException`.
  - If you want to show the user a different message for a `FileNotFoundException`, you must place the `catch` clause before the clause for an `IOException`
- If a `NumberFormatException` occurs,
  - then the second `catch` clause is executed.
- A `NoSuchElementException` is not caught by any of the `catch` clauses.

Syntax Catching Exceptions

Syntax

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

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

This constructor can throw a `FileNotFoundException`.

This is the exception that was thrown.

When an `IOException` is thrown, execution resumes here.

A `FileNotFoundException` is a special case of an `IOException`.

Additional catch clauses can appear here. Place more specific exceptions before more general ones.

Catching Exceptions (*try-catch-finally*)

- **Finally** is another block you can add to the typical "try-catch" blocks
- The first match in a series of catch code blocks will be where the program exists. However, what if there is an error while a resource is in use?
  - The **finally** block will always execute – even if the program encounters an error

```
try {
    // open some files for exclusive
    // access and… do something risky
}
catch ( Exception e ) {
    // handle errors
}
finally {
    // close the files
}
```

Exceptions [Handling exceptions] - throws

(1) One of two ways to deal with exceptions: try-catch

- You can **handle the checked exception** in the same method that throws it

```
public void readData(String filename) {
    try
    {
        File inFile = new File(filename);
        Scanner in = new Scanner(inFile); //ThrowsFileNotFoundException
        . . .
    }
    catch (FileNotFoundException exception) // Exception caught here
    {
        . . .
    }
}
```

Here you have to deal with (handle) the exception that was caught, unless you want to…

Exceptions [Ducking Exceptions] - throws

(2) When you do not want to handle the exception: throws

- **Declare** the checked exception in the method header
  - You want the method to **terminate** if the exception occurs
  - Often the current method ***cannot handle the exception***.
  - Tell the compiler you are aware of the exception
- Add a ***throws*** clause to the method header

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

Exceptions [Ducking Exceptions] - throws

- The `throws` clause signals to the caller of your method that it ***may*** encounter a `FileNotFoundException`.
  - The caller must decide
    - To handle the exception
    - Or declare the exception may be thrown
- Throw early, catch late
  - **Throw** an exception as soon as a problem is detected.
  - **Catch** it only when the problem **can be handled**

- *Just as trucks with large or hazardous loads carry warning signs, the throws clause warns the caller that an exception may occur.*

Syntax throws Clause

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

```
public void readData(String filename)
      throws FileNotFoundException, NumberFormatException
```

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

You may also list unchecked exceptions.

Designing Your Own Exception Types

- ***You can design your own exception types*** — **subclasses** of `Exception` or `RuntimeException`.

- **E.g.:** Throw an `InsufficientFundsException` when the amount to withdraw from a bank account exceeds the current balance.

```
if (amount > balance)
{
   throw new InsufficientFundsException( "withdrawal of " +
      amount + " exceeds balance of " + balance);
}
```

- Make `InsufficientFundsException` an ***unchecked*** exception
  - Extend **`RuntimeException`** or one of its subclasses
  - Programmer could have avoided it by calling `getBalance` first

Designing Your Own Exception Types

- Supply **two constructors** for the class
  - A constructor with **no** arguments
  - A constructor that accepts a **message string** describing reason for exception

```
public class InsufficientFundsException extends RuntimeException
{
   public InsufficientFundsException() {} // constructor – no arguments
   public InsufficientFundsException(String message) // with message
   {
      super(message);
   }
}
```

- When the exception is caught, its message string can be retrieved
  - Using the **getMessage** method of the **Throwable** class.

Exception Rules
(Key Points about Handling and Declaring Exceptions)

1. You *cannot* have a catch or finally without a try

```
public void foo() {
   Foo f = new Foo();
   f.foof();
   catch(FooException ex) {}
}
```

2. You cannot put code between the try & catch

```
try {
   x.doStuff();
}
int y = 40;
} catch(Exception ex) {}
```

3. A try MUST be followed by a catch or a finally, but still declare an exception if no catch

```
void go() throws FooException {
   try {
      x.doStuff();
   }
   finally {// cleanup}
}
```

4. Multiple exceptions can be declared in a method after the throws keyword (incl. unchecked)

```
void go() throws ArithmeticException,
       FileNotFoundException {
   Foo f = new Foo();
   f.foof();
   // do more risky stuff
}
```

In-Class Activity: Handling Input Errors

- 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
```

- ***What can go wrong?***
  - File might not exist
  - File might have data in wrong format

In-Class Activity: Add Catch Statements

- Download **ReadInData.java**, Test2.txt, Test3.txt, Test4.txt files from Collab.
- Find the **readData()** method.
- Inside this method is a try block without any catch statements.
- **Add the following 5 catch statements to this file**
  (Hint: put them in the right order)
  - `RuntimeException`
  - `Exception`
  - `FileNotFoundException`
  - `NumberFormatException`
  - `InputMismatchException`
- For each of these print out a **short message** describing the exception that occurred.
- **Run** the main method and make sure you *understand* why the output is the way it is.
- **Submission**: Upload your **ReadInData.java** file to **Collab**.

Activity: Other things to think about:

- ***Who can detect the faults?***
  - `Scanner` 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 `Scanner` constructor
  - `BadDataException`, a custom checked exception class for reporting wrong data format

Activity: Other things to think about:

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