



# CS 2100: Data Structures & Algorithms 1

## Concurrency

Aside: Exception Handling

Dr. Nada Basit // [basit@virginia.edu](mailto:basit@virginia.edu)

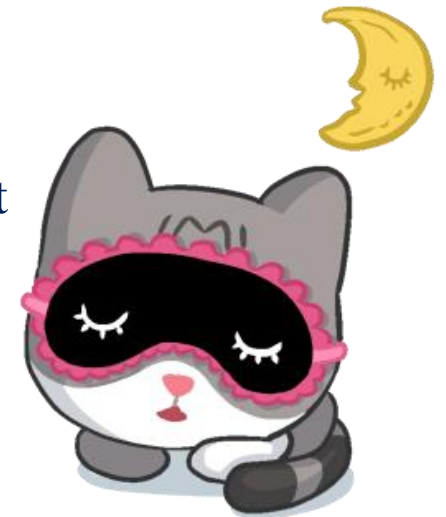
Spring 2022



# Friendly Reminders

---

- The University updated the mask policy. As per my Request on Mar 28, 2022 (see Collab), I would greatly appreciate if you would do me a kind favor by **continuing to wear your masks** in CS 2100 (Ridley G008). I know it is a lot to ask, and it is **voluntary**, but I appreciate your understanding.
- If you forget your mask (or mask is lost/broken), I have a few available
  - **Just come up to me at the start of class and ask!**
- No eating or drinking in the classroom, please
- Our lectures will be **recorded** (see Collab) – please allow 24-48 hrs to post
- If you feel **unwell**, or think you are, **please stay home**
  - *We will work with you!*
  - At home: eye mask instead! **Get some rest** 😊



# Announcements

---

- **Final Exam:**

- **Date:** Saturday, May 7, 2022
- **Time/Duration:** 7:00pm – 9:00pm ET (two hours)
- **Location:** TBD (Registrar will confirm rooms, will let you know soon)

- **Make-up Exam:** [Email me if you haven't already]

- If you have a conflict with the following courses, **email me:**

- APMA 3100
- APMA 3140
- ECON 2020 (sections 001 and 002 only)

- **Make-up Date:** Sunday, May 8, 2022

- *At this time we do not have a time or a location; however, given there are no officially held final exams on this day (May 8) we anticipate the chosen time will suit your schedule*

# The Final Exam – Saturday, May 7 (Make-up: May 8)

- Mode: Taken **in-person**
- Duration: **two (2) hours**
- Policies:
  - Closed-book / Closed-notes
  - Closed-Google/Internet (except to access the quiz itself)
  - Closed-Eclipse/other IDE
  - Closed-friend/any other person
  - Closed... everything 😊
  - **Can retake as many quizzes as you want**
  - The work you do must represent your **individual effort**, and involve **no outside assistance** from any one or any resource
- Location of Quizzes: ONLINE AS BEFORE. **Explicit instructions will be given on the day!**
- Students with accommodations with SDAC:
  - Please see email that I have sent to you.
  - If you choose to book a testing appointment with SDAC, please do so as soon as possible!
  - You will have your extended time accommodations

## ➤ What to bring with you to the final exam:

- **Fully charged laptop (+ charging cable)**
- Pen/pencil to write on scratch paper (*not necessary, only if you want*)
- Student ID card

---

# An Aside: Introduction to Exceptions

We need to talk about exceptions because when writing multithreaded programs, these programs often throw exceptions that need to be handled.

To understand this, we will discuss exceptions today!

# Terminating Threads

---

- A thread terminates when the **run()** method is complete
- Or, you can call:
  - `t.interrupt();` // notifies the thread that it should terminate
- Does **not** stop the thread (immediately), rather it just sets a **boolean**
  - The run method should **check** for this interrupt periodically

```
public void run(){
    for(int i=0; i<=REPETITIONS && !Thread.interrupted(); i++){
        //DO STUFF
    }
    //Clean up if necessary
}
```

# Terminating Threads

---

```
public void run(){
    for(int i=0; i<=REPETITIONS && !Thread.interrupted(); i++){
        //DO STUFF
    }
    //Clean up if necessary
}
```

- To suspend execution of a thread, you can call: `sleep()`
  - If a thread is **sleeping** at the time it is **interrupted...** the thread is **not awake** to check `Thread.interrupted()` condition!
  - This is generally NOT a good setup to use
- To better understand how to proceed we have to detour and speak about **EXCEPTIONS!**

# Motivation

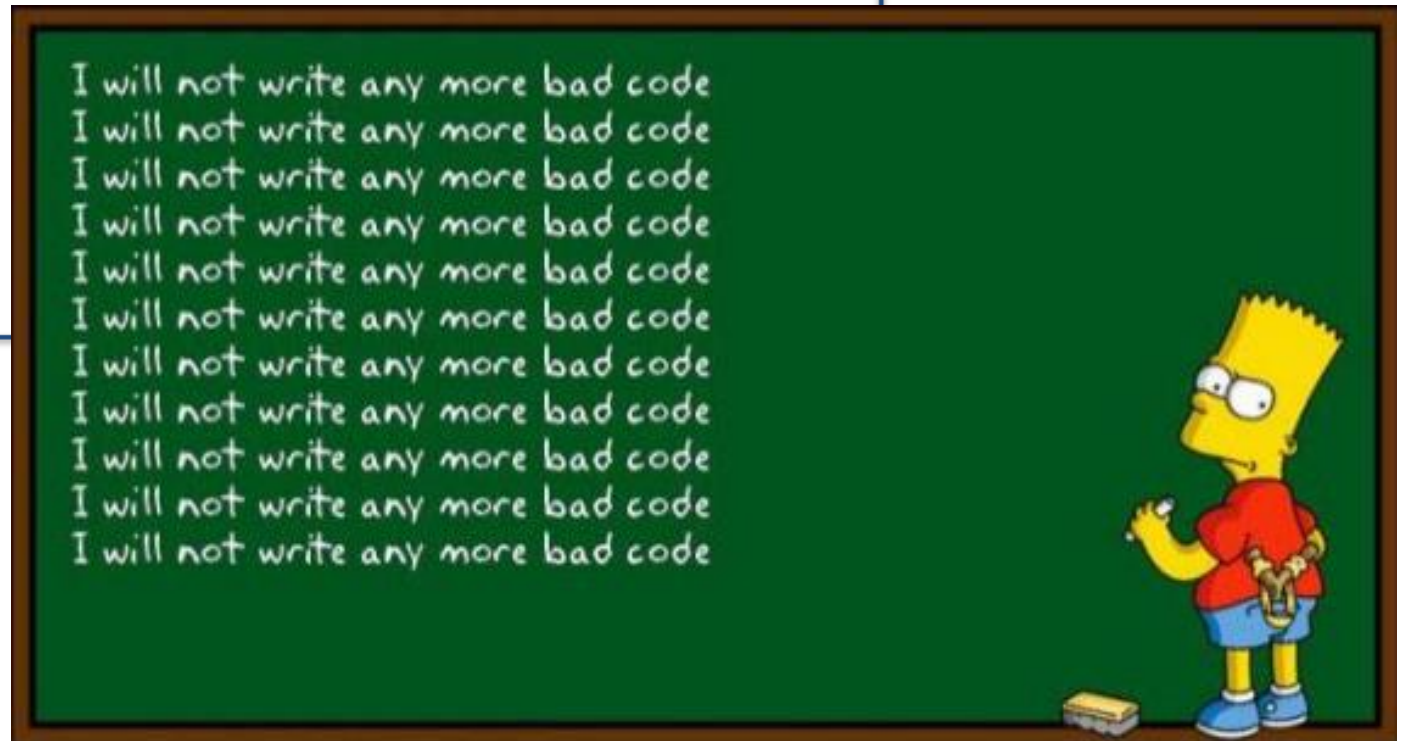
---

- **Problem:** How do we deal with errors in code
  - e.g., you divide a parameter by another parameter, what if the invoker gave you 0 as a divisor?
  - e.g., what do you do if a method tries to go off the end of an array
- **Solution 1:** Ignore the method call, or return a dummy value (null).
- **Java Solution:** Exceptions



# Exceptions

- Exceptions are events that disrupt the intended program flow
- **Exception** is short for “exceptional events”
- All exceptions must be:
  - Detected
  - Handledduring coding to **prevent halting the program** (*with no explanation*)
- In Java, exceptions are objects that are created when an **error** or problem has been **detected**
  - E.g. **accessing a null pointer** or **going off the end of an array**



# Exceptions

- Java is Safe! By default, the Java approach to handling errors is to end the program
  - **Terminal** error handling approach
- Aside: Resumptive error handling approach is pretty rare

```
java.lang.NullPointerException  
    at PhotographContainer.addPhoto (PhotographContainer.java:36)  
    at PhotoLibrary.main (PhotoLibrary.java:165)
```



# Exception Handling and Defensive Driving!

---

- If you'd like to use a “buzzword” term, use “**defensive programming.**”
- “**Defensive driving**” is when you drive expecting people around you to **be bad drivers**, so you pay attention to what they **COULD do wrong**, that way you're ready to deal with it if/when it happens. “**Defensive Programming**” is the same idea!
- Defend yourself against **bad coders/users!** 😊



# 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**

## 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.

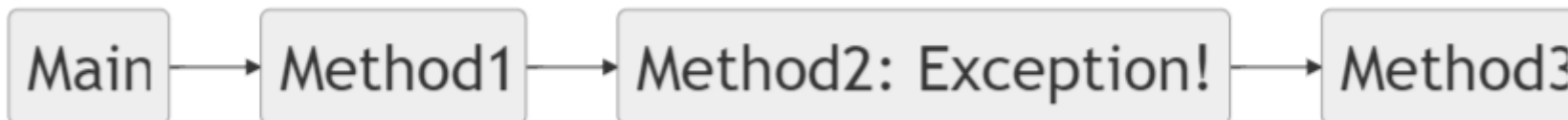
Most exception objects can be constructed with an error message.

This line is not executed when the exception is thrown.

# Exceptions

---

- When an exception is created, there are **two ways to handle it**:
  - **THROW**: this means that the current method will NOT deal with the exception. The exception is thrown to the method that invoked this one. Then, *that* method can either throw or catch it.
  - **CATCH**: This means that you deal with the exception right now. Crash the program, spit out an error message, ignore the exception, etc.
- In the example below, Method 2 can **either catch the exception or throw it** to Method 1 to deal with:



# Example Exceptions

---

- Here are some exceptions that can be generated in Java

```
/* NullPointerException */  
Object o1 = null;  
System.out.println(o1.toString()); // !!!
```

```
/* ArrayIndexOutOfBoundsException */  
int[] arr = new int[10];  
arr[12] = 5; // !!!
```

```
java.lang.NullPointerException: Cannot invoke "Object.toString()" because "o1" is null  
at Test1.main(Test1.java:27)
```

# JVM Exceptions:

## How the JVM handles errors during run-time

---

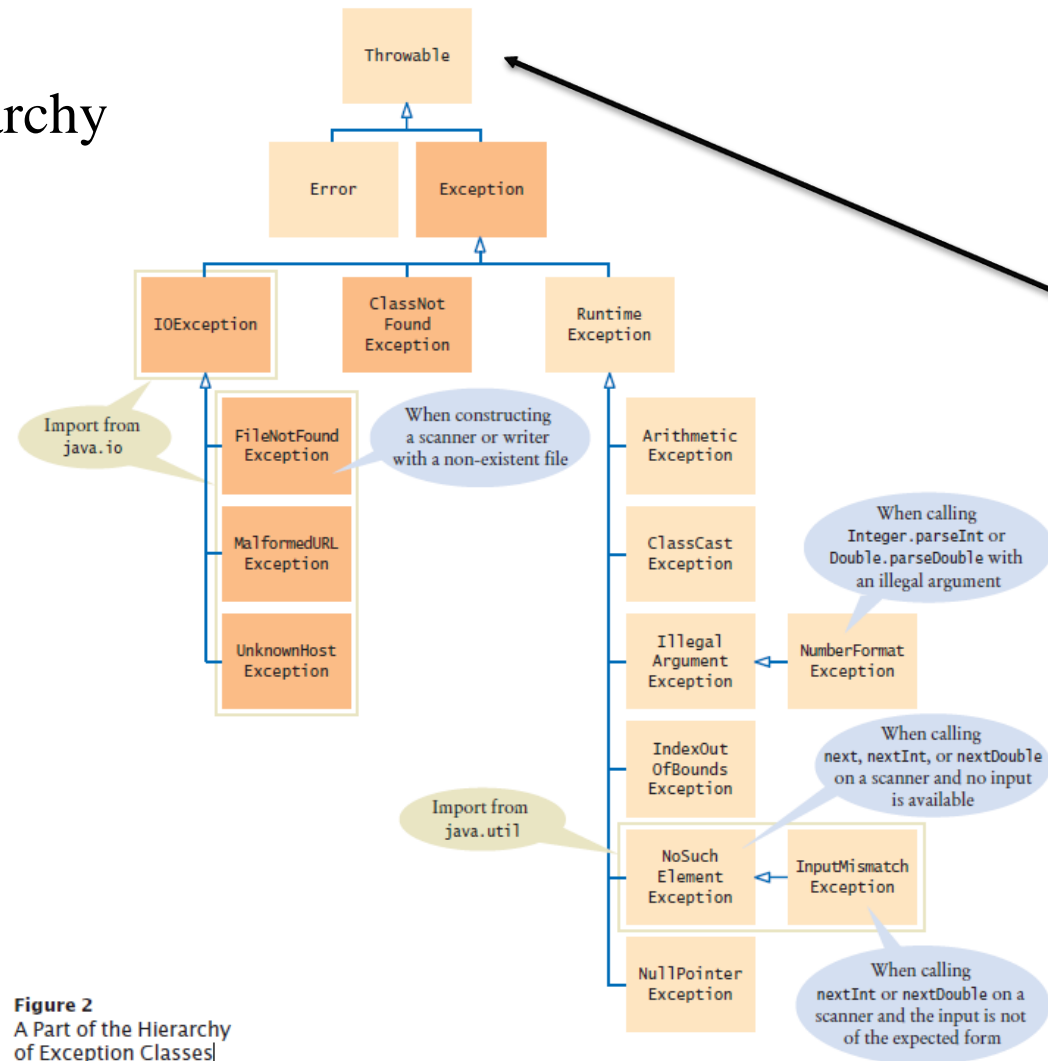
- When the JVM detects an **exception (error)**, Java
  - Creates an **Exception (error) object** that has all the known information
  - Looks for and passes that object to the **best known exception handler**
    - Java “**throws**” the error/exception and the handler “**catches**” it
    - (Execution 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* errors
  - We need to manually “catch” and handle them!
  - The JVM at runtime will find **the best error handler to pass the error to**
- 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.*



© Lisa F. Young/iStockphoto.

# Hierarchy of Exception Classes

**Figure:** A Part of the Hierarchy of Exception Classes



**Figure 2**  
A Part of the Hierarchy of Exception Classes

Note:

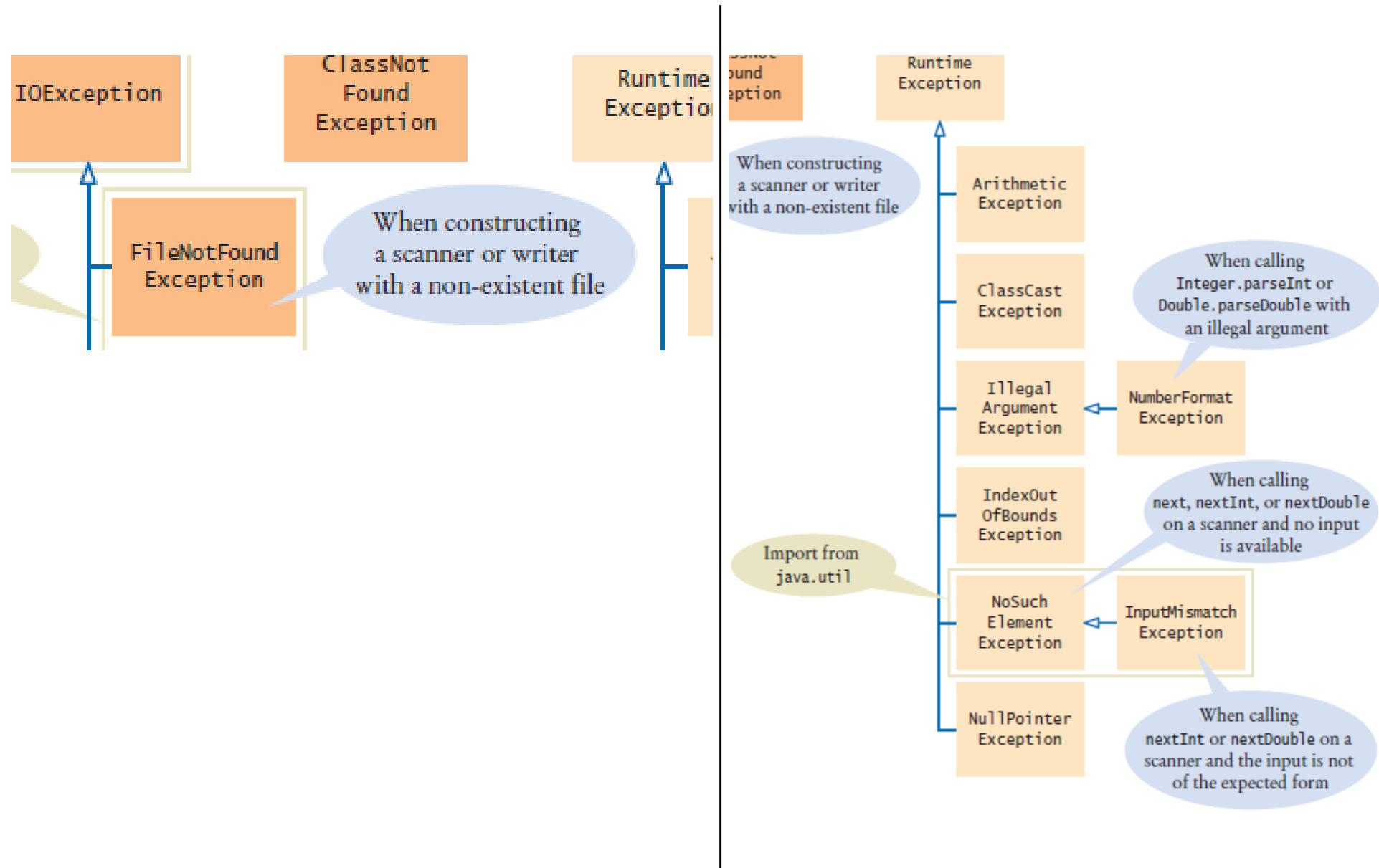
All errors/  
exceptions must  
inherit from

**Throwable!**

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



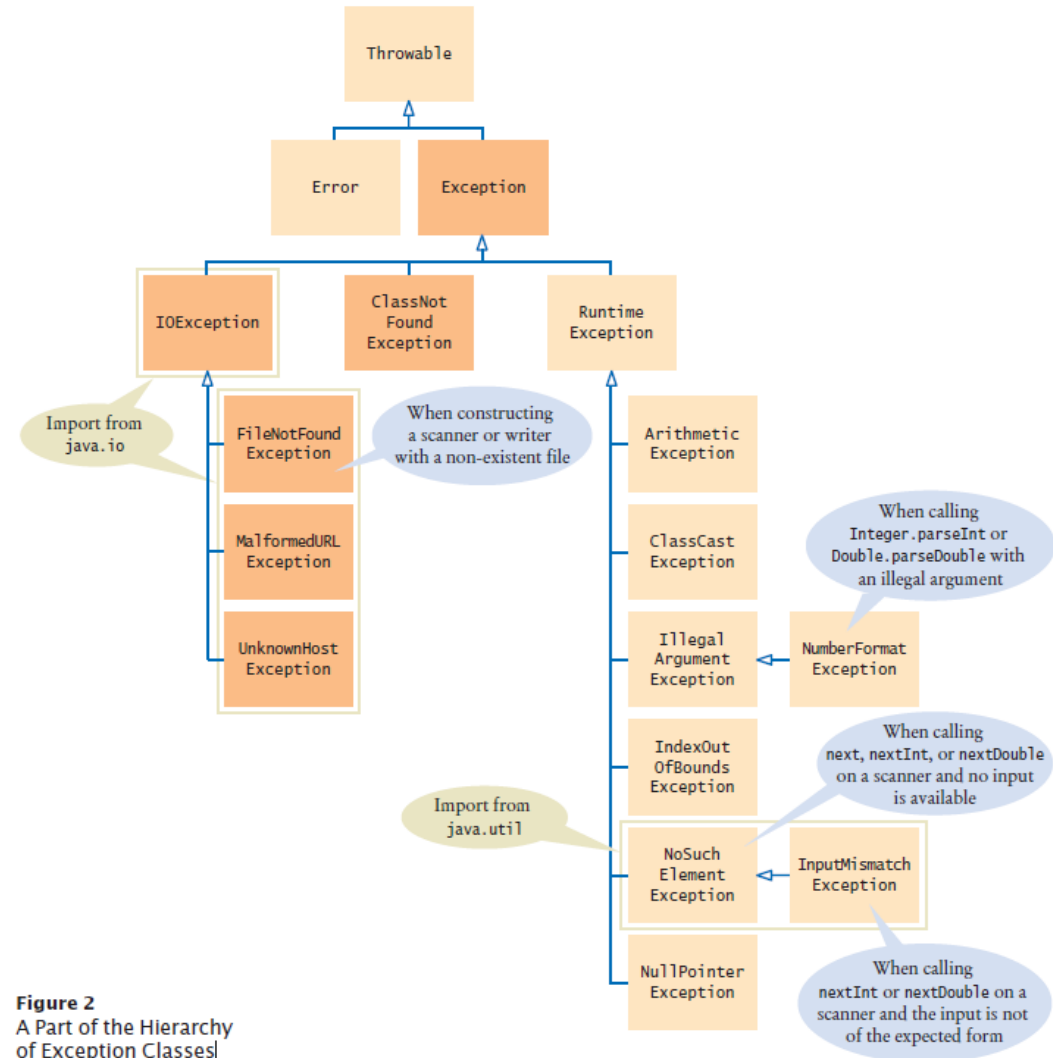
# Hierarchy of Exception Classes (closer look)



# Common Exceptions

A few common exceptions that are encountered:

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

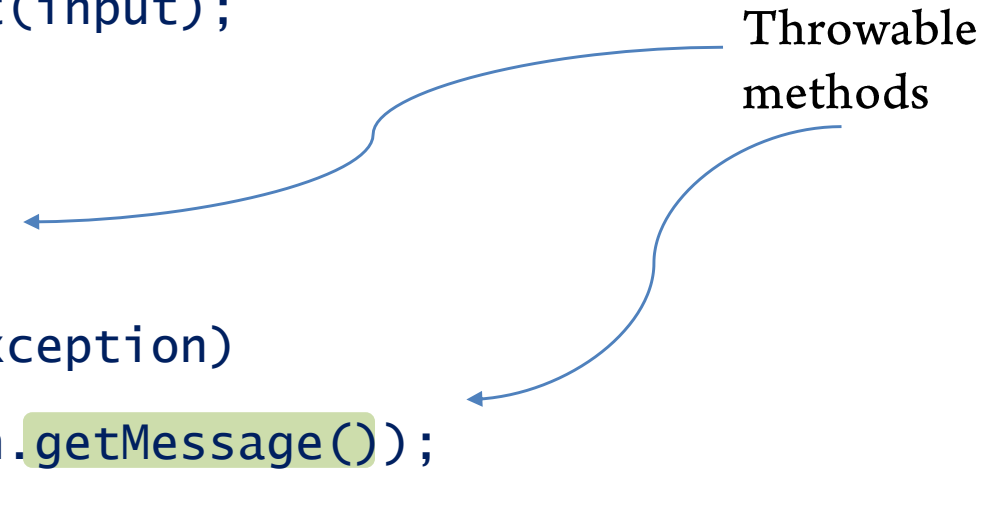


# Catching Exceptions (*try-catch*)

- 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 - 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");  
}
```

# Syntax Catching Exceptions

*Syntax*

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

This constructor can throw a `FileNotFoundException`.

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

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

```
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 is the exception that was thrown.

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

# 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  
}
```

# Custom Exceptions: EmptyStackException

---

- When popping off an empty stack, let's throw an exception

```
/* Create our own exception class */  
/* Don't worry much about extends, and other things here */  
public class EmptyStackException extends Exception{  
  
}
```

```
public class Stack<T>{  
    private LinkedList<T> theStack;  
  
    public T pop() throws EmptyStackException{  
        if(theStack.size == 0)  
            throw new EmptyStackException();  
  
        return theStack.removeFirst();  
    }  
}
```

# Custom Exceptions: EmptyStackException

---

- This code now **throws an error!** =====>
  - [From the previous slide we see that the **pop()** method throws an EmptyStackException exception!]
  - Must **deal** with the potential exception that gets thrown

```
Stack<Integer> s = new Stack<Integer>();  
  
public void someMethod(){  
    /* Some code here */  
  
    s.pop();  
}
```

```
//FIX 1: Throw method up another level  
public void someMethod() throws EmptyStackException{  
    /* Some code here */  
  
    s.pop();  
}
```



# Custom Exceptions: EmptyStackException

- This code now **throws an error!** Must **deal** with the potential exception that gets thrown

```
/* FIX 2: Deal with the error NOW! */
public void someMethod() {
    /* some code here */

    try {
        s.pop();
    }
    catch(EmptyStackException e) {
        /* You can do anything here */
        e.printStackTrace();
        System.exit(1);
    }
    finally {
        // This code will run regardless if
        // exception is thrown or not
    }
}
```



# Which Fix To Use?

---

- Which fix should I use? Matter of code design / style
- Generally:
  - If error NOT your fault and not fatal, then give **invoking method** a chance to react to it by **throwing**
  - If fatal, then **catch and crash** (e.g., null pointer exceptions do this)

# I Thought We Were Talking About Threads?!



- Yes, but Java's concurrency libraries **throw a lot of exceptions**, so you will often need to use **try/catch statements** to handle those.

- **REMEMBER...**

```
public void run(){
    for(int i=0; i<=REPETITIONS && !Thread.interrupted(); i++){
        //DO STUFF
    }
    //Clean up if necessary
}
```

- **NOW IT WILL BE:**

```
public void run(){
    try{
        for(int i=0; i<=REPETITIONS; i++){
            //DO STUFF
            sleep(1000) //thread waits. but might throw exception
        }
    }
    catch(InterruptedException e){
        //Do something if you want here
    }
    finally{
        //clean everything up
    }
}
```

---

# Key Points About Handling and Declaring Exceptions

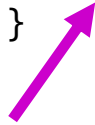
SOME IMPORTANT NOTES ABOUT TRY, CATCH, AND FINALLY!

# 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.foo();  
    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.foo();  
    // do more risky stuff  
}
```

