

New Requirement

- Assume your drawing application also allows user to put text on the drawing area and Drawing class has a new method that draws Shapes and Texts by calling the their draw methods

```
public class Text {  
  
    private String text;  
  
    public Text(String text){  
        this.text = text;  
    }  
  
    public void draw(){  
        //to be implemented  
    }  
}
```

New Requirement

- Similarly each Shape class also has draw method

```
public class Circle {  
    ...  
    public void draw(){  
        //to be implemented  
    }  
}  
public class Rectangle {  
    ...  
    public void draw(){  
        //to be implemented  
    }  
}
```

New Requirement

- And we will have a draw method in Drawing class that can draw all the shapes and texts it contains

```
public class DrawingV4{  
    ArrayList<Shape> shapes = new ArrayList<Shape>();  
    ArrayList<Text> texts = new ArrayList<Text>();  
    ...  
    public void draw(){  
        //call draw methods of all the shapes and texts  
    }  
}
```

draw method in Drawing

- And we will have a draw method in Drawing class that can draw all the shapes and texts it contains

```
public class DrawingV4{  
    ...  
    public void draw(){  
        for (Shape shape : shapes){  
            //call draw methods of each shape  
        }  
  
        for (Text text : texts){  
            text.draw();  
        }  
    }  
}
```

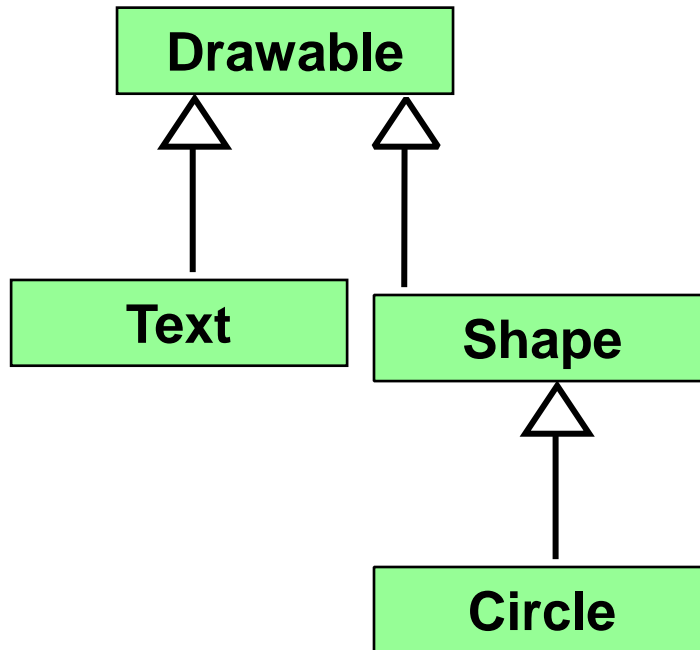
DrawingV4

- We have two lists and in the class to hold drawable objects
 - We have to declare new List for each drawable type
- We have two for loops in the draw method
 - We have to add new loop to draw new drawable types

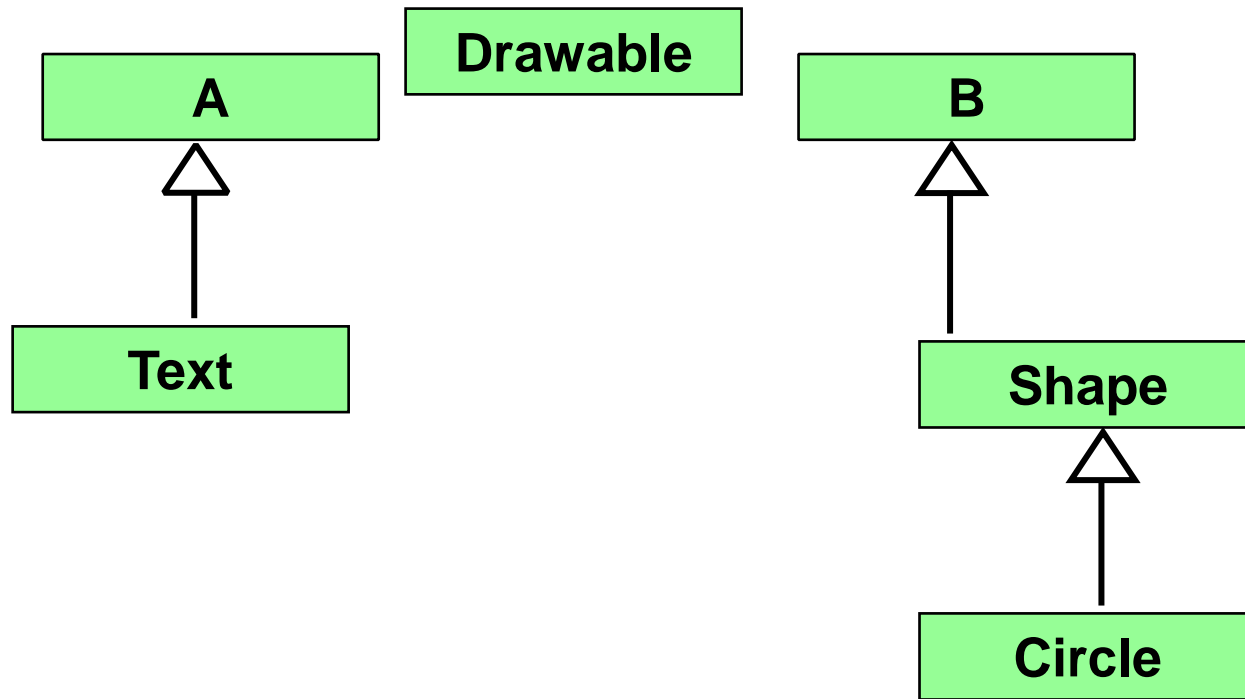
Drawing (Version 5)

```
public class DrawingV5{  
    ArrayList<Drawable> drawables= new ArrayList<Drawable>();  
    ...  
    public void draw(){  
        for (Drawable drawable: drawables){  
            drawable.draw();  
        }  
    }  
}
```

We can again use inheritance



Assume Text and Shape have already superclasses



- Multiple Inheritance is not allowed in Java

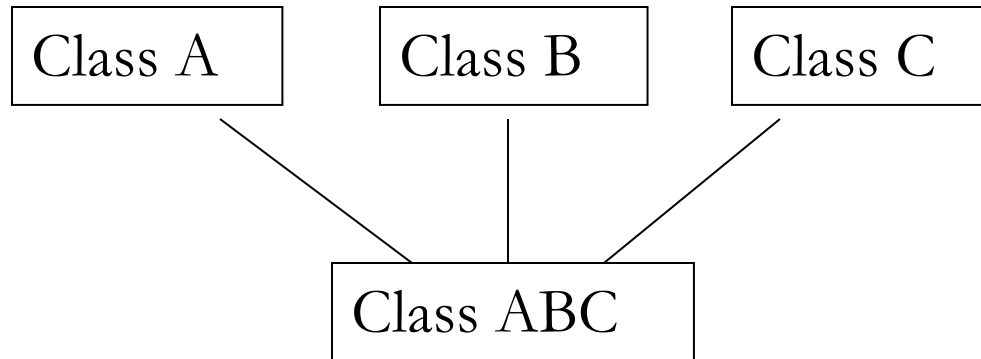
Interface

- An interface specifies a group of behaviors and gives them a name.
- Classes can choose to implement interfaces which require them to implement all of the methods in the interface.
- An interface is a named collection of method definitions and constants **ONLY**.
- A class that implements the interface agrees to implement **all** the methods defined in the interface, thereby agreeing to certain behaviors.

Interface and Abstract Classes

- An interface cannot implement any methods, whereas an abstract class can.
- A class can implement many interfaces but can have only one superclass.
- An interface is not part of the class hierarchy. Unrelated classes can implement the same interface.

Multiple Inheritance



Class ABC inherits all variables and methods from Class A, Class B, and Class C.

Java does NOT support multiple inheritances.

However, you can use **interface** to implement the functionality of multiple inheritance.

Interface Body

- The interface body contains method declarations for **ALL** the methods included in the interface.
- A method declaration within an interface is followed by a semicolon (;) because an interface does not provide implementations for the methods declared within it.
- All methods declared in an interface are implicitly public and abstract.

Drawable Interface

```
public interface Drawable {  
  
    public void draw();  
  
}
```

Drawable Text

```
public class Text implements Drawable{  
  
    private String text;  
  
    public Text(String text){  
        this.text = text;  
    }  
  
    public void draw(){  
        //to be imlemented  
    }  
}
```

Drawable Shape

```
public abstract class Shape implements Drawable{  
  
    public abstract double area();  
  
    public abstract double perimeter();  
  
}
```

Exception Handling

How to Handle Errors

- Assume we have a class that converts numbers in word representation to integer .

```
public int convert(String text) ;
```

- “seventy five” converted to 75

How to Handle Errors

- What should we do if the given input causes errors in methods

```
convert("xyz") ;  
convert("one sixty") ;
```

Separating Error-Handling Code from "Regular" Code

```
errorCodeType readFile {
    initialize errorCode = 0;

    open the file;
    if (theFileIsOpen) {
        determine the length of the file;
        if (gotTheFileLength) {
            allocate that much memory;
            if (gotEnoughMemory) {
                read the file into memory;
                if (readFailed) {
                    errorCode = -1;
                }
            } else {
                errorCode = -2;
            }
        } else {
            errorCode = -3;
        }
        close the file;
        if (theFileDidntClose && errorCode == 0) {
            errorCode = -4;
        } else {
            errorCode = errorCode and -4;
        }
    } else {
        errorCode = -5;
    }
    return errorCode;
}
```

- Using return values to return errors is not a good approach.
- Programming languages have exception handling mechanism to transfer and handle errors .

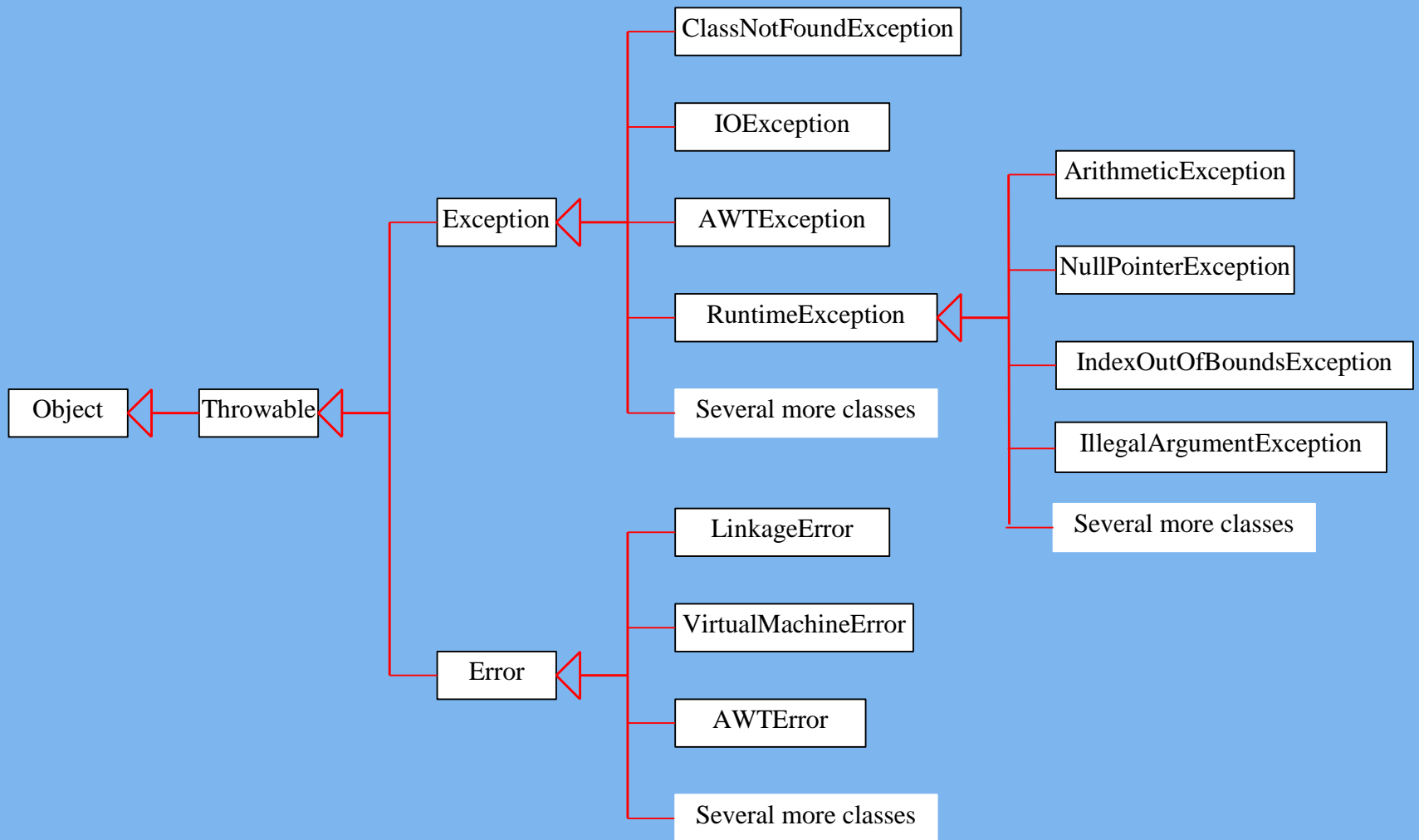
Separating Error-Handling Code from "Regular" Code

- Exception handling mechanism in Java lets you put all your error-handling code in one easy-to-read place.
- You don't have to use return values to forward errors.
- It also lets you to know that the method you are calling is risky that is the call may results errors
- If you know you might get an exception when you call a method then you can be prepared of the problem.

What Is an Exception?

- The Java programming language uses exceptions to handle errors and other exceptional events.
- **Definition:** An exception is an event, which occurs during the execution of a program, that breaks the normal flow of the program's instructions.

Exception Classes



Example

- Assume you have a class converting numbers to their text representations and your class can only convert numbers between 0 and 9

```
public class NumberConverter {  
    String[] numbers = {"Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine"};  
  
    public String convert(int number){  
        return numbers[number];  
    }  
}
```

Example

- If the input number is not between 0 and 9 how will the class behave?

Answer:

- We should report the problem to caller of the convert method. We can either
 - use an existing Exception class or
 - create our own Exception class to represent the Exception

ConvertException

// You can create your own exception by
//subclassing the Exception class

```
public class NumberConvertException extends Exception {  
    public NumberConvertException() {  
        super("Only numbers between 0 and 9 are convertible");  
    }  
}
```

Throwing an Exception?

- When an error occurs within a method, the method creates an object and hands it off to the runtime system.
- The object, called an exception object, contains information about the error, including its type and the state of the program when the error occurred.
- Creating an exception object and handing it to the runtime system is called **throwing an exception**.

Throwing an Exception

- All methods use the throw statement to throw an exception.
 - The throw statement requires a single argument: a throwable object.
 - Throwable objects are instances of any subclass of the Throwable class.

```
public class NumberConverter {  
  
    String[] numbers = {"Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine"};  
  
    public String convert(int number) throws NumberConvertException {  
        if (number > 9 || number < 0)  
            throw new NumberConvertException();  
        return numbers[number];  
    }  
}
```

Throwing an Exception

```
public Object pop() {  
    Object obj;  
  
    if (size == 0) {  
        throw new EmptyStackException();  
    }  
  
    obj = objectAt(size - 1);  
    setObjectAt(size - 1, null);  
    size--;  
    return obj;  
}
```

The Catch or Specify Requirement

- If an exception occurs in a method you should either
 - Catch and handle the exception or
 - Specify the exception to be thrown by the method
- Code that fails to meet the Catch or Specify will not compile
- Not all exceptions are subject to the Catch or Specify Requirement.

The Three Kinds of Exceptions

- Checked exception
 - exceptional conditions that a well-written application should anticipate and recover from.
 - checked for during compile time
 - are subject to the Catch or Specify Requirement
 - All exceptions are checked exceptions, except for those indicated by Error, RuntimeException, and their subclasses.

Checked exception

- If a checked exception is thrown in a method, it is subject to the **Catch** or Specify Requirement.

```
public void countWords(String filename){  
    BufferedReader br;  
    try {  
        br = new BufferedReader(new FileReader(filename));  
    } catch (FileNotFoundException e) {  
        // TODO Auto-generated catch block  
        e.printStackTrace();  
    }  
}
```

Checked exception

- If a checked exception is thrown in a method, it is subject to the Catch or **Specify** Requirement.

```
public FileInputStream(File file) throws FileNotFoundException {  
    ...  
    if (file.isInvalid()) {  
        throw new FileNotFoundException("Invalid file path");  
    }  
    ...  
}
```


The Catch or Specify Requirement

```
public class NumberConverter {  
  
    String[] numbers = {"Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine"};  
  
    public String convert(int number) throws NumberConvertException {  
        if (number > 9 || number < 0)  
            throw new NumberConvertException();  
        return numbers[number];  
    }  
  
    public static void main(String[] args) {  
        try {  
            foo(5);  
        } catch (NumberConvertException nce) {  
            System.out.println(nce.getMessage());  
        }  
    }  
  
    public static void foo(int num) throws NumberConvertException {  
        NumberConverter converter = new NumberConverter();  
        System.out.println(converter.convert(num));  
    }  
}
```

Specify

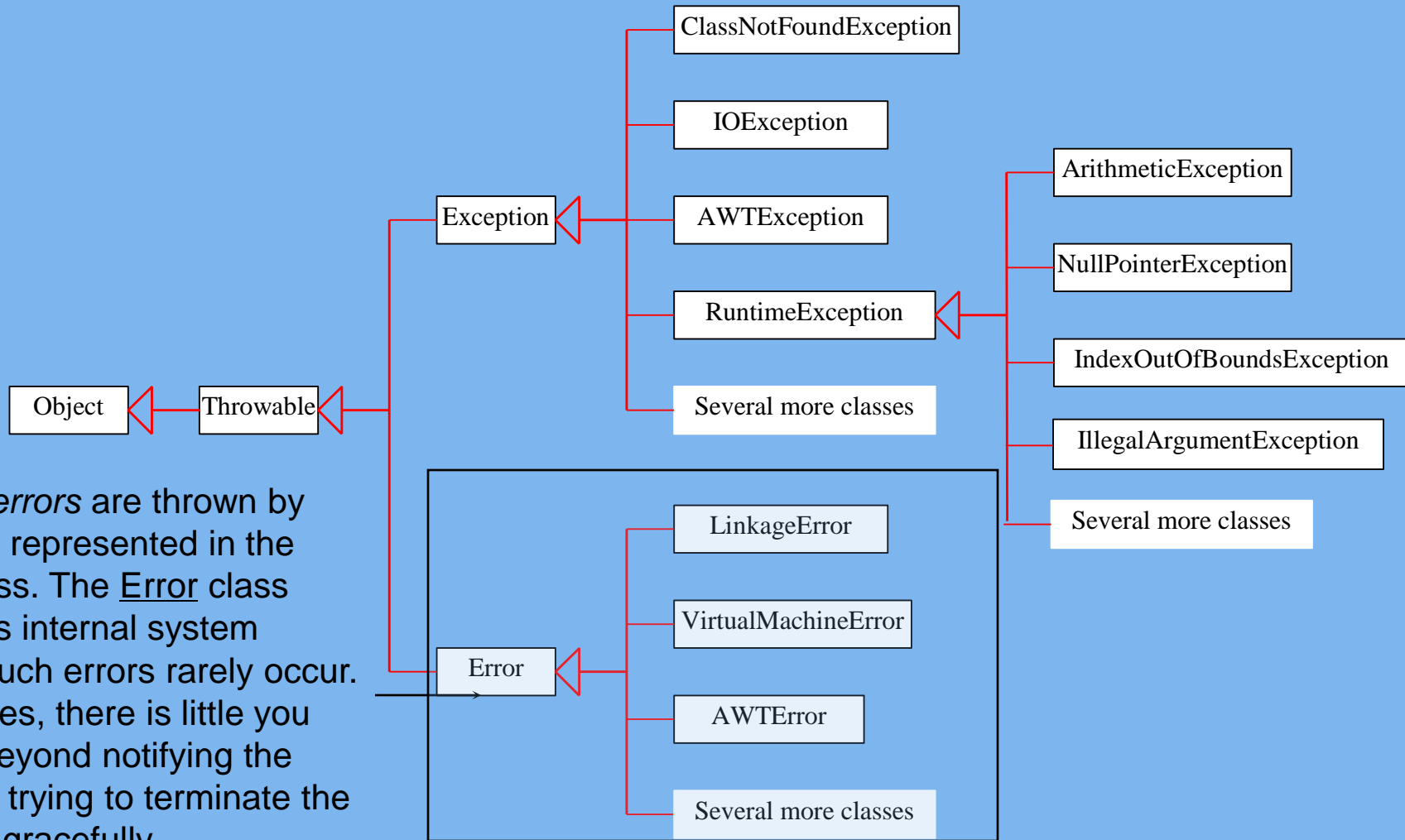
Catch

Specify

The Three Kinds of Exceptions

- Error
 - exceptional conditions that are external to the application, and that the application usually cannot anticipate or recover from.
 - An application might choose to catch this exception, in order to notify the user of the problem
 - but it also might make sense for the program to print a stack trace and exit.
 - Errors are not subject to the Catch or Specify Requirement.
 - Errors are those exceptions indicated by Error and its subclasses.

System Errors



System errors are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.

Error Example

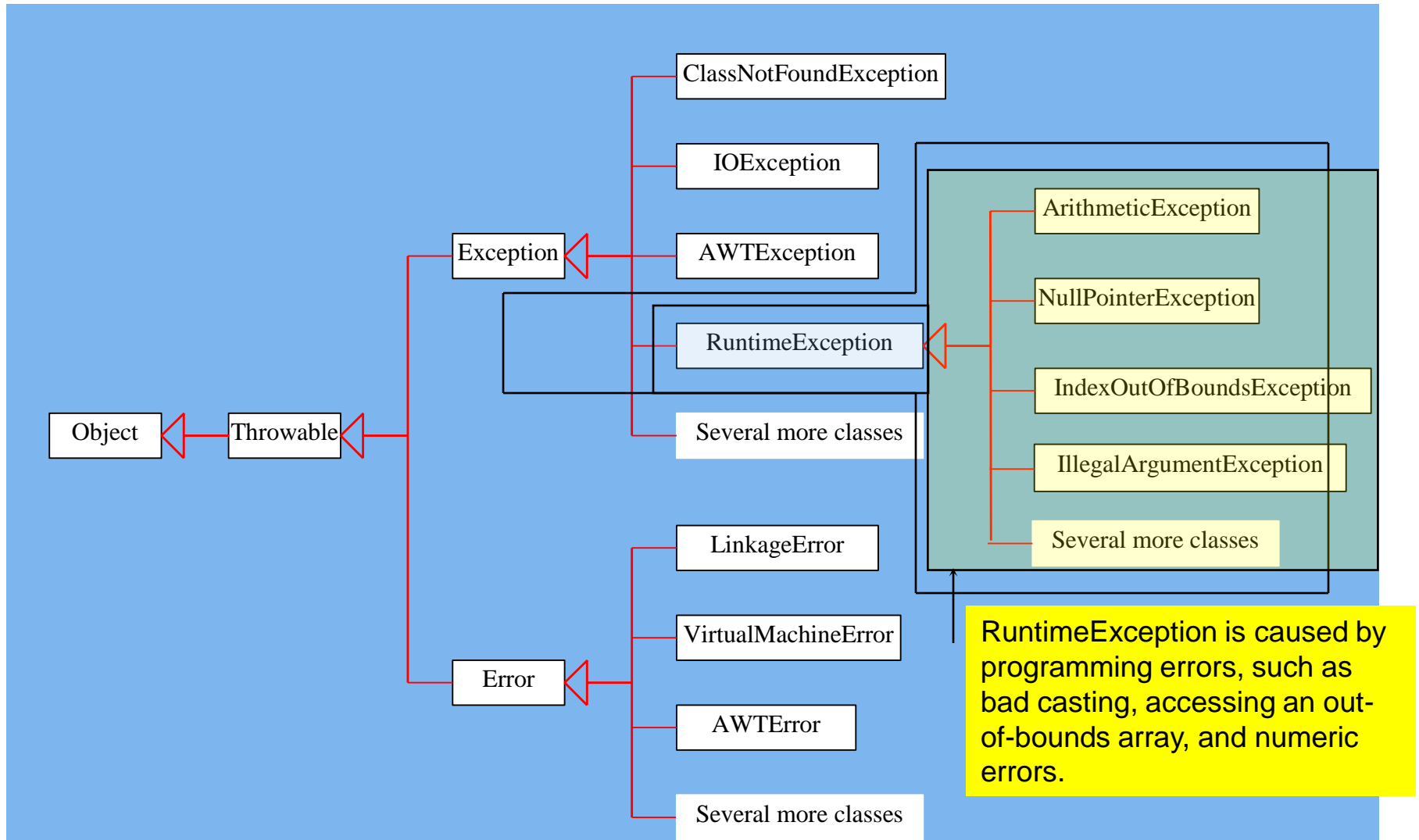
```
public class ErrorDemo {  
    public static void main(String[] args) {  
        abc();  
        System.out.println("Done");  
    }  
  
    public static void abc() {  
        int[][][] arr = new int [100000][100000][100000][100000];  
    }  
}
```

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
at exceptionhandling.ErrorDemo.abc(ErrorDemo.java:11)
at exceptionhandling.ErrorDemo.main(ErrorDemo.java:6)

The Three Kinds of Exceptions

- Runtime Exception
 - Exceptional conditions that are internal to the application, and that the application usually cannot anticipate or recover from.
 - usually indicate programming bugs, such as logic errors or improper use of an API.
 - Consider the application described previously that passes a file name to the constructor for `FileReader`.
 - If a logic error causes a null to be passed to the constructor, the constructor will throw `NullPointerException`.
 - The application can catch this exception, but it probably makes more sense to eliminate the bug that caused the exception to occur.
 - Runtime exceptions are not subject to the Catch or Specify Requirement.
 - Runtime exceptions are those indicated by `RuntimeException` and its subclasses.

Runtime Exceptions



Runtime Exception

```
public FileInputStream(File file) throws FileNotFoundException {  
    String name = (file != null ? file.getPath() : null);  
    ...  
    if (name == null) {  
        throw new NullPointerException();  
    }  
    if (file.isInvalid()) {  
        throw new FileNotFoundException("Invalid file path");  
    }  
    ...  
}
```

RuntimeException Class

- reserved for exceptions that indicate incorrect use of an API
 - An example of a runtime exception is **NullPointerException**, which occurs when a method tries to access a member of an object through a null reference.

RuntimeException Example

```
public class RunTimeExpDemo {  
    public static void main(String[] args) {  
        abc(5,0);  
    }  
    public static int abc(int a, int b) {  
        return a/b;  
    }  
}
```

```
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at exceptionhandling.RunTimeExpDemo.abc(RunTimeExpDemo.java:13)  
    at exceptionhandling.RunTimeExpDemo.main(RunTimeExpDemo.java:7)
```

Checked Exceptions vs. Unchecked Exceptions

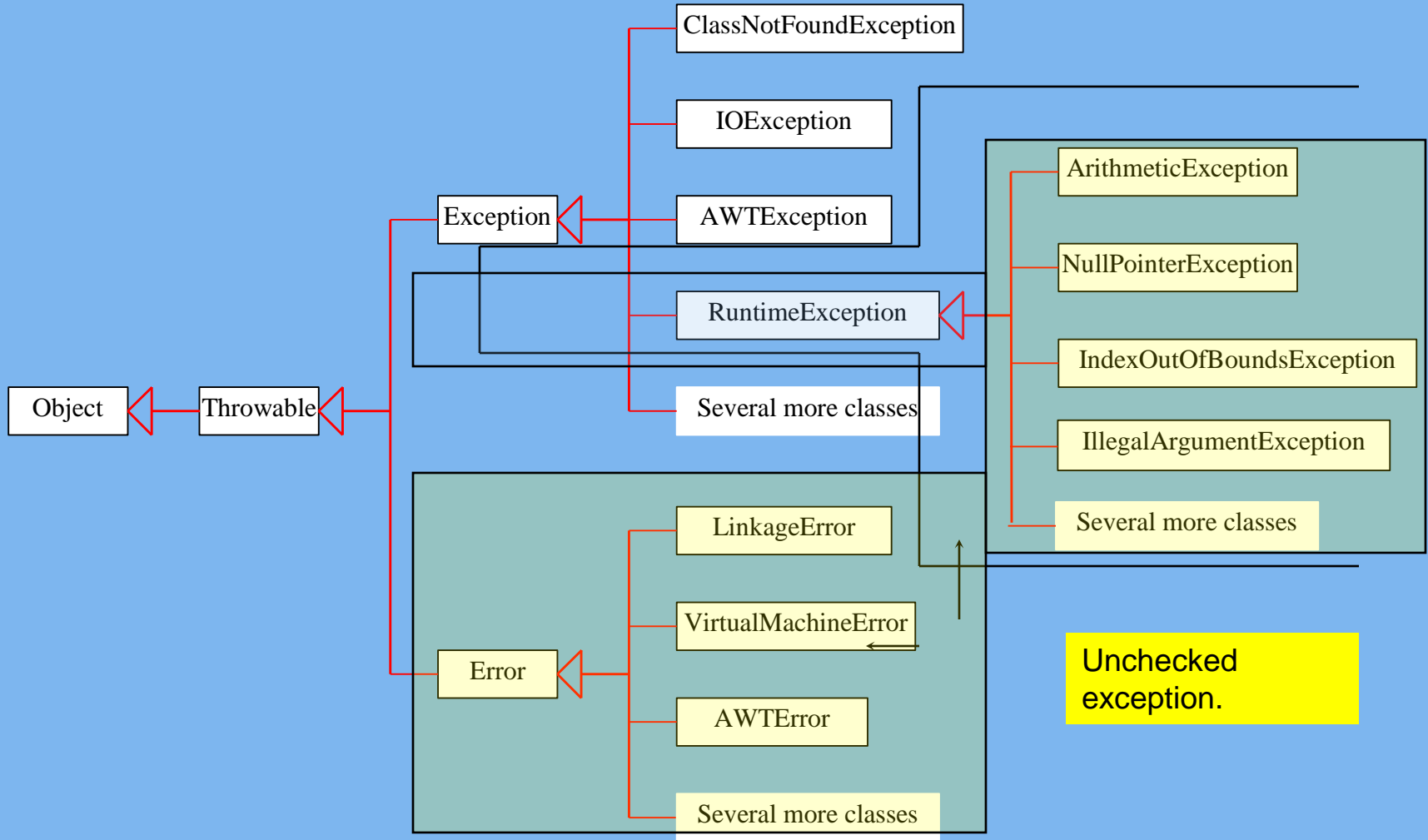
RuntimeException, Error and their subclasses are known as ***unchecked exceptions***.

All other exceptions are known as ***checked exceptions***, meaning that the compiler forces the programmer to check and deal with the exceptions.

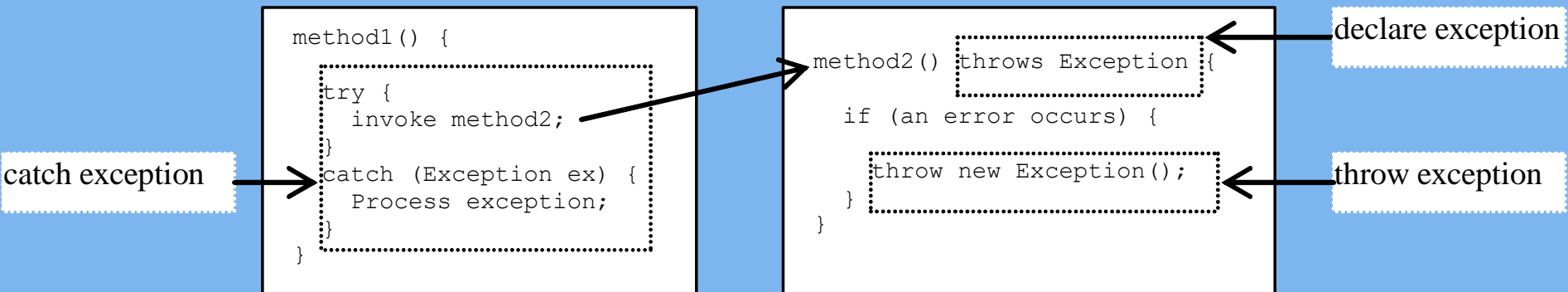
Unchecked Exceptions

In most cases, unchecked exceptions reflect programming logic errors that are not recoverable. For example, a `NullPointerException` is thrown if you access an object through a reference variable before an object is assigned to it; an `IndexOutOfBoundsException` is thrown if you access an element in an array outside the bounds of the array. These are the logic errors that should be corrected in the program. Unchecked exceptions can occur anywhere in the program. To avoid cumbersome overuse of try-catch blocks, **Java does not mandate you to write code to catch unchecked exceptions.**

Checked or Unchecked Exceptions



Declaring, Throwing, and Catching Exceptions



Declaring/Specifying Exceptions

Every method must state the types of checked exceptions it might throw. This is known as *declaring/specifying exceptions*.

```
public void myMethod()  
    throws IOException
```

```
public void myMethod()  
    throws IOException, OtherException
```

Throwing Exceptions

When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example,

```
throw new TheException();
```

```
TheException ex = new TheException();  
throw ex;
```

Throwing Exceptions Example

```
/** Set a new radius */  
public void setRadius(double newRadius)  
    throws IllegalArgumentException {  
    if (newRadius >= 0)  
        radius = newRadius;  
    else  
        throw new IllegalArgumentException(  
            "Radius cannot be negative");  
}
```


Handling Exceptions

- The first step in constructing an exception handler is to enclose the code that might throw an exception within a try block.

```
try {  
    code  
}
```

```
catch and finally blocks . . .
```

Handling Exceptions

- A *try block* is:
 - one or more statements that are executed, and
 - can potentially throw an exception.
- The application will not halt if the try block throws an exception.
- After the try block, a `catch` clause appears.

The catch Blocks

- You associate exception handlers with a try block by providing one or more catch blocks directly after the try block.

```
try {  
  
} catch (ExceptionType1 name) {  
  
} catch (ExceptionType2 name) {  
  
}
```

Catching Exceptions

```
try {  
    statements;    // Statements that may throw exceptions  
}  
catch (Exception1 exVar1) {  
    handler for exception1;  
}  
catch (Exception2 exVar2) {  
    handler for exception2;  
}  
...  
catch (ExceptionN exVar3) {  
    handler for exceptionN;  
}
```

The catch Blocks

- Each catch block is an exception handler that handles the type of exception indicated by its argument.
- ExceptionType, declares the type of exception that the handler can handle and must be the name of a class that inherits from the Throwable class.
- The handler can refer to the exception with variable name.

Handling Exceptions

- This code is designed to handle a `FileNotFoundException` if it is thrown.

```
try
{
    File file = new File ("MyFile.txt");
    Scanner inputFile = new Scanner(file);
}
catch (FileNotFoundException e)
{
    System.out.println("File not found.");
}
```

- The Java Virtual Machine searches for a `catch` clause that can deal with the exception.
- Example: [OpenFile.java](#)

Handling Exceptions

- The parameter must be of a type that is compatible with the thrown exception's type.
- After an exception, the program will continue execution at the point just past the catch block.

The catch Blocks

- Exception handlers can do more than just print error messages or halt the program.
 - They can do error recovery,
 - Prompt the user to make a decision, or
 - Propagate the error up to a higher-level handler using chained exceptions,

Catching Exceptions

```
main method {  
    ...  
    try {  
        ...  
        invoke method1;  
        statement1;  
    }  
    catch (Exception1 ex1) {  
        Process ex1;  
    }  
    statement2;  
}
```

```
method1 {  
    ...  
    try {  
        ...  
        invoke method2;  
        statement3;  
    }  
    catch (Exception2 ex2) {  
        Process ex2;  
    }  
    statement4;  
}
```

```
method2 {  
    ...  
    try {  
        ...  
        invoke method3;  
        statement5;  
    }  
    catch (Exception3 ex3) {  
        Process ex3;  
    }  
    statement6;  
}
```

An exception
is thrown in
method3

Catch or Specify Checked Exceptions

Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than Error or RuntimeException), you must invoke it in a try-catch block or declare to throw the exception in the calling method. For example, suppose that method p1 invokes method p2 and p2 may throw a checked exception (e.g., IOException), you have to write the code as shown in (a) or (b).

```
void p1() {  
    try {  
        p2();  
    }  
    catch (IOException ex) {  
        ...  
    }  
}
```

(a)

```
void p1() throws IOException {  
    p2();  
}
```

(b)

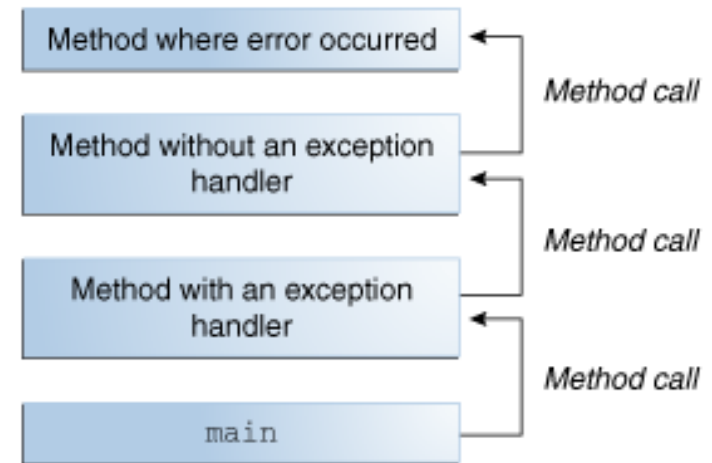
Catch Runtime Errors

```
1      import java.util.*;
2
3      public class HandleExceptionDemo {
4          public static void main(String[] args) {
5              Scanner scanner = new Scanner(System.in);
6              boolean continueInput = true;
7
8              do {
9                  try {
10                     System.out.print("Enter an integer: ");
11                     int number = scanner.nextInt();
12
13                     // Display the result
14                     System.out.println(
15                         "The number entered is " + number);
16
17                     continueInput = false;
18                 }
19                 catch (InputMismatchException ex) {
20                     System.out.println("Try again. (" +
21                         "Incorrect input: an integer is required)");
22                     scanner.nextLine(); // discard input
23                 }
24             } while (continueInput);
25     }
```

If an exception occurs on this line,
the rest of lines in the try block are
skipped and the control is
transferred to the catch block.

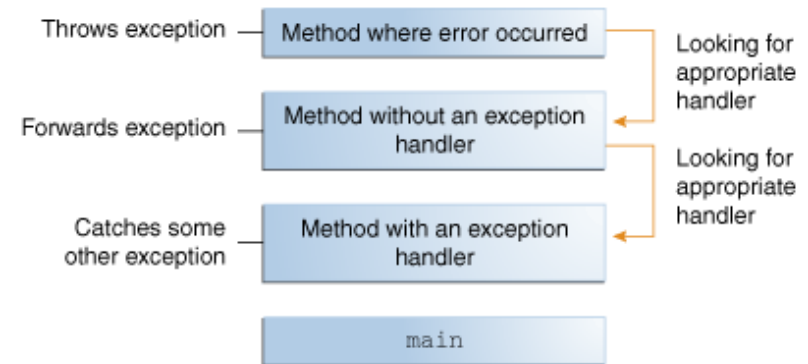
Call Stack

- After a method throws an exception, the runtime system attempts to find something to handle it.
- The set of possible "somethings" to handle the exception is the ordered list of methods that had been called to get to the method where the error occurred.
- The list of methods is known as the **call stack**



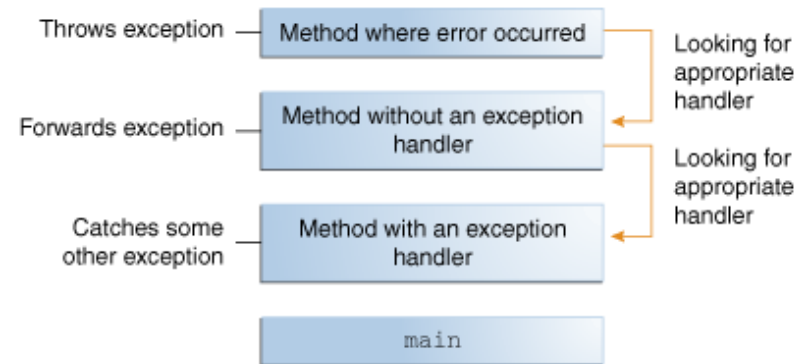
Exception Handler

- The runtime system searches the call stack for a method that contains a block of code that can handle the exception.
- This block of code is called an exception handler.
- The search begins with the method in which the error occurred and proceeds through the call stack in the reverse order in which the methods were called.



Exception Handler

- The exception handler chosen is said to catch the exception.
- If the runtime system exhaustively searches all the methods on the call stack without finding an appropriate exception handler, the runtime system (and, consequently, the program) terminates.



The catch Blocks

- The catch block contains code that is executed if and when the exception handler is invoked.
- The runtime system invokes the exception handler when the handler is the first one in the call stack whose `ExceptionType` matches the type of the exception thrown.

Catching More Than One Type of Exception with One Exception Handler

- A single catch block can handle more than one type of exception.
 - This feature can reduce code duplication and lessen the temptation to catch an overly broad exception.

```
catch (IOException | SQLException ex) {  
    logger.log(ex) ;  
    throw ex;  
}
```


Rethrowing Exceptions

```
try {  
    statements;  
}  
catch (TheException ex) {  
    perform operations before exits;  
    throw ex;  
}
```

The `finally` Block

```
try {  
    statements;  
}  
catch (TheException ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}
```

The finally Block

- The finally block always executes when the try block exits.
- This ensures that the finally block is executed even if an unexpected exception occurs.

```
finally {  
    if (out != null) {  
        System.out.println("Closing PrintWriter");  
        out.close();  
    } else {  
        System.out.println("PrintWriter not open");  
    }  
}
```

Trace a Program Execution

Suppose no exceptions in the statements

```
try {  
    statements;  
}  
catch (TheException ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}
```

Next statement;

Trace a Program Execution

```
try {  
    statements;  
}  
catch (TheException ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}  
  
Next statement;
```



The final block is always executed

Trace a Program Execution

```
try {  
    statements;  
}  
catch (TheException ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}
```

Next statement;

Next statement in the
method is executed

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}
```

Suppose an exception of type Exception1 is thrown in statement2

Next statement;

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}
```



The exception is handled.

Next statement;

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}
```

The final block is always executed.

Next statement;

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
finally {  
    finalStatements;  
}
```

Next statement;

The next statement in the method is now executed.

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
catch (Exception2 ex) {  
    handling ex;  
    throw ex;  
}  
finally {  
    finalStatements;  
}
```

statement2 throws an exception of type Exception2.

Next statement;

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch(Exception1 ex) {  
    handling ex;  
}  
catch(Exception2 ex) {  
    handling ex;  
    throw ex;  
}  
finally {  
    finalStatements;  
}
```



Handling exception

Next statement;

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch(Exception1 ex) {  
    handling ex;  
}  
catch(Exception2 ex) {  
    handling ex;  
    throw ex;  
}  
finally {  
    finalStatements;  
}
```

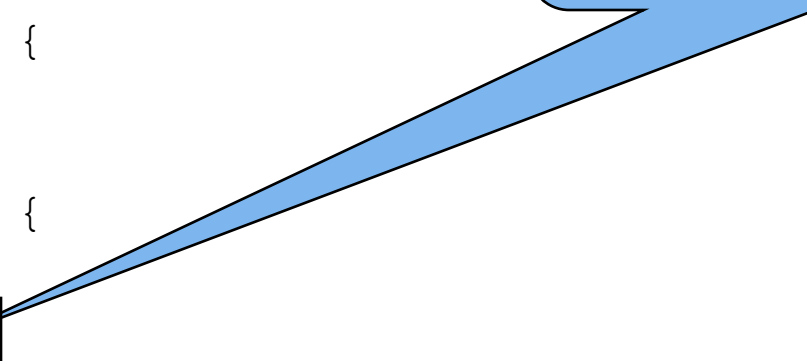
Execute the final block



Next statement;

Trace a Program Execution

```
try {  
    statement1;  
    statement2;  
    statement3;  
}  
catch (Exception1 ex) {  
    handling ex;  
}  
catch (Exception2 ex) {  
    handling ex;  
    throw ex;  
}  
finally {  
    finalStatements;  
}
```



Rethrow the exception and control is transferred to the caller

Next statement;

When to Throw Exceptions

- An exception occurs in a method.
 - If you want the exception to be processed by its caller, you should create an exception object and throw it.
 - If you can handle the exception in the method where it occurs, there is no need to throw it.

Chained Exceptions

- An application often responds to an exception by throwing another exception.
- The first exception causes the second exception

```
try {  
  
} catch (IOException e) {  
    throw new SampleException("Other IOException", e);  
}
```


Chained Exceptions

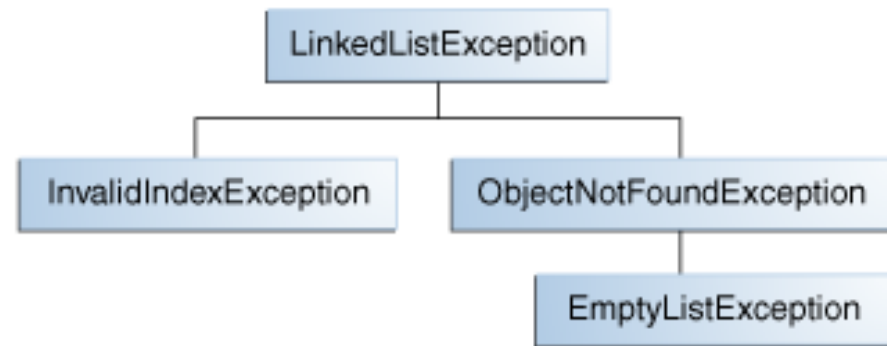
- The following are the methods and constructors in `Throwable` that support chained exceptions.
- `Throwable getCause()`
- `Throwable initCause(Throwable)`
- `Throwable(String, Throwable)`
- `Throwable(Throwable)`

Creating Exception Classes

- When faced with choosing the type of exception to throw,
 - you can either use one written by someone else (the Java platform provides a lot of exception classes you can use)
 - or you can write one of your own.

An Example: Linked List Class

- The class supports the following methods:
 - **objectAt(int n)** — Throws an exception if the argument is less than 0 or more than the number of objects currently in the list.
 - **firstObject()** — Throws an exception if the list contains no objects.
 - **indexOf(Object o)** — Throws an exception if the object passed into the method is not in the list.



Checked vs. Runtime

- Runtime exceptions can occur anywhere in a program, and in a typical one they can be very numerous.
- Having to add runtime exceptions in every method declaration would reduce a program's clarity.
- Thus, the compiler does not require that you catch or specify runtime exceptions
- In general, do not throw a RuntimeException or create a subclass of RuntimeException simply because you don't want to be bothered with specifying the exceptions your methods can throw.

Checked vs. Runtime

- Here's the bottom line guideline:
 - If a client can reasonably be expected to recover from an exception, make it a checked exception.
 - If a client cannot do anything to recover from the exception, make it an unchecked exception.

Creating Custom Exception Classes

- ❑ Use the exception classes in the API whenever possible.
- ❑ Create custom exception classes if the predefined classes are not sufficient.
- ❑ Declare custom exception classes by extending Exception or a subclass of Exception.

Advantages of Exceptions

- Advantage 1: Separating Error-Handling Code from "Regular" Code
 - consider the pseudocode method here that reads an entire file into memory.

readFile {

open the file; //What happens if the file can't be opened?

determine its size; // What happens if the length of the file can't be determined?

allocate that much memory; //What happens if enough memory can't be allocated?

read the file into memory; //What happens if the read fails?

close the file; //What happens if the file can't be closed?

}

Advantages of Exceptions

- **Advantage 2: Propagating Errors Up the Call Stack**
 - Suppose that the `readFile` method is the fourth method in a series of nested method calls made by the main program: `method1` calls `method2`, which calls `method3`, which finally calls `readFile`.

```
method1 {  
    call method2;  
}  
  
method2 {  
    call method3;  
}  
  
method3 {  
    call readFile;  
}
```


Advantages of Exceptions

- **Advantage 2: Propagating Errors Up the Call Stack**
 - Suppose also that **method1** is the only method interested in the errors that might occur within **readFile**.

```
method1 {
    errorCodeType error;
    error = call method2;
    if (error)
        doErrorProcessing;
    else
        proceed;
}

errorCodeType method2 {
    errorCodeType error;
    error = call method3;
    if (error)
        return error;
    else
        proceed;
}

errorCodeType method3 {
    errorCodeType error;
    error = call readFile;
    if (error)
        return error;
    else
        proceed;
}
```

```
method1 {
    try {
        call method2;
    } catch (exception e) {
        doErrorProcessing;
    }
}

method2 throws exception {
    call method3;
}

method3 throws exception {
    call readFile;
}
```

Advantages of Exceptions

- **Advantage 3: Grouping or categorizing of exceptions as class hierarchy.**
 - IOException is the most general and represents any type of error that can occur when performing I/O.
 - Its descendants represent more specific errors. For example, FileNotFoundException means that a file could not be located on disk.
 - A method can write specific handlers that can handle a very specific exception.

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

- A method can catch an exception based on its group or general type by specifying any of the exception's superclasses in the catch statement.

```
catch (IOException e) {  
    ...  
}
```

Advantages of Exceptions

- **Advantage 3: Grouping or categorizing of exceptions as class hierarchy.**
 - You could even set up an exception handler that handles any Exception with the handler here.

```
// A (too) general exception handler
catch (Exception e) {
    ...
}
```

Questions

- Is the following code legal?

```
try {  
  
} finally {  
  
}
```

Questions

- What exception types can be caught by the following handler?

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

Questions

- Is there anything wrong with this exception handler as written? Will this code compile?

```
try {  
  
} catch (Exception e) {  
  
} catch (ArithmeticException a) {  
  
}
```

References

- <http://math.hws.edu/javanotes/>
- <http://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-092-introduction-to-programming-in-java-january-iap-2010/lecture-notes/>
- <https://www.tiobe.com/tiobe-index/>
- Head First Java 2nd Edition
- Java a Beginner's guide 7th Edition