

## **Lesson 12: Exception-Handling in Java**

### **Wholeness of the Lesson**

Prior to the emergence of OO languages, error-handling was typically done using an unsystematic use of error codes. This approach led to confusion, programming errors, and costly maintenance. Java's exception-handling model (which is similar to those in other OO languages) systematizes the task of handling error conditions and integrates it with the OO paradigm supported by the language. This advance in programming practice illustrates the theme that “deeper knowledge has more profound organizing power.”

## **What Is Exception-Handling All About?**

1. Problems can arise during execution of an application.
2. Examples:
  - a. Try to open a file but can't
  - b. Try to access a database, but it's unavailable
  - c. Try to save data, but disk is full
  - d. Try to call a method on an uninitialized object
  - e. Try to access an array index beyond the defined array length
  - f. Try to divide a number by 0

3. Regardless of the problem, any error condition that arises in an application should be handled in one of two ways (depending on the situation):

Either:

- a. Return to a safe state and enable the user to execute other commands (for example, the user accidentally inputs incorrect data, such as an incomplete phone number – the application should ask the user to try again)
- b. Allow the user to save all work and terminate the application gracefully (for example, a database may not be accessible, so the user should be allowed to "try again later")

But what is the right way to accomplish this objective?

## **The Old (Non-OO) Way Of Doing It**

1. In languages like C, it used to be common to handle error conditions by returning to a calling function an "error code", such as  $-1$ , to indicate that some operation has failed or problem has occurred. In case many different types of problems need to be anticipated, there could be many different error codes. The calling function would then have to decipher these codes and determine a course of action in each case.
2. There are two fundamental problems with this approach:

Problem 1: Violation of Open-Closed Principle. Suppose when a piece of code is first written, the developer can think of just two possible problems that might arise. So his function is defined so that if one problem arises, the code returns -1, and if the other problem arises, it returns -2. Then he writes error-handling code in the calling function to handle each error condition.

```
callingFunction(String s) {
    int code = saveData(s);
    if(code == -1) {
        handleOpenFileError();
    }
    else if(code == -2) {
        handleWriteToFileError();
    }
    else {
        doTheNormalThing(s);
    }
}

int saveData(String s) {
    int result = openFile("Special File");
    if(result == -1) return -1;
    else {
        result = writeToFile(s);
        if(result == -1) return -2;
        else return 0;
    }
}
```

Suppose later a new developer takes over the code and discovers that the file ought to be closed, but it is possible that the "close" operation will fail. So he has to go into the code and add new conditions inside code blocks.

```
callingFunction(String s) {
    int code = saveData(s);
    if(code == -1) {
        handleOpenFileError();
    }
    else if(code == -2) {
        handleWriteToFileError();
    }
    else if(code == -3) {
        handleCloseFileError();
    }
    else {
        doTheNormalThing(s);
    }
}
int saveData(String s) {
    int result = openFile("Special File");
    if(result == -1) return -1;
    else {
        result = writeToFile(s);
        if(result == -1) return -2;
        else {
            result = closeFile();
            if(result == -1) return -3;
            else return 0;
        }
    }
}
```

This is already getting messy, especially in the saveData method.

Problem 2: Confusion of data. If a method such as `saveData` in our example is really intended to return an integer, an integer such as `-1` might possibly represent a valid result of a computation rather than an error code. (And this problem doesn't go away just by picking a number different from `-1` to represent an error code.)

As a contrived example, suppose you have a function that performs the following operation: it accepts 3 integers `a`, `b`, `c` as input and then performs two successive integer divisions and returns the result:

```
contrivedDivide(int a, int b, int c) {  
    int temp = a / b;  
    return temp / c;  
}
```

The developer may decide that an error code of  $-1$  should be returned if either  $b$  or  $c$  is  $0$  (since division by  $0$  is illegal).

```
contrivedDivide(int a, int b, int c) {  
    if(b == 0 || c == 0) return -1;  
    int temp = a / b;  
    return temp / c;  
}
```

But if this is done, how can the calling function distinguish between an error condition and the result of a legitimate execution of the function, as in

```
contrivedDivide(1, -1, 1); // returns -1, but no error
```



## **An Object-Oriented Error-Handling Strategy**

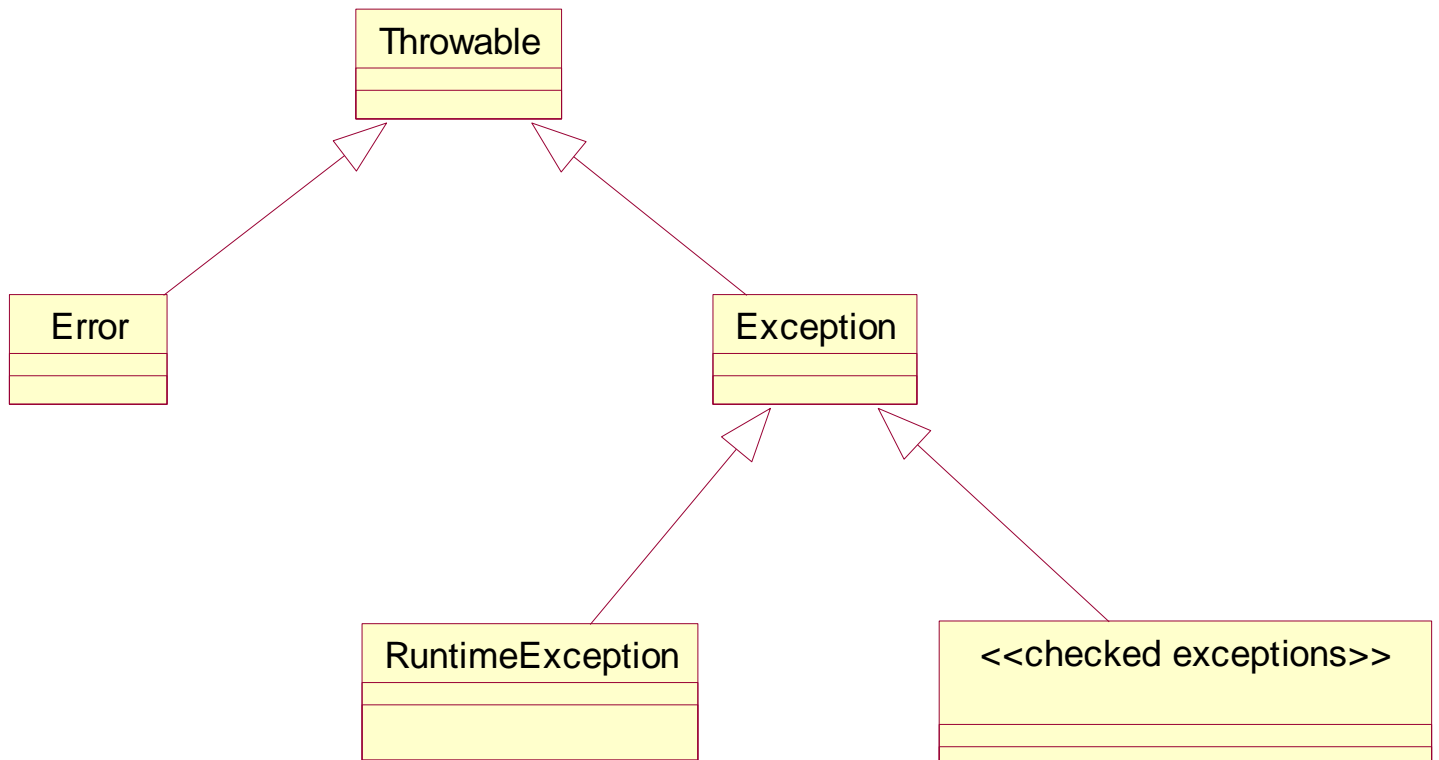
Java's solution to the problem is very similar to the solution offered in most OO languages:

- a. An error of any kind is represented as a special kind of object
- b. When an error condition arises, an instance of the object is created by the Java runtime and "thrown" (similar to the way an "event" is triggered by a button click or other user action on a GUI)
- c. Code written by the developer then "catches" the error-related object, analyzes the information in this object as needed, and performs some action to handle it.

## MAIN POINT

Java's exception-handling model supports best practices in handling exceptions that arise during program execution. Likewise, establishing awareness in the home of all the laws of nature results in a life spontaneously lived in accord with natural law.

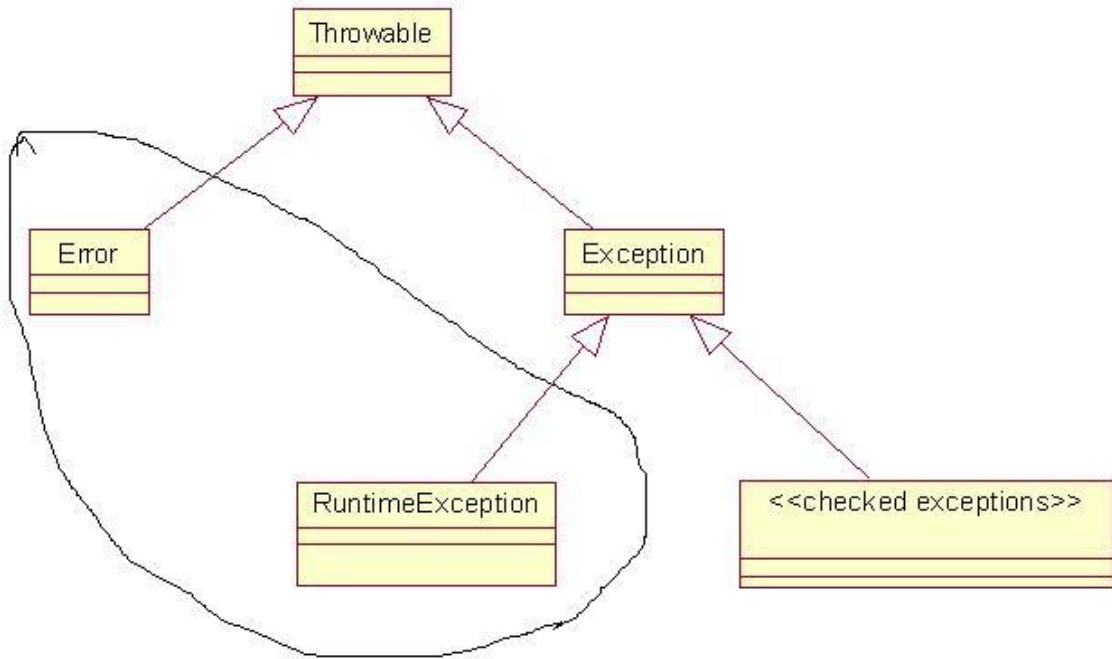
## The Hierarchy of Classes That Represent Error Conditions



## Classification of Error-Condition Classes

In Java, error-condition classes belong to one of three categories:

- *Error* – Objects in this category belong to the inheritance hierarchy headed by the `Error` class
- *Other Unchecked Exceptions* – Besides `Error` objects, unchecked exceptions include all objects that belong to the inheritance hierarchy headed by the class `RuntimeException`.
- *Checked Exceptions* – Exceptions in this category are subclasses of `Exception` but not subclasses of `RuntimeException`.



**"Unchecked Exceptions"**

## Objects of type **Error**

1. `Error` objects describe internal errors, JVM execution errors, or resource exhaustion. They occur rarely, but usually, if they do occur, the application must be terminated.
2. From the developer's point of view, there is nothing that must be done to handle this kind of error during execution of the application – no “catch” clause could resolve the problem. It's beyond the control of the developer, like a system or hard disk crash.  
[*Note: A `StackOverflowError` is an example of an `Error` that can typically be handled by rewriting the code, but nothing can be done to solve this problem (or any other `Error`) during program execution.*]
3. When one of these errors occurs, the JVM *throws* an `Error` object. Typically, since the developer has not written special code to handle the error event, the JVM will handle the `Error` object by displaying a message to the console indicating the type of `Error` and the sequence of method calls that led to the error condition (called a *stack trace*).

**Example:** (From Lesson 4 – Recursion). If you create an illegal recursion, you will typically cause a `StackOverflowError` to occur because the sequence of self-calls overflows the call stack.

```
class MyClass {
    public static void main(String[] args) {
        new MyClass();
    }
    MyClass() {
        recurse("Hello");
    }
    String recurse(String s) {
        if (s == null)
            return null;
        int r = RandomNumbers.getRandomInt();
        int n = s.length();
        if (r % 2 == 0)
            return recurse(s.substring(0, n / 2));
        else {
            return recurse(s.substring(n / 2, n));
        }
    }
}
```

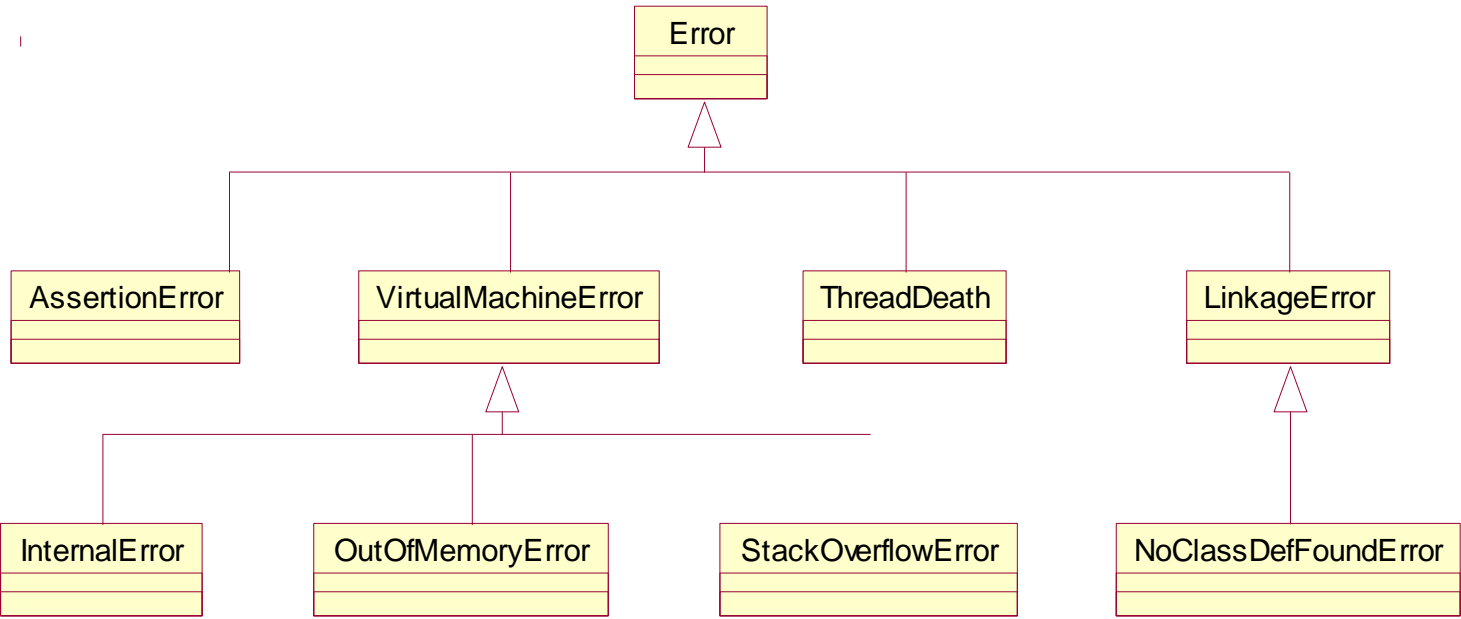
Running this code leads to the following output:

```
Exception in thread "main" java.lang.StackOverflowError
    at java.util.Random.nextInt(Unknown Source)
    at
pencil_4.probl.RandomNumbers.getRandomInt(RandomNumbers.java:20)
    at pencil_4.probl.MyClass.recurse(MyClass.java:15)
    at pencil_4.probl.MyClass.recurse(MyClass.java:20)
    at pencil_4.probl.MyClass.recurse(MyClass.java:20)
    at pencil_4.probl.MyClass.recurse(MyClass.java:18)
    at pencil_4.probl.MyClass.recurse(MyClass.java:20)
    at pencil_4.probl.MyClass.recurse(MyClass.java:20)
    at pencil_4.probl.MyClass.recurse(MyClass.java:20)
    at pencil_4.probl.MyClass.recurse(MyClass.java:18)
    at pencil_4.probl.MyClass.recurse(MyClass.java:20)
```

```
//output abbreviated
```



# The Error Hierarchy



## Other Unchecked Exceptions

1. Apart from `Error` objects, the other unchecked exceptions are those that belong to the hierarchy `RuntimeException`. As with `Error` objects, when one of these exceptions occurs, the JVM throws an object (in this case, of type `RuntimeException` or one of its subclasses) up the call stack.
2. Generally, if an exception of this kind is thrown, it indicates that the programmer has made a logic error in the code, and needs to fix it.
3. Examples:

```
NullPointerException    //an object needs to be initialized
ClassCastException      //improper cast needs to be fixed
ArrayIndexOutOfBoundsException //adjust bounds of a loop
NumberFormatException    //could arise from trying to turn a
                        //non-numeric String into an Integer
```

4. The techniques for handling exceptional conditions at runtime that need to be used for dealing with *checked exceptions* (discussed below) do *not* need to be done (and shouldn't be done) for unchecked exceptions.

## Examples

```
class Test1 {
    private Employee emp;
    public static void main(String[] args) {
        Test1 test = new Test1();

        //NullPointerException at runtime
        String name = test.emp.getName();
    }
}

class Test2 {
    public static void main(String[] args) {
        List employees = new ArrayList();
        employees.add(new Employee("Joe"));
        employees.add(new Employee("Tim"));

        //ClassCastException at runtime
        Employee first = (Manager)employees.get(0);
    }
}
```

## Throwing Runtime Exceptions

Because exceptions of type `RuntimeException` are unchecked, they can also be used by developers to indicate a problem that needs to be corrected (useful during development, not for production code).

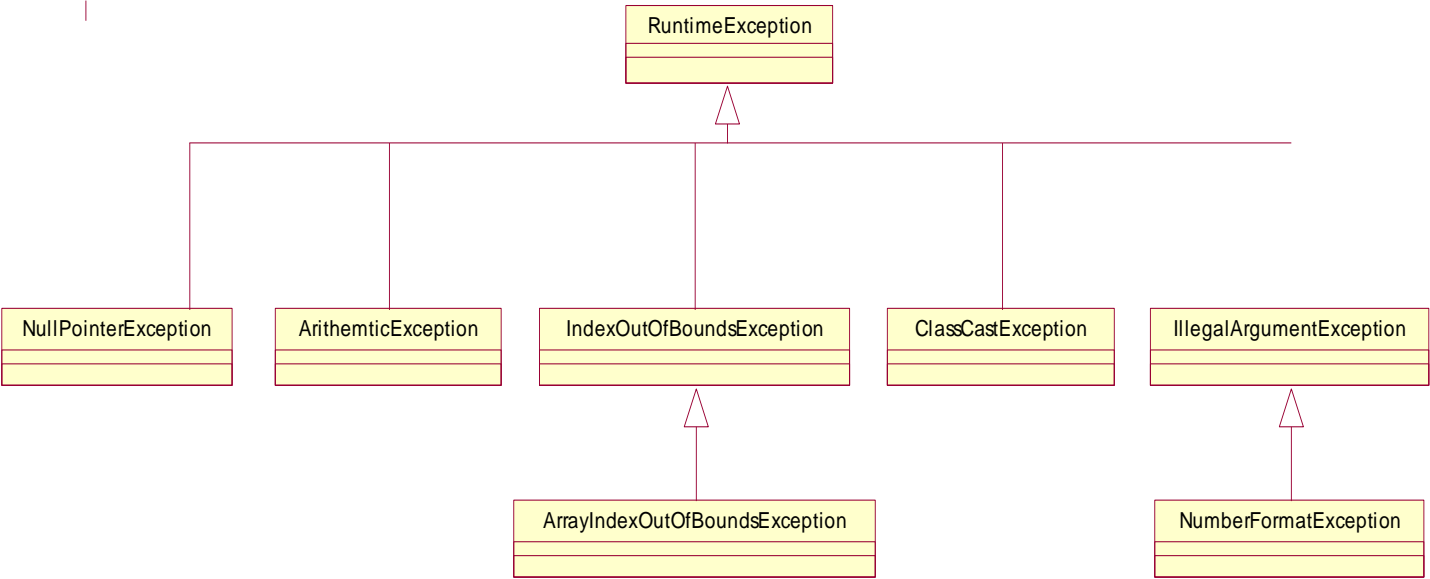
Two examples are `IllegalArgumentException` and `IllegalStateException`.

**Example:** (From the `Rational` class that was described in an earlier lab)

```
public Rational(int num, int denom) {  
    if (denom <= 0) {  
        throw new IllegalArgumentException("Denominator must be positive");  
    }  
    this.num=num;  
    this.denom=denom;  
}
```

[We will discuss what it means for exceptions to be “thrown” later in this lesson]

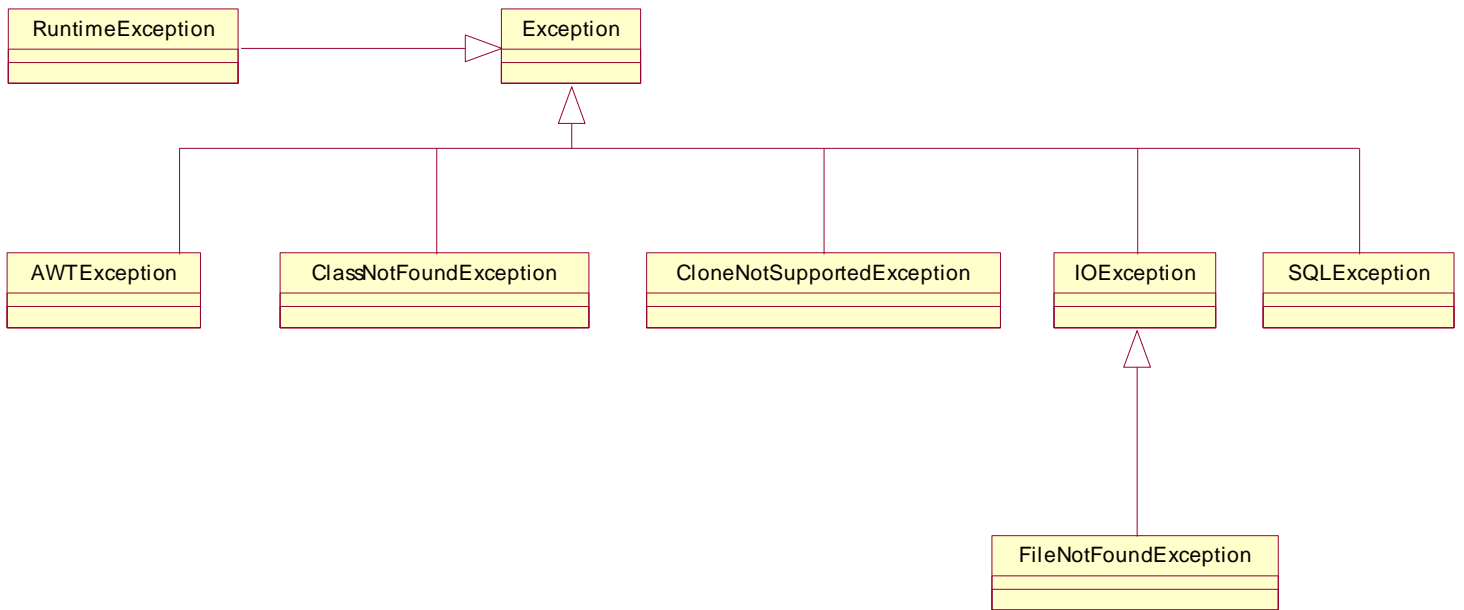
# Hierarchy of RuntimeException



## Checked Exceptions

1. This kind of exception is considered by the JVM to be the kind of error a developer must be prepared to handle. Examples:  
    CloneNotSupportedException  
    FileNotFoundException  
    SQLException  
    AWTException
2. Often, these exceptions arise when something goes wrong with the application's environment (can't find a file or class) or with an external system (an SQL query can't be executed).
3. The JVM expects you the developer to *handle* any exception of this type that could possibly be thrown, and will *issue a compiler error if you fail to do so*. (This is the reason for the terminology "checked exception".)

## Hierarchy of Checked Exceptions



## Dealing With Checked Exceptions

1. Every method in the Java API (and, as we discuss shortly, any user-created method) that is capable of throwing an `Exception` belonging to the Checked Exception Hierarchy indicates this fact with a `throws` clause in its declaration.
2. Examples:
  - a. The `clone` method in `Object`:  
`protected clone() throws CloneNotSupportedException`
  - b. The constructor of the class `FileWriter` (which is used for writing text to a file)  
  
`public FileWriter(File file) throws FileNotFoundException`



3. When you call a method that has a `throws` clause in its declaration, there are four ways of dealing it:
- a. Do not attempt to handle directly; instead, declare that *your* method **throws** this kind of exception too
  - b. Surround the calling code in a **try** block, and then do one of the following:
    - i. write exception-handling code in a **catch** block
    - ii. partially handle the exception in a **catch** block, and then *re-throw* the exception to allow other methods in the call stack to handle it further
    - iii. **throw** a new kind of exception from within the **catch** block

Note: Be aware of the difference between **throw** and **throws**

## Examples: Overriding the clone method

```
//Here exception is not handled directly
//Instead, declare that your method also
//throws this type of exception
class Employee implements Cloneable {
    . . . . .

    public Employee clone() throws CloneNotSupportedException{
        Employee copy = (Employee)super.clone();
        copy.hireDate = (Date)hireDate.clone();
        return copy;
    }

    . . . . .
}

class OtherClass {
    void callingMethod() {
        Employee e = new Employee();
        e.clone(); //compiler error - must enclose in try/catch
                  //or declare that the method throws
                  //CloneNotSupportedException
    }
}
```

```

//Here, we enclose the calling code in a try block, and then
//write exception-handling code in a catch block
class Employee implements Cloneable {
    . . . . .

    public Object clone(){
        try {
            Employee copy = (Employee)super.clone();
            copy.hireDate = (Date)hireDate.clone();
            return copy;
        }
        catch(CloneNotSupportedException ex) {
            System.err.println("Unable to make a copy");
        }
    }

    . . . . .
}

class OtherClass {
    void callingMethod() {
        Employee e = new Employee();
        e.clone(); //this is ok - Exception already handled
    }
}

```

```

//Here we enclose the calling code in a try block,
//then, in a catch block,
//write some exception-handling code
//and then re-throw the exception
class Employee implements Cloneable {
    . . . . .

    public Object clone() throws CloneNotSupportedException{
        try {
            Employee copy = (Employee)super.clone();
            copy.hireDate = (Date)hireDate.clone();
            return copy;
        }
        catch(CloneNotSupportedException ex) {
            System.err.println("Unable to make a copy");
            throw ex;
        }
    }

    . . . . .
}

class OtherClass {
    void callingMethod() {
        Employee e = new Employee();
        //must handle or pass on the exception
        try {
            e.clone();
        }
        catch(CloneNotSupportedException ex) {
            System.exit(1);
        }
        //this code will not execute if catch clause is invoked
        System.out.println(e.getName());
    }
}

```

```

//Here we enclose the calling code in a try block, then
//in catch block, optionally write
//write some exception-handling code
//and throw an application-specific Exception object
class Employee implements Cloneable {
    . . . . .

    public Object clone() throws ApplicationSpecificException{
        try {
            Employee copy = (Employee)super.clone();
            copy.hireDate = (Date)hireDate.clone();
            return copy;
        }
        catch(CloneNotSupportedException ex) {
            String msg = ex.getMessage() +
                " Inside Employee.clone()."
            throw new ApplicationSpecificException(msg);
        }
    }

    . . . . .
}

class OtherClass {
    void callingMethod() {
        Employee e = new Employee();
        //must handle or pass on the exception
        try {
            e.clone();
        }
        //compiler error - no such Exception is thrown
        //catch(CloneNotSupportedException ex) {

        catch(ApplicationSpecificException ex) {
            System.err.println("Call for help");
        }
        //other code will execute since catch clause
        //does not force an exit of the method
        System.out.println(e.getName());
    }
}

```

## What Happens in Each Case

1. Whenever an exception is thrown at runtime, the JVM looks to see if the active method has a `catch` clause whose `Exception` type matches the type of the thrown `Exception`. If not, it moves up the call stack to see if any calling methods provide a `catch` clause with a match.
2. Two possibilities:
  - a. If the method is declared with a `throws` clause, as in  

```
... throws XXException
```

then if an `Exception` of type `XXException` is thrown at runtime (and no `catch` clause has been provided for this type of `Exception`), the `Exception` object is passed up to the caller of this method.
  - b. If `try/catch` blocks have been provided, and the `catch` block's parameter matches `XXException`, then:
    - i. the program skips the remainder of code in the `try` block
    - ii. the program executes the code in the `catch` block

3. The code inside a `catch` block may

- a. gracefully handle the error condition – in which case the program will continue to run immediately after the `catch` block, or
- b. cause the application to terminate (using `System.exit()`), or
- c. re-throw the `Exception` that it just caught, or
- d. throw a new `Exception` of a different type

In cases b-d, the method immediately exits (unless there is a `finally` block—see below)

## MAIN POINT

Methods whose declaration includes a *throws* clause can be called by another method only if the calling method is declared with the same *throws* clause, or if a try/catch block is included to catch any of the declared exceptions that are thrown. This phenomenon is reminiscent of the Principle of Diving: once the initial conditions have been met, a correct dive into the depths occurs automatically. (The *throws* clause is the initial condition; the compiler then automatically requires additional coding in order to handle exceptions that may occur.)



## Summary of Exception Types

1. *Errors*. When an `Error` is thrown, it indicates an internal JVM error or other problem beyond the control of the developer. No attempt should be made to catch `Errors` and typically, no adjustment to the code needs to be done to prevent them (*except* for `StackOverflowError`, which is usually thrown because of an illegal recursion).
2. *Other unchecked exceptions* are thrown as objects of type `RuntimeException`, or one of its subclasses. These exceptions indicate a programming error needs to be fixed (like `NullPointerException`, `ClassCastException`, and `ArrayIndexOutOfBoundsException`). These objects should not be "caught" (i.e. used in conjunction with `try/catch` blocks), though for debugging purposes, this can be done.
3. *Checked exceptions* are exceptions that are subclasses of `Exception` but that are not part of the `RuntimeException` hierarchy. They must be dealt with in code by the developer. Failure to write such code results in a compiler error. Each call of a method that declares that it `throws` such an exception must either explicitly handle (in a `try/catch` block) exceptions that may arise from the call, or must pass the exception object up the call stack (using a `throws` declaration).

## Using/Creating Exception Classes

1. Sometimes in designing/coding an application, you may wish to indicate that an error condition has arisen, and you may find that one of Java's pre-defined exception classes will provide a sensible implementation.

**Example:** You have a method `readData` that reads in a file and one day, for a file whose header promised

`Content-length: 1024`  
you discover that the end of file is reached after only 733 characters.

So you equip `readData` with an exception. You read in the Java API docs that the `EOFException`

*Signals that an EOF has been reached unexpectedly during input*

You rewrite your `readData` method as follows:

```
String readData(Scanner in) throws EOFException {
    . . .
    while(true) {
        if(!in.hasNext()) { //EOF encountered—may be ok
            if(actualLen < PROMISED_LEN){ //something bad has happened
                String msg =
                    "expected " + promisedLen +
                    " but got only " + actualLen;
                throw new EOFException(msg);
            }
            else{
                . . .
            }
        }
    }
}
```

2. Usually, the best solution is to create your own `Exception` class. Define so that it is a subclass of `Exception`.

`Exception` has two main constructors – a default constructor and a one-argument constructor (of `String` type) designed to store an error message. Typically, you override both of these:

```
public class MyException extends Exception {
    public MyException() {
        super();
    }
    public MyException(String msg) {
        super(msg);
    }
}
//usage:
//throw new MyException("An exception has occurred");
//OR
//try {
//    . . .
//}
//catch(MyException e) {
//    System.out.println(e.getMessage()); //read stored msg
//}
```

## **Best Practices: When to Handle, When to Throw, When to Log**

### **Which Class Should Handle An Exception?**

- Exceptions are thrown at the exact point during execution where a problem arises
- Exceptions should be handled by a class that has among its responsibilities the proper knowledge about what should be done.
- One or more classes in an application should be delegated the responsibility of knowing what to do in case an exception occurs. Often, this responsibility entails nothing more than displaying an appropriate message to the user if an exception occurs.

[illegible]

## **Best Practices: How to Set up Your Own Exception Classes**

This example illustrates a good programming practice: For production-level applications, it is good practice to pre-define a set of application-specific Exception classes as part of an overall error-handling policy. These classes should represent a simple classification of the kinds of errors that might occur, and a mapping of these to a classification of the kinds of information you want to log and/or present to the user.

Example: In a small application, you may need only two kinds of exception: a `UserException` and a `SystemException`.

`UserException`: When the user makes a mistake

`SystemException`: When something goes wrong that is not the user's fault

Once this design decision has been made, then all exceptions that could arise in the application would be caught and either a `UserException` or `SystemException` would then be thrown.

## Importance of Logging

1. When an exception occurs, it is usually important to record this fact for later review by interested parties (developers, business team, etc). Messages presented to the user or printed to the console are not adequate for this purpose. What is needed is a *Log file*.
2. Pattern
  - Log a warning or error message when the exception first occurs
  - Throw an appropriate `Exception` up call stack to appropriate controller
  - Controller either handles or creates a user exception with a user-appropriate message



## Using Oracle's Logger

1. jdk 1.4 introduced the `Logger` class

2. Create an instance like this:

```
private static Logger LOG =  
    Logger.getLogger("com.mycompany.myapp");
```

3. Permits setting of level (SEVERE, WARNING, INFO, FINE), handlers and formatters. Can be done using code but better to use properties file located in `<java_home>\jre\lib.` (See Demo)
4. When an event occurs during runtime that needs to be logged, insert a line like this:

```
LOG.warning("Unauthorized user has attempted " +  
            "perform an action.");
```

## Example

```
private static final Logger LOG =
    Logger.getLogger(Bookstore.class.getPackage());

Bookstore(String id){
    this.id = id;
}
int getNumBooks() throws BadIdException {
    if(!isBadId(id)){
        return numBooks;
    }
    else {
        LOG.warning(LOG_WARN_BAD_ID);
        throw new BadIdException(BAD_ID_MSG);
    }
}
```

## MAIN POINT

To use Exceptions effectively, when an Exception is thrown, a message should be *logged* so that the support team can review later; the Exception should be *thrown* up the call stack until a class that knows how to handle the Exception is reached; and this final class should *catch* and *handle* the Exception in an appropriate way (often, this means presenting an error message to the user). In a similar way, creation itself is structured in layers; the activity at each layer has its own unique set of governing laws; laws that pertain to one level or layer may no longer be applicable at another level.

## Some Syntax Rules For Try/Catch

1. Every use of `try` must have at least one corresponding `catch` (or `finally` – see below) clause.
2. When an exception object is thrown, it will be caught by the nearest catch clause for which the catch clause parameter matches the class of the exception object, or *is a superclass of this class*

Example: The following produces a compiler error – 2nd catch clause is unreachable

```
try {
    . . .
}
catch(IOException ex1) {
    . . .
}
catch(FileNotFoundException ex2) {
    . . .
}
```

The following however is legal

```
try {
    . . .
}
catch(FileNotFoundException ex1) {
    . . .
}
catch(IOException ex2) {
    . . .
}
```

This also explains why you never catch an "Exception" object directly – otherwise, you could end up catching all kinds of unchecked exceptions (by accident).

```
//bad programming style
try {
    . . .
}
catch(Exception e) {
    . . .
}
```

3. It is legal to have `try/catch` blocks inside other `try` blocks and inside other `catch` blocks (sometimes this is necessary)

```
try{
    try {
        . . .
    }
    catch(AnExceptionType ex1) {
        . . .
    }
}
catch(AnotherExceptionType ex2) {
    try {
        . . .
    }
    catch(ThirdExceptionType ex3) {
        . . .
    }
}
```

## The `finally` Keyword

- A `finally` clause can be introduced after all `catch` clauses.
- Any `finally` block is guaranteed to run after a `try/catch` block, even if a `return` or `break` occurs; even if another exception is thrown inside those blocks.
- Exception to the rule: If `System.exit()` occurs in one of the blocks, the `finally` clause is skipped.
- A `finally` clause is used to cleanup resources (like database connections, open files)

## **finally Exercise**

```
class FinallyTest{
    public static void test() throws Exception {
        try {
            // return;                // 1
            // System.exit(0);        // 2
            // throw new Exception("first");    // 3a
        }
        catch (Exception x){
            System.out.println(x.getMessage());
            // throw new Exception("second");    // 3b
        }
        finally {
            System.out.println("finally!");
        }
        System.out.println("last statement");
    }
    public static void main(String[] args){
        try{
            test();
        }
        catch(Exception x){
            System.out.println(x.getMessage());
        }
    }
}
```

## Program Output

0:  
 finally!  
 last statement

1:  
 finally!

2: no output

3a: first  
 finally!  
 last statement

3a + 3b:  
 first  
 finally!  
 second



## Examples of Proper Use of Java's Exception-Handling Model

### Needs Improvement

```
//Triangle constructor
public Triangle(double side1, double side2, double side3) {
    double[] arr = sort(side1,side2,side3);
    double x = arr[0];
    double y = arr[1];
    double z = arr[2];
    if(x + y < z) {
        System.out.println("Illegal sizes for a triangle:
                           "+side1+", "+side2+", "+side3);
        System.out.println("Using default sizes.");
        setValues(DEFAULT_SIDE,DEFAULT_SIDE,DEFAULT_SIDE);
        computeBaseAndHeight(DEFAULT_SIDE,
                              DEFAULT_SIDE,
                              DEFAULT_SIDE);
    }
    else {
        setValues(x,y,z);
        computeBaseAndHeight(x,y,z);
    }
}
//from Test class
public static void main(String[] args) {

    ClosedCurve[] objects = {new Triangle(4,5,6),
                             new Square(3),
                             new Circle(3)};

    //compute areas
    for(ClosedCurve cc : objects) {
        System.out.println(cc.computeArea());
    }

}
```

## Improved Version

```
Logger log=Logger.getLogger("closedcurve.good");
//Triangle constructor
public Triangle(double side1, double side2, double side3) {
    double[] arr = sort(side1,side2,side3);
    double x = arr[0];
    double y = arr[1];
    double z = arr[2];
    if(x + y < z) {
        log.warning("Illegal sizes of sides passed in");
        throw new IllegalArgumentException("Illegal sizes "+
            "for a triangle: "+
            "+side1+", "+side2+", "+side3);
    }
    else {
        setValues(x,y,z);
        computeBaseAndHeight(x,y,z);
    }
}
//from Test class
public static void main(String[] args) {
    ClosedCurve[] objects = null;
    Triangle t = null;
    try {
        t = new Triangle(4,5,6);
    }
    catch(IllegalArgumentException e) {
        String msg = e.getMessage();
        JOptionPane.showMessageDialog(this,
            msg,
            "Error",
            JOptionPane.ERROR_MESSAGE);

        System.exit(0);
    }
    objects = { t, new Square(3), new Circle(3)};
    //compute areas
    for(ClosedCurve cc : objects) {
        System.out.println(cc.computeArea());
    }
}
```

## CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

*Right action in the field of exception-handling*

1. If a Java method has a *throws* clause in its declaration, the compiler requires the developer to (write code to) handle potential exceptions whenever the method is called.
2. To handle exceptions in the best possible way, *logging* should occur as soon as an exception is thrown, and the exception should be re-thrown up the call stack until a method belonging to a class with an appropriate set of responsibilities is reached – and within this method, the exception should be *caught* and handled.

3. **Transcendental Consciousness:** TC is the home of all the laws of nature, the home of "right action".
4. ***Wholeness moving within Itself:*** Action in the state of Unity Consciousness is spontaneously right and uplifting to the creation as a whole.

