CS 112 – MiraCosta College

Introduction to Computer Science II
Java

Module 4 – Exception Handling

Chris Merrill
Computer Science Dept
cmerrill@miracosta.edu

Agenda

- Review
  - Quiz #1
  - Homework, Module 2
  - Polymorphism & Abstract Classes
- Stacks
- Exception Handling
- Writing Exception Classes
- Lab 4 – Exceptions
- Quiz 2 – Exceptions (next week)

Stacks

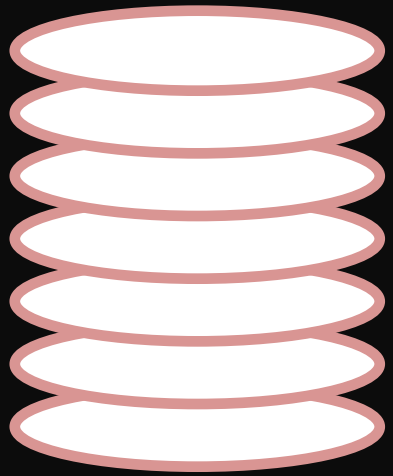
Computer systems often use two types of data structures called *stacks* and *queues*.

The most common analogy for a stack is a stack of plates on a spring at a buffet line

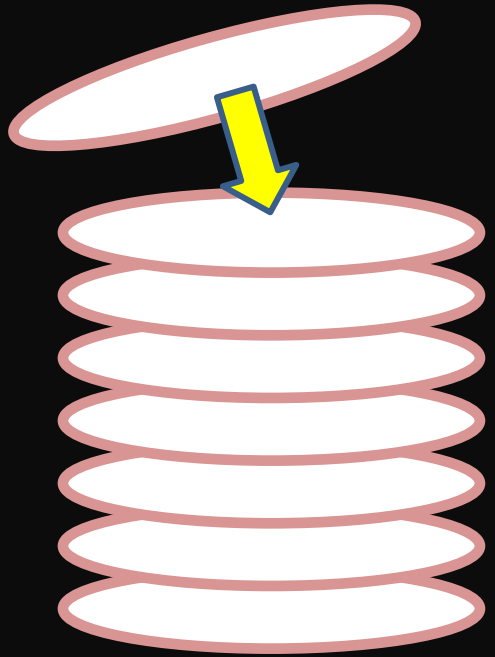
- As plates are washed and dried, they are placed on the *top* of the stack
- When hungry customers takes a clean plate, they take plates from the *top* of the stack

Stack Example

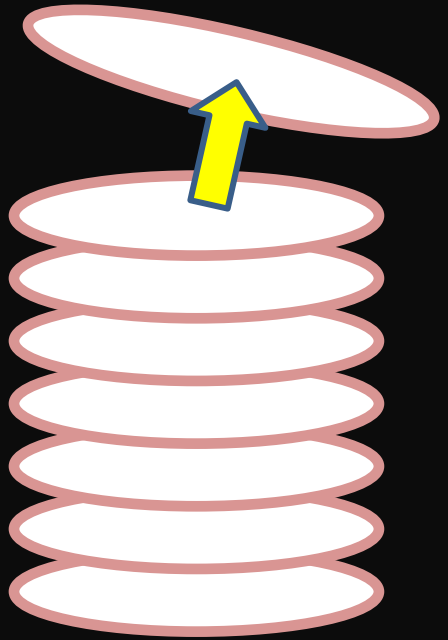
Stack of Plates



Clean Plates
Goes on *Top*



Hungry Customers
Take New Plates
From the *Top*



Queues

Queues work from opposite sides. A popular example is people waiting in line

- As people enter the queue, they go to the *back* of the line
- When people are serviced, they are taken from the *front* of the line

In fact, “lines” are called “queues” in the UK

Queue Example

Exit Here

Enter Here

Stacks and Queues

Since the last (or most recent) plate on a stack is the first one out, stacks are often referred to as "last-in-first-out" structures, or

LIFO

Since the first (or oldest) person in a queue is the first one out, queues are often called "first-in-first-out" structures, or

FIFO

Stacks Used In Programs

Windows Explorer – going forward, then back to the last folder or folders

Browsers – going forward, then back to the next page or pages

Java programs – invoking, then returning from a method or methods

Introduction to Exception Handling

- The best outcome we can ever hope for is when nothing unusual happens
- However, we need to deal with exceptional things in computer science, and the more of them we handle, the better the user's experience will be
  - Java exception handling facilities are used when the invocation of a method may cause something exceptional to occur
  - Often the exception is some type of error condition

Introduction to Exception Handling

Both Java's library software and code written by outside programmers provide a mechanism which signals when something unusual happens

- This is called *throwing an exception*
- In another place in the program, the programmer must provide code that deals with the exceptional case
  - This is called *handling* or *catching the exception*

`try-throw-catch` Mechanism

The basic way of handling exceptions in Java consists of the **`try-throw-catch`** trio

- Sometimes a **`finally`** clause is added
- The **`try`** block contains the code for the basic algorithm
  - It tells Java what to do when everything goes smoothly, no problems or issues

`try-throw-catch` Mechanism

- It is called a **`try`** block because it "tries" to execute the case where all goes as planned
  - It can also contain code that *throws an exception* if something unusual happens

```
try {
    (CodeThatMayThrowAnException)
}
```

`try-throw-catch` Mechanism

Examples of code that can throw exceptions:

- Converting a `String` to a number which is invalid
- Dividing a number by 0
- Accessing an array element that doesn't exist
- Opening a file for reading that doesn't exist
- Closing a stream that is not open
- Accessing a method using a calling object variable set to `null`

`try-throw-catch` Mechanism

- In addition, an exception can be thrown explicitly by using the **`throw`** statement:

```
throw new FileNotFoundException();
```

- The value thrown is the argument to the **`throw`** operator, and is always an object of some exception class. The execution of a throw statement is called *throwing an exception.*

try-throw-catch Mechanism

A **throw** statement is similar to a method call:

```
throw new <ExceptionClassName>(aString);
```

- In the above example, the object of class ***ExceptionClassName*** is created using **aString** as its argument
- This object, which is an argument to the **throw** operator, is the exception object thrown
- Instead of calling a method, a **throw** statement calls a **catch** block

`try-throw-catch` Mechanism

When an exception is thrown, the **`catch`** block begins

- The **`catch`** block has one parameter
- The exception object thrown is plugged in for the **`catch`** block parameter
- The execution of the **`catch`** block is called *catching the exception*, or *handling the exception*
  - Whenever an exception is thrown, it should ultimately be handled (or caught) by a **`catch`** block

`try-throw-catch` Mechanism

A **`catch`** block looks like a method definition that has a parameter of type ***`<ExceptionName>`*** class

```
catch(<ExceptionName> e) {
    (Exception Handling Code)
}
```

It is not really a method definition, though, as **`catch`** is a Java keyword.

`try-throw-catch` Mechanism

A **`catch`** block is a separate piece of code that is executed when a program encounters a problem or when executing a **`throw`** statement in a preceding **`try`** block

A **`catch`** block is often referred to as an *exception handler*

The identifier **`e`** in the **`catch`** block heading is called the **`catch`** block parameter

```
catch(<ExceptionName> e) { … }
```

`try-throw-catch` Mechanism

The `catch` block parameter does two things:

1. It specifies the type of thrown exception object that the `catch` block can catch (e.g., an `Exception` class object above)
2. It provides a name for the thrown object that is caught, and on which it can operate in the `catch` block
   - Note: The identifier `e` is used by convention, but any non-keyword identifier can be used

Use of `e` for the `catch` Parameter

My dislike of single-letter variable names is well-known.

However, it is *customary* to use `e` as the parameter for a `catch` block.

`try-throw-catch` Mechanism

When a `try` block is entered, two things can happen:

1. No exception is thrown in the `try` block
   - The code in the `try` block is executed to the end of the block
   - The `catch` block is skipped
   - The execution continues with the code placed after the `catch` block

`try-throw-catch` Mechanism

2. An exception is thrown in the `try` block and caught in a `catch` block
   - The rest of the code in the `try` block is skipped
   - Control is transferred to a `catch` block
   - The thrown object is plugged in for the `catch` block parameter
   - The code in the `catch` block is executed
   - The code that follows the `catch` block is executed

Checking for Division by Zero

We can use an **`ArithmeticException`** to catch a divide-by-zero condition when using a numerator and denominator both of type **`int`**

(Note what happens when we change the divisor to a **`double`**!)

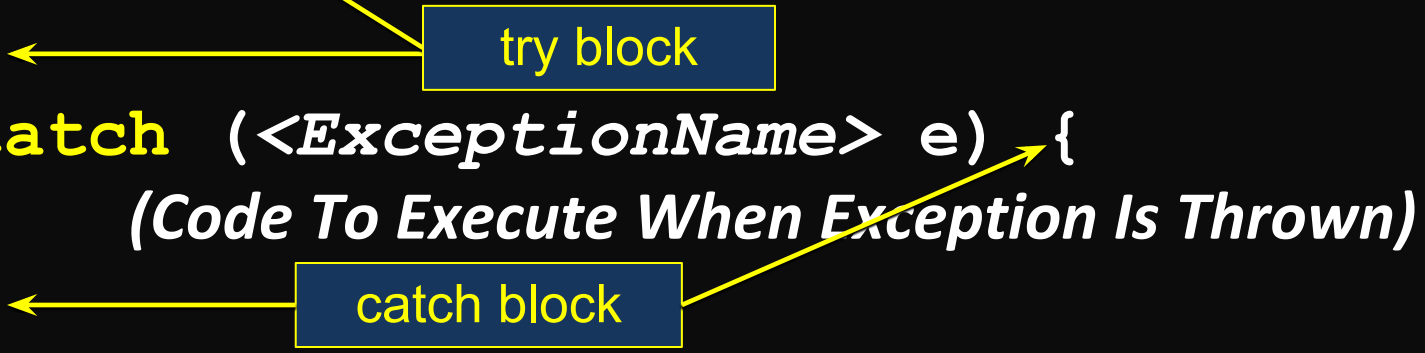
(ArithmeticExceptionDemo)

`try-throw-catch` Mechanism

When an exception is thrown, the execution of the surrounding try block is stopped

- Normally, the flow of control is transferred to another portion of code known as the `catch` block

```
try {
      (Code That May Throw An Exception)
}
catch (<ExceptionName> e) {
      (Code To Execute When Exception Is Thrown)
}
```



Alternate Style for `catch` Block

In a prior slide, the `catch` block starts on its own line, so that the closing `}` for the `try` block is on its own line

Another common style is to put `catch` on the same line

```
try {
    (Code That May Throw An Exception)
} catch (<ExceptionName> e) {
    (Code To Execute When Exception Is Thrown)
}
```

moved up onto prior line

I *prefer* separate lines, but either style is fine.

`ArrayIndexOutOfBoundsException`

This exception is thrown whenever a program attempts to use an array index that is either less than 0 or else greater than or equal to the size of the array

- Like all other descendants of the class **`RuntimeException`**, it is an unchecked exception
  - Therefore, there is no requirement to handle it
  - If it isn't handled, the program will end

`ArrayIndexOutOfBoundsException`

When this exception is thrown, it is an indication that the program contains a *logic* error

- Instead of attempting to handle the exception, the program should simply be fixed
- Note that the compiler *never* checks for valid index values

`ArrayIndexOutOfBoundsException` Demo

Some examples of this Exception,

including one that you might think would

be caught by the Java compiler…

IndexOutOfBoundsDemo

Mini-Lab #1

Modify the program just demonstrated

(**`ArrayOutOfBounds2`**) to catch the exception

rather than just stopping

(See instructions on Canvas)

Exception Example

In many cases your own code won't throw an exception, but one *is* thrown by a method in the Java library

Example: entering an integer using **`nextInt()`**

- What if the user doesn't enter a valid integer?
- Then when Java invokes the **`nextInt`** method, the method will stop executing the try block and throw an **`InputMismatchException`**

Exception Handling with the `Scanner` Class

If a user enters something other than a well-formed `int` value, an `InputMismatchException` will be thrown

- Unless this exception is caught, the program will end with an error message (i.e, it is *unchecked**)
- If the exception is caught, the `catch` block can give code for some alternative action, such as asking the user to reenter the input

The `InputMismatchException`

The `InputMismatchException` is in `java.util`, so any program throwing it must import it:

```
import java.util.InputMismatchException;
```

- It is a descendent class of `RuntimeException`
  - Therefore, it is an unchecked* exception and does not have to be caught in a `catch` block or declared in a `throws` clause
  - However, catching it in a `catch` block is allowed and usually preferable

* Unchecked Exceptions

You have probably used methods which *could* throw an exception, but didn't appear in a try-catch block in your code

- For example, the **`nextInt`** and **`nextDouble`** methods in the **`Scanner`** class, and **`parseInt`** in the **`Integer`** class
- These exceptions are *unchecked:*
  - If an exception is thrown and not caught, then the default exception handler is invoked

Checked Exceptions

Conversely, we also have used Java methods that are *required* to be in a try-catch block

- For example, trying to open an InputStream to a file (which might not exist), or trying to close a stream (which might not be open)
- These exceptions are *checked:*
  - If an exception is *can* be thrown, then code *must* exist which handles the exception.
- More on this later…

Exception–Controlled Loops

Sometimes it is better to simply loop through an action over and over again when an exception is thrown, as follows:

```
boolean done = false;
while (!done) {
    try {
        (Code That May Throw An Exception)
        done = true;
    }
    catch (SomeExceptionClass e) {
        (Some More Code)
    }
}
```

Exception–Controlled Loop Demo

Let's look at two slightly different looping methods for prompting the user for a number without exiting the program.

ExceptionLoops

Exception Classes

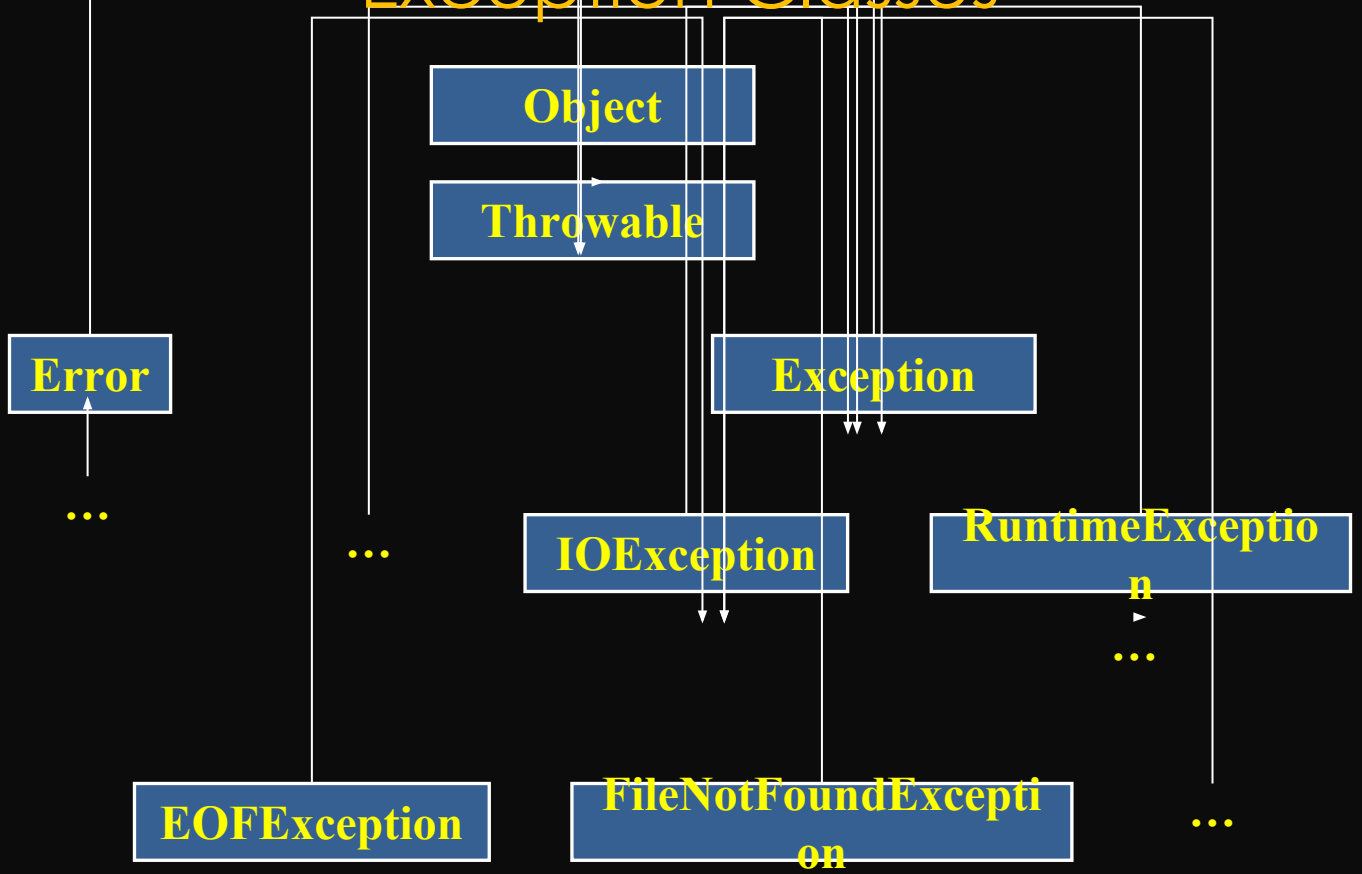
An exception is an object.

- Exception objects are created from classes in the Java API hierarchy of exception classes.
- All of the exception classes in the hierarchy are derived from the **`Throwable`** class.
- **`Error`** and **`Exception`** are derived from the **`Throwable`** class.

Exception Classes

- Classes that are derived from **`Error`** are thrown when critical errors occur, such as internal errors in the Java Virtual Machine, or running out of memory.
- Applications generally do *not* try to handle these errors, as they are the result of a serious condition.
- You *should* handle the exceptions which are instances of classes derived from the **`Exception`** class.

Exception Classes



Exception Classes from Standard Packages

The predefined exception class **`Exception`** is the root (ancestor) class for *all* exceptions

- Every exception class is a descendent class of the class **`Exception`**
- Although the **`Exception`** class can be used directly in a class or program, it is most often used to define a derived class
- The class **`Exception`** is in the **`java.lang`** package, and so requires no **`import`** statement

Exception Classes from Standard Packages

Numerous predefined exception classes are included in the standard packages that come with Java, for example:

```
IOException
NoSuchMethodException
FileNotFoundException
```

- Many exception classes must be imported in order to use them:

```
import java.io.IOException ;
```

Exception Classes

There are several subclasses in the standard Java libraries such as **`RuntimeExceptions`** and **`IOExceptions`**

- New exception classes can be defined by a programmer like any other class…
- … and we have all the tools that we need to do this ourselves!

Constructors and Accessor Methods

All exception classes (both pre-defined and programmer-defined) have the following properties:

- There is a constructor that takes a single argument of type **`String`**
- The class has an accessor method **`getMessage`** that can recover the **`String`** given as an argument to the constructor when the exception object was created

Using the `getMessage` Method

```
try {
    . . .
    throw new Exception(<StringArgument>);
    . . .
}
catch (Exception e) {
    System.out.println(e.getMessage());
    System.exit(0);
}   . . .
```

Using the **`getMessage`** Method

In the previous example, ***`<StringArgument>`*** is an argument to the **`Exception`** constructor

- This is the string used for the value of the string instance variable of exception **`e`**
  - Therefore, the method call **`e.getMessage()`** returns this string

Defining Your Own Exception Classes

Exception classes can be programmer-defined

- These can be tailored to carry the precise kinds of information needed in the **`catch`** block
- Different exceptions can identify different situations
- Every exception class to be defined must be a derived class of some already-defined exception class
  - These can be an exception class in the standard Java libraries, or a programmer-defined exception class

Defining Exception Classes

Constructors for exceptions are the most important members to define in an exception class

- Normally there are no other members except those inherited from the base class (i.e., the message)
- Typically two constructors are provided:
  1. A no-argument constructor which provides a default exception message
  2. A one-argument constructor which takes the message to be set as a string

A User-Defined Exception Class

```
public class MissingFileException extends Exception {

    // Default exception message
    public MissingFileException() {
        super("File not found! ") ;
    }

    // User-defined exception message when thrown
    public MissingFileException(String message) {
        super(message) ;
    }
}
```

Exception Object Characteristics

The two most important things about an exception object are its type (i.e., exception class) and the message it carries

- The message is sent along with the exception object as an instance variable
- This message can be recovered with the accessor method **`getMessage`**, so that the catch block can use the message

Programmer-Defined Exception Class Guidelines

- Exception classes may be programmer-defined, but every such class must be a derived class of an already existing exception class
- The class **`Exception`** can be used as the base class, unless another exception class would be more suitable
- At least two constructors should be defined
- The exception class should allow for the fact that the method **`getMessage`** is inherited

Preserve `getMessage`

- For all predefined exception classes, `getMessage` either:
  - returns the `String` that is passed to its constructor as an argument
  - Or it will return a default string if no argument is used with the constructor
- This behavior must be preserved in all programmer-defined exception class

Preserve `getMessage`

- A constructor must be included having a string parameter, and whose body begins with a call to **`super`**
  - The call to **`super`** must use the parameter as its argument
- A no-argument constructor must also be included whose body begins with a call to **`super`**
  - This call to **`super`** must provide a default string as its argument

Check or Unchecked?

Finally, you must decide which exception class your class will extend, which will determine whether your exception class is checked or unchecked

- To make your class *unchecked*, it must be a descendant of the **`RuntimeException`** class
- Otherwise, it will be a *checked* exception.

Still more on this subject later…

Mini-Lab #2

Write your own class which catches the Exception thrown in Mini-Lab #1

(See Instructions on Canvas)

Multiple `catch` Blocks

A `try` block can potentially throw any number of exception values, and they can be of differing types

- In any one execution of a `try` block, at most one exception can be thrown (since a throw statement ends the execution of the `try` block)
- However, different types of exception values can be thrown on different executions of the `try` block

Multiple `catch` Blocks

- Each `catch` block can only catch values of the exception class type given in the `catch` block heading
- Different types of exceptions can be caught by placing more than one `catch` block after a `try` block
  - Any number of `catch` blocks can be included, but they must be placed in the correct order

Catch the More *Specific* Exception First

When catching multiple exceptions, the order of the **`catch`** blocks is critical

- When an exception is thrown in a **`try`** block, the **`catch`** blocks are examined in the order they appear in your code
- The first one that matches the type of the exception thrown is the one that is executed

Catch the *More Specific* Exception First

Consider the following two catch blocks appearing after a **`try`** block:

```
catch (Exception e) {
    . . .
}
catch (FileNotFoundException e) {
    . . .
}
```

more general

more specific

Catch the *More Specific* Exception First

- Because a **`FileNotFoundException`** is a type of **`Exception`**, all **`FileNotFoundExceptions`** will be caught by the first **`catch`** block before ever reaching the second block
  - The **`FileNotFoundException`** block will never be used!
- For the correct ordering, simply reverse the two blocks

Throwing an Exception in a Method

Sometimes it makes sense to throw an exception in a method, but not catch it in the same method

- In such cases, the program using the method should enclose the method invocation in a **`try`** block, and catch the exception in a **`catch`** block that follows
- In this case, the method itself would not include **`try`** and **`catch`** blocks
- However, it would have to include a **`throws`** *clause*

Declaring Exceptions in a **`throws`** Clause

If a method can throw an exception but does not catch it, then the method must provide a ***`throws`*** *clause*

- The process of including an exception class in a throws clause is called *declaring the exception*

`throws <ExceptionName>`

- The following heading for **`aMethod`** declares that it could throw **`ExceptionName`**

```
public void aMethod() throws <ExceptionName>
```

Multiple Exceptions in a **`throws`** Clause

If a method can throw more than one type of exception, then the exception types should be separated by commas in the method heading

```
public void aMethod() throws <ExceptionName>,
                             <AnotherExceptionName>
```

`throws`

If a method doesn't use a try/catch statement, then the method *itself* can throw an `Exception` using `throws` after the method name

(Also demonstrate multiple `catch` blocks from one try block).

(MethodThrowsAnException)

The "Catch or Declare" Rule

Exceptions that might be thrown within a method should be accounted for in one of two ways:

1. The code that can throw an exception is placed within a **`try`** block, and the possible exception is caught in a **`catch`** block within the same method

2. The possible exception can be declared at the start of the method definition by placing the exception class name in a **`throws`** clause in the method's heading

The “Catch or Declare” Rule

The second technique is a way to shift exception handling responsibility to the method that invoked the exception throwing method

- In this case, the invoking method must handle the exception in a **`catch`** block, unless it too uses the same technique to "pass the buck“

Ultimately, every exception that is thrown should be caught by a **`catch`** *block in some method that does not declare the exception class in a* **`throws`** *clause*

Checked and Unchecked Exceptions

Exception classes which are subject to the “Catch or Declare” rule are called *checked* exceptions.

- The compiler checks to see if they are handled with either a *catch block* or a *throws clause.*
- Programs in which these exceptions can be thrown but are *not* handled will not compile until they *are* handled.

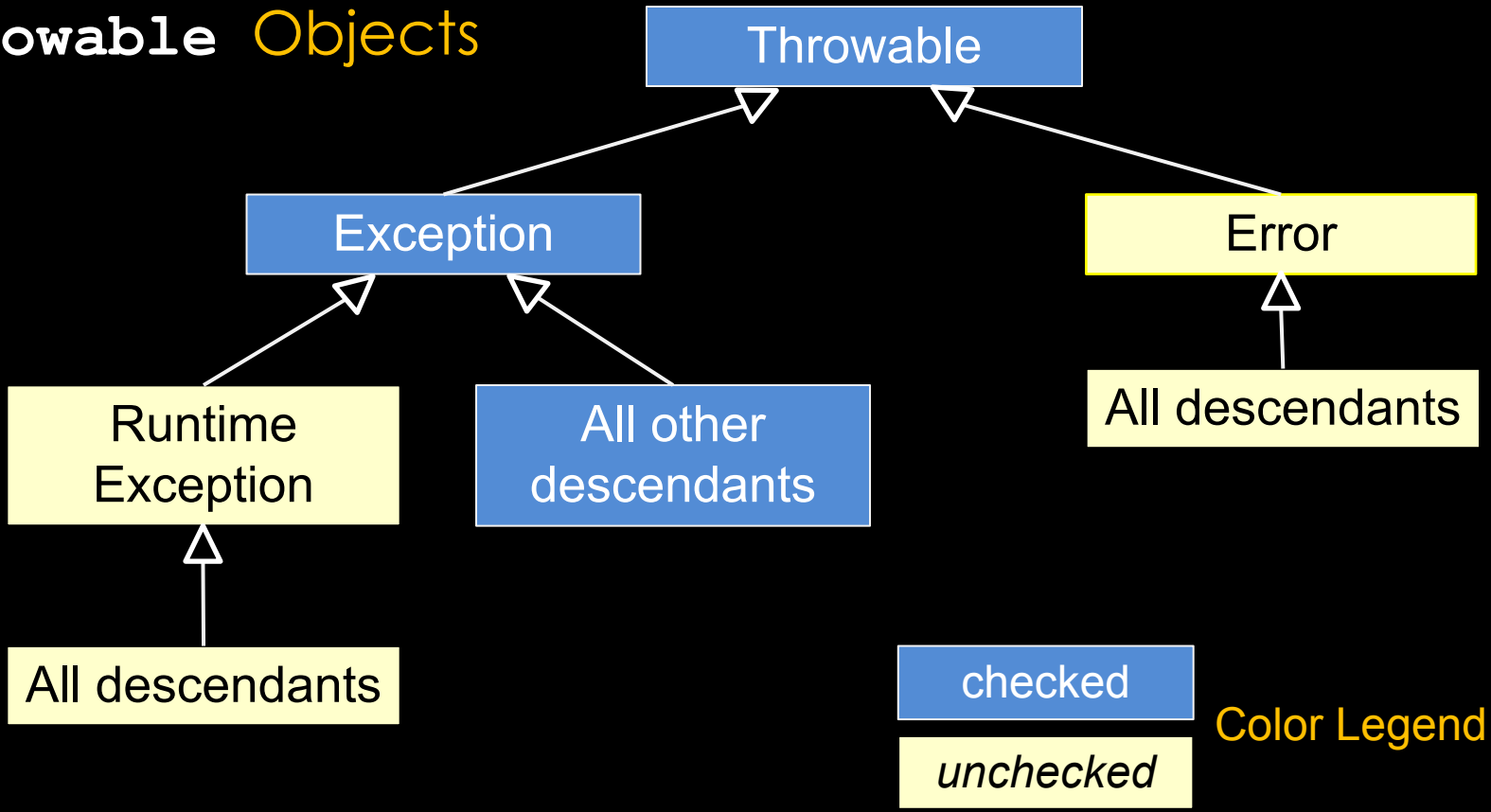
Exceptions to the "Catch or Declare" Rule

Unchecked exceptions are exempt from the "Catch or Declare" Rule

- The class **`RuntimeException`** and all of its descendants are *unchecked.*
- *All other exceptions are checked!*

The class **`Error`** and all of its descendants are called *error classes* and are *not* subject to the "Catch or Declare" Rule.

Hierarchy of `Throwable` Objects



The `charAt` method

Let's look at the `charAt` method in the `String` class

- This method creates an exception object by using the `throw` and `new` operators.
- Also notice that the method does not pass the exception (using `throws`) or `catch` the exception.
  - It is "unchecked" (how can we verify this?)
  - Does the "Catch or Declare" rule apply here?

The `throws` Clause in Derived Classes

- When a method in a derived class is overridden, it should have the same exception classes listed in its `throws` clause that it had in the base class
  - Or it should have a subset of them
- A derived class may not add any exceptions to the `throws` clause
  - But it can delete some

What Happens If an Exception Isn't Caught?

If every method just includes a **`throws`** clause for an exception, that exception may be thrown but never caught

- In a GUI program, nothing happens - but the program may be no longer be reliable
- In non-GUI programs, the program will terminate with an error message

Well-written programs should catch every exception eventually by a **`catch`** block in some method

Demonstrate the “Stack”

What do those intimidating messages mean when your program stops running?

(TheStackDemo)

What happens when Java throws an Exception?

Let's look at an example of Java code which does *not* use a try/catch block, then *does* use one to handle a problem more gracefully.

(StringIndexExceptionDemo)

Nested `try-catch` Blocks

It is possible to place a `try` block and its following catch blocks inside a larger `try` block, or inside a larger `catch` block

- If a set of `try-catch` blocks are placed inside a larger `catch` block, different names must be used for the `catch` block parameters in the inner and outer blocks, just like any other set of nested blocks.

Nested `try-catch` Blocks

- If a set of **`try-catch`** blocks are placed inside a larger **`try`** block, and an exception is thrown in the inner **`try`** block *that is not caught*, then the exception is thrown to the outer **`try`** block for processing, and may be caught in one of its **`catch`** blocks.

The `finally` Block

- **`finally`** is a *block* of code that will be executed after the **`try/catch`** block has completed and before the code following the **`try/catch`** block.
- The **`finally`** block will execute whether or not an exception is thrown, even if no **`catch`** statement matches the exception.

The **finally** Block

If it is used, a **finally** block is placed after a **try** block and its associated **catch** blocks

```
try {
    (Code which throws an exception)
}
catch(ExceptionClass e) {
    (Code execute when exception is thrown)
}
finally {
    (Code To Be Executed In All Cases)
}
```

The `finally` Block

Using a `try-catch-finally` block, there are three possibilities when the code is run:

1. The `try` block runs to the end, no exception is thrown, and the `finally` block is executed
2. An exception is thrown in the `try` block, caught in one of the `catch` blocks, and the `finally` block is executed
3. An exception is thrown in the `try` block, there is no matching `catch` block in the method, the `finally` block is executed, the method invocation ends, and the exception object is thrown to the enclosing method

The `finally` Block

Note that in the prior slide's first two possibilities, there really isn't a need for a `finally` block

- Control could simply be passed onto the code following the `try-catch` blocks

However, in the third possibility, there definitely is a need for a `finally` block

In all cases, a `finally` block is typically used to perform "clean-up" work, such as flushing and closing streams, updating logs, and closing unneeded windows or pop-ups

Rethrowing an Exception

A **`catch`** block can contain code that throws an exception

- Sometimes it is useful to catch an exception and then, depending on the string produced by **`getMessage`** or an instance variable in the Exception class, throw the same or a different exception for handling further up the chain of exception handling blocks
- This is called *rethrowing the exception*

Practices to Avoid

1. Avoid any indication that an exception has been thrown:

```
try {
        (do work that throws an exception)
}
catch (Throwable t) {
        (do nothing)
}
```

Instead, provide code which corrects the condition that throws the exception, and eliminate the `try` block

Code that I've Used Many Times…

```
FileInputStream stream = . . . ;
try {
    <open stream>
}

    . . .  Process the file's contents . . .

try {
   stream.close() ;
}
catch (IOException) {
   // do nothing
}
```

The Solution???

There is no easy way to test if a stream is open, so…

Avoid trying to close the file if it wasn't successfully opened:

- If the file couldn't be opened, then set **`stream`** to **`null`**
- Enclose the **`try`** block that closes the stream within an **`if`** statement such as

```
if (stream ==  null)
```

Practices to Avoid

2. Indicate that a problem exists, but don't provide or record a stack trace and simply continue processing:

```
try {
    (do work that throws an exception)
}
catch (Throwable t) {
    System.err.println("Error: " +
    t.getMessage());
}
```

Practices to Avoid

In this case, you can write the stack trace to the system's error log

```
try {
     (do work that throws an exception)
}
catch (Throwable t) {
     System.err.printStackTrace();
     System.err.println("Error: " +
               t.getMessage());
}
```

Event Driven Programming

- Exception handling is an example of a programming methodology known as *event-driven programming*
- When using event-driven programming, objects are defined so that they send events to other objects that handle the events
  - An event is also an object
  - Sending an event is called *firing an event*

Event Driven Programming

In exception handling, the event objects are the exception objects

- They are fired (or thrown) by an object when the object invokes a method that throws the exception
- An exception event is sent to a **`catch`** block, where it is handled

Summary Slides

Summary

An exception is an object that is generated as the result of an error or an unexpected event.

- Exception are said to have been “thrown.”
- It is the programmer’s responsibility to write code that detects and handles exceptions.
- Unhandled exceptions will crash a program.
- Java allows you to create exception handlers.

Summary

An *exception handler* is a section of code that gracefully responds to exceptions.

- The process of intercepting and responding to exceptions is called *exception handling*.
- The *default exception handler* deals with unhandled exceptions.
  - It prints an error message and crashes the program.

Summary

To handle an exception, you use a try statement.

```
try {
      (try block statements...)
}
catch (ExceptionType ParameterName) {
      (catch block statements...)
}
```

- First the keyword **`try`** indicates a block of code will be attempted (the curly braces are required). This block is known as a *try block*.

Summary

- A *try block* is one or more statements that are executed, and can potentially throw one or more exceptions.
- The application will not halt if the *try block* throws an exception.
- After the *try block*, at least one *catch block* appears which should “catch” the exception.

Summary

A catch clause begins with the key word **`catch`**:

`catch (`*`ExceptionType ParameterName`*`)`

- *ExceptionType* is the name of an exception class and
- *ParameterName* is a variable name which will reference the exception object, if thrown.

- The code that immediately follows the catch clause is known as a *catch block* (curly braces are required), and is executed if the try block throws an exception.

Summary

- The parameter type must be compatible with the thrown exception's type.
- After an exception, the program will continue execution at the point just past the **`catch`** block.
- Each exception object has a method named **`getMessage`** that can be used to retrieve the default error message for the exception.

Homework

- Complete old homework and labs.
- Complete the Exception Lab
- Homework Module 4, projects 1, 2, and 3
- Prepare for Quiz 3 on Exceptions

Group Lab

Create several new exception classes:

- In the first part, your group will create an exception class with two constructors which return Tornado warnings, and a program to test your class

Group Lab (continued)

- The second part involves creating a “stack” of objects (in this case, people) and creating exceptions when:

  1. “pushing” objects onto the stack when its full
  2. “popping” objects off the stack when its empty
  3. “pushing” objects onto the stack of the wrong type

  This type of structure is used when hiring and firing in a union or tenured environment.

The lab is due next class.