

Lecture 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.”

Outline of Topics

- What Is Exception-Handling All About?
- An Object-Oriented Error-Handling Strategy
- Syntax Rules For Try/Catch
- Classification of Error-Condition Classes
 - Objects of Type Error
 - Other Unchecked Exceptions
 - Checked Exceptions
- The finally Keyword
- Using/Creating Exception Classes
- Best Practices: When to Handle, When to Throw

What Is Exception-Handling All About?

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

- All error conditions should be handled in one of two ways:

1. Return to a safe state and enable the user to execute other commands

Example: A user accidentally inputs incorrect data, such as an incomplete phone number – the application should ask the user to try again

2. Allow the user to save all work and terminate the application gracefully

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?

An Object-Oriented Error-Handling Strategy

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

- An error of any kind is represented as a special kind of object
- 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)
- 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.

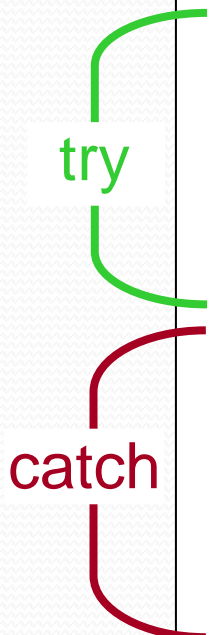
Not Catching Exceptions

```
Scanner scanner = new Scanner(System.in);  
System.out.println("Enter integer:");  
int number = scanner.nextInt();
```

Error message for invalid input

```
Exception in thread "main" java.lang.InputMismatchException  
    at java.util.Scanner.throwFor(Scanner.java:819)  
    at java.util.Scanner.next(Scanner.java:1431)  
    at java.util.Scanner.nextInt(Scanner.java:2040)  
    at java.util.Scanner.nextInt(Scanner.java:2000)  
    at Ch8Sample1.main(Ch8Sample1.java:35)
```

Catching an Exception

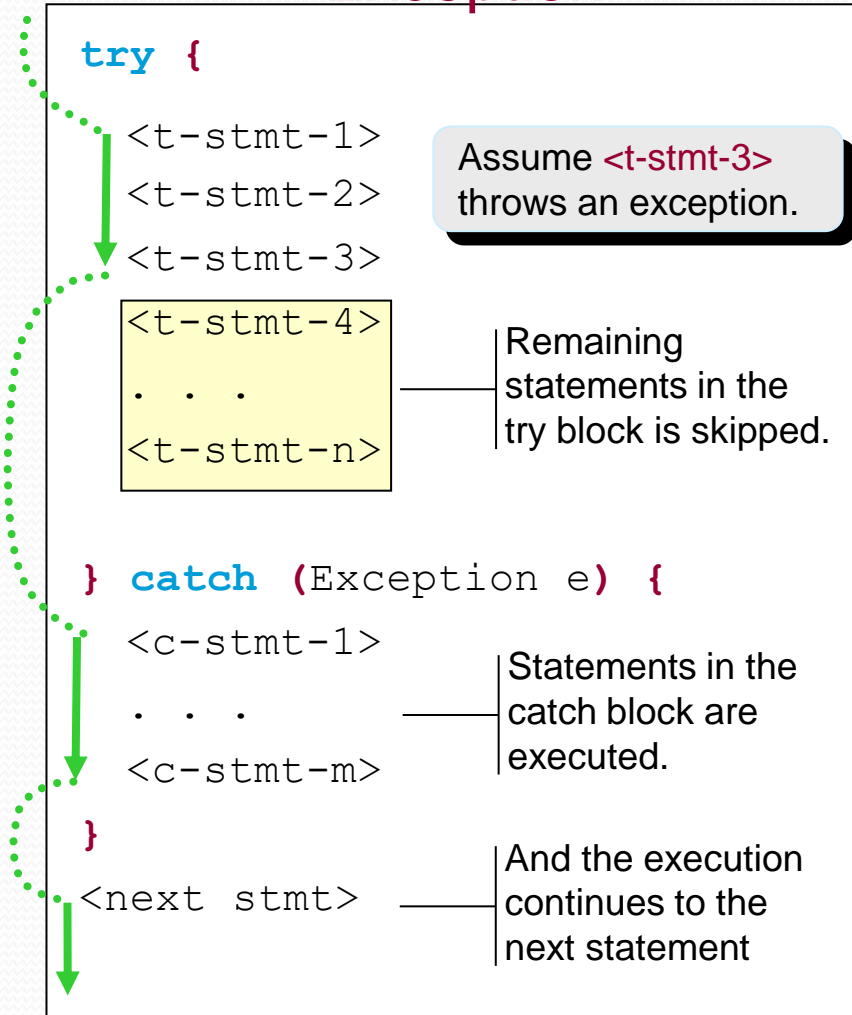


The diagram illustrates the structure of a try-catch block. A green bracket on the left side groups the `try` block, which contains the code `age = scanner.nextInt();`. A red bracket on the left side groups the `catch` block, which contains the code `System.out.println("Invalid Entry. " + "Please enter digits only");`. The `try` and `catch` keywords are also highlighted in green and red respectively.

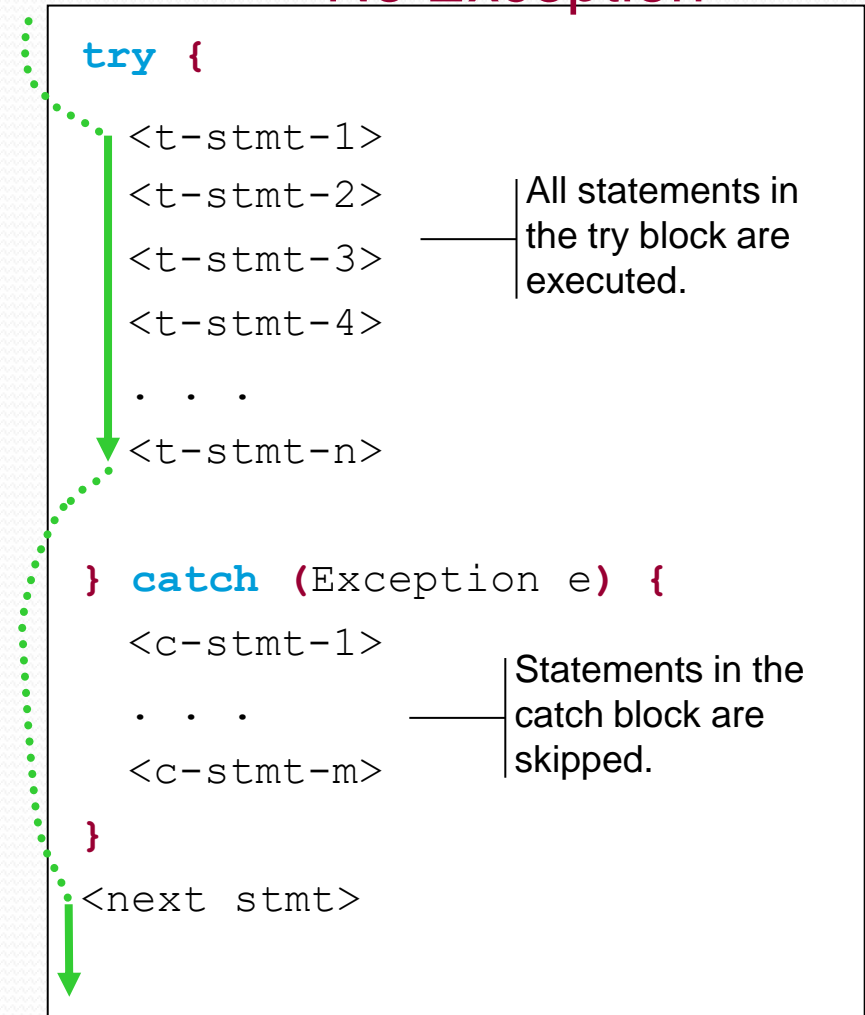
```
try {  
    age = scanner.nextInt( );  
} catch (InputMismatchException e) {  
    System.out.println("Invalid Entry. "  
        + "Please enter digits only");  
}
```


try-catch Control Flow

Exception



No Exception



Getting Information

- There are two methods we can call to get information about the thrown exception:
 - **getMessage**
 - **printStackTrace**

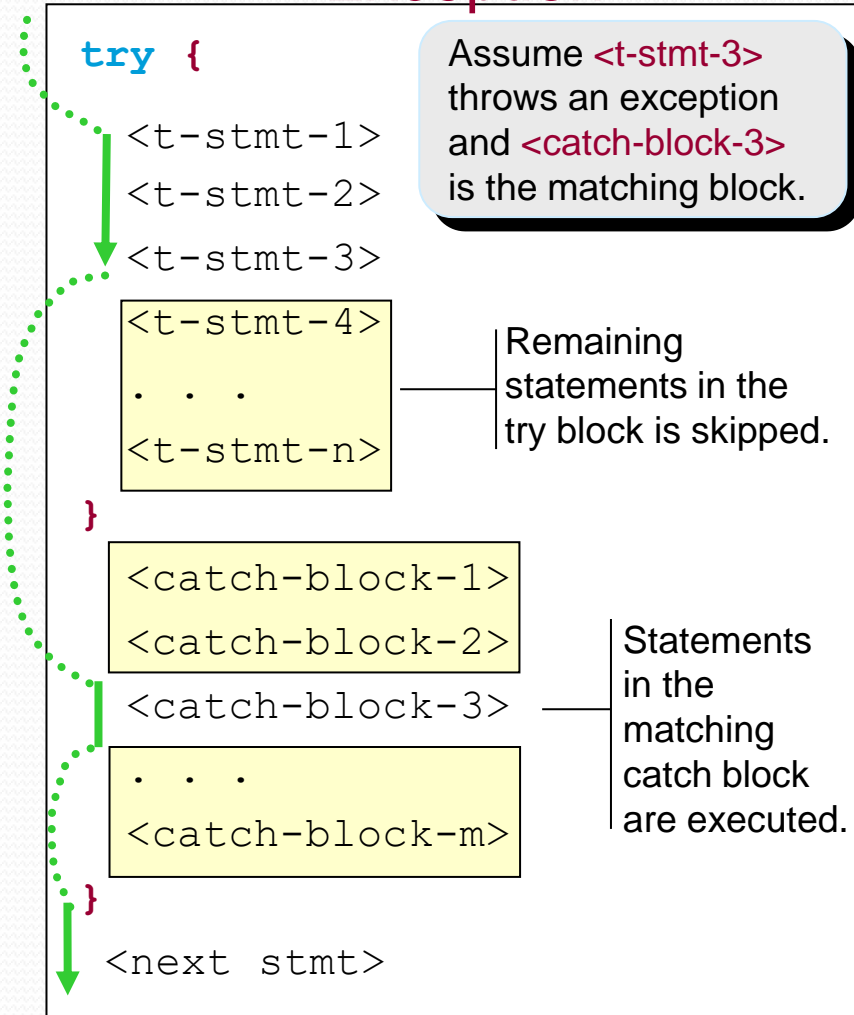
```
try {  
    . . .  
} catch (InputMismatchException e) {  
    System.out.println(e.getMessage());  
    //or e.printStackTrace();  
}
```

Multiple catch Blocks

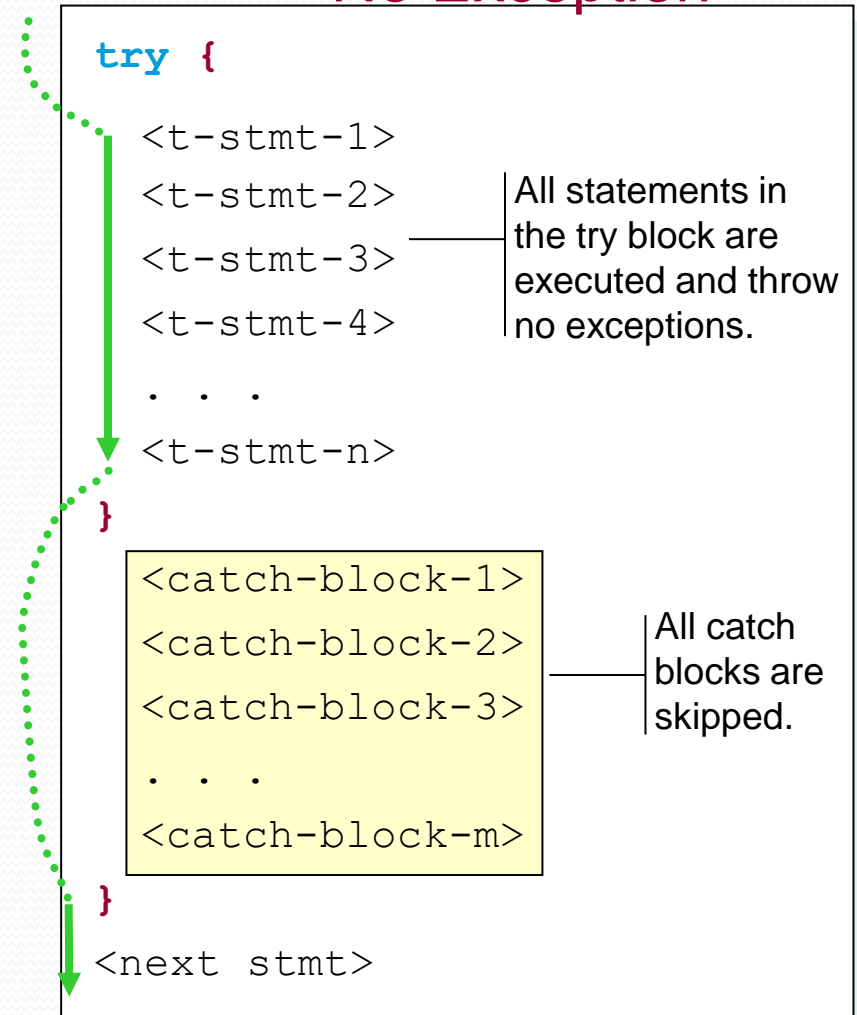
- A single try-catch statement can include multiple catch blocks, one for each type of exception.

```
try {  
    . . .  
    age = scanner.nextInt( );  
    . . .  
    FileReader f = new FileReader("filename.txt");  
    . . .  
} catch (InputMismatchException e) {  
    . . .  
} catch (FileNotFoundException e) {  
    . . .  
}
```

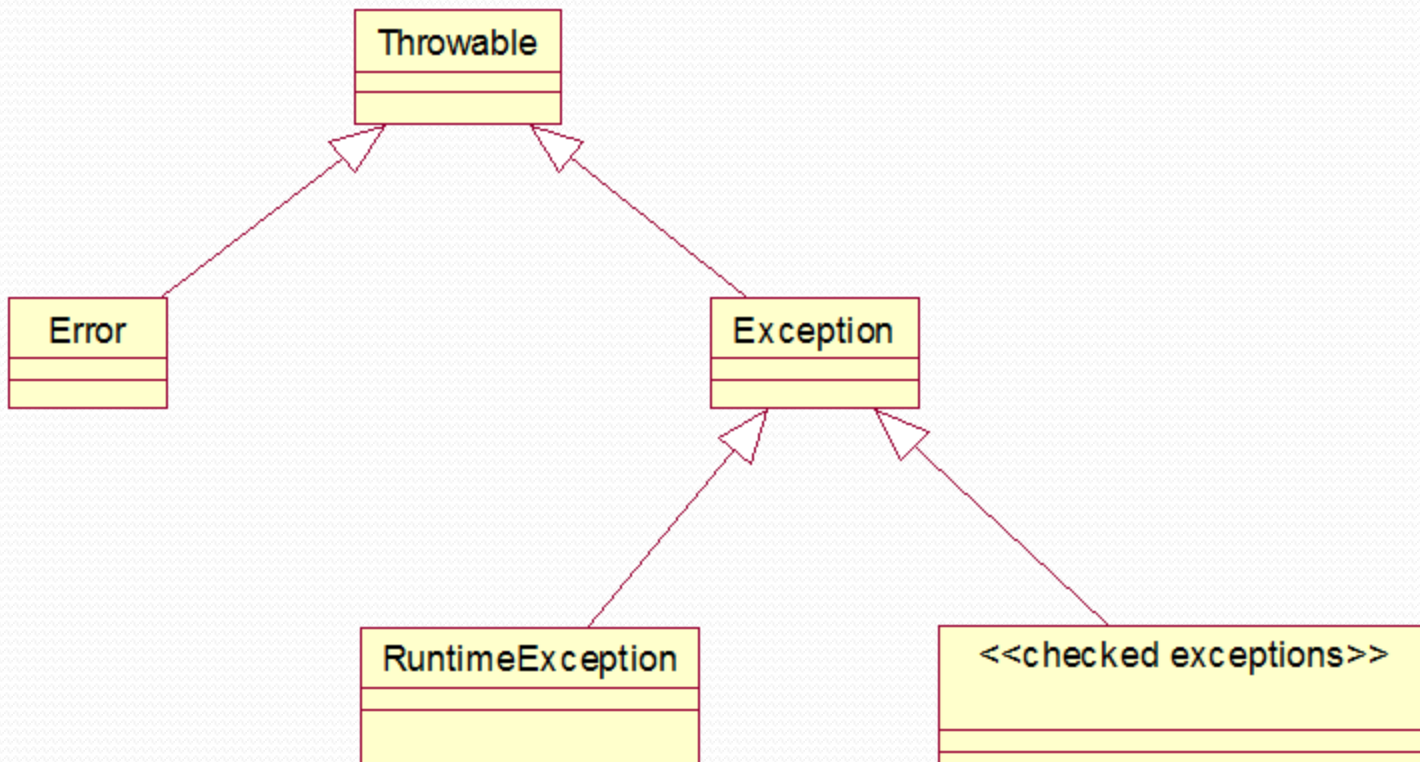
Exception



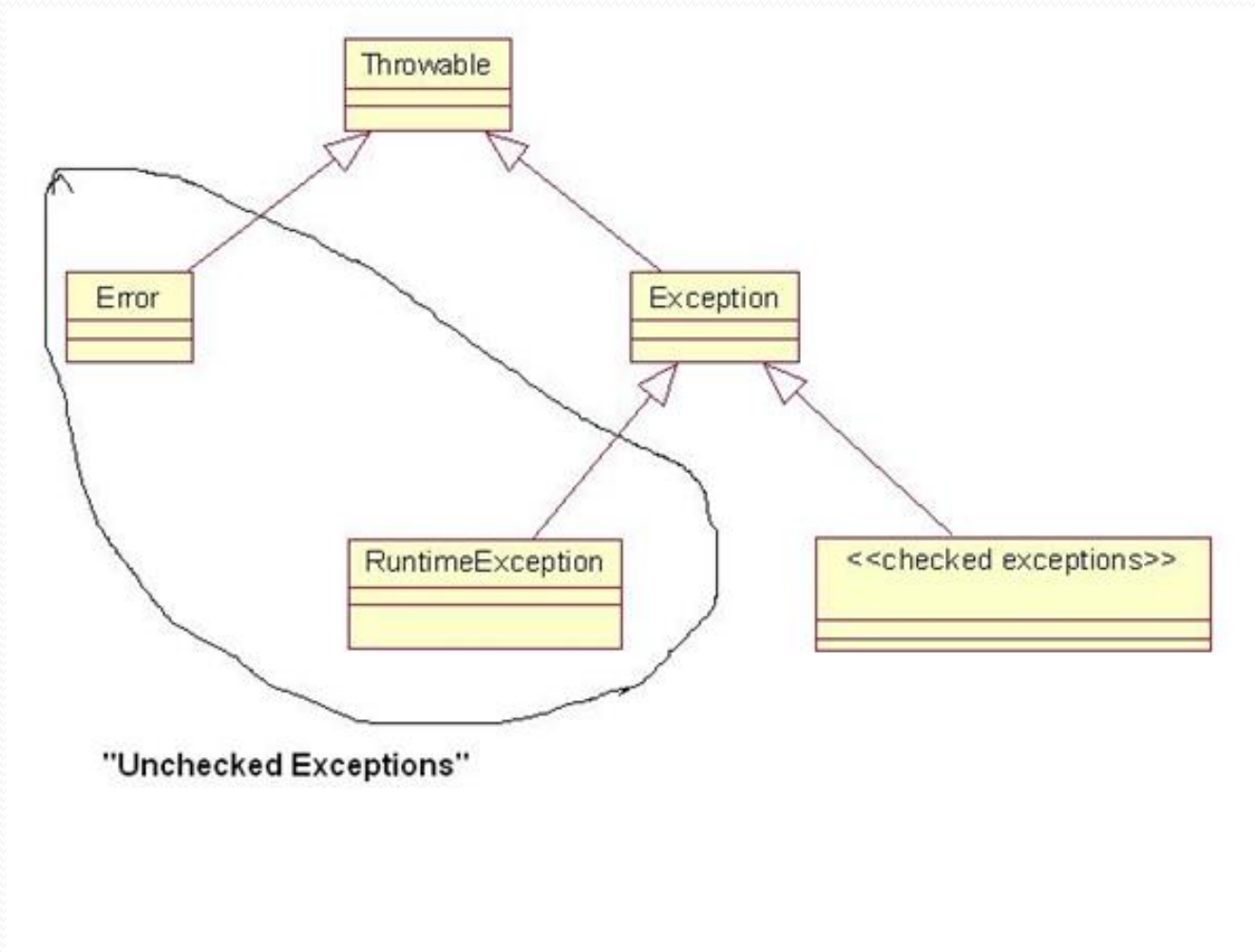
No Exception



The Hierarchy of Classes That Represent Error Conditions



Classification of Error-Condition Classes



Objects of type Error

- 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.
- No handling necessary. Errors of this kind indicate conditions that cannot be resolved in a "catch" block, so developers do not attempt to handle Errors that might be thrown in an application.

Example: `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.

- JVM displays stack trace When one of these errors occurs, the JVM *throws* an `Error` object. Since the Error is not caught, a stack trace is displayed in the console (showing the chain of method calls leading to the error).

Example: (From Recursion lecture). 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

Other Unchecked Exceptions

- ***RuntimeException Hierarchy.*** Subclasses of `RuntimeException` are also unchecked, and when one is thrown, a stack trace is displayed.
- ***RuntimeExceptions Indicate Failed Programming Logic.***

Examples:

```
NullPointerException    //uninitialized object  
ClassCastException      //improper cast  
ArrayIndexOutOfBoundsException //adjust loop bounds  
NumberFormatException    //e.g. try to change a  
                        //non-numeric string to an integer.
```

- ***How to Handle***
 - Developer does not attempt to catch these exceptions during execution
 - Instead, during development, before releasing software, developers handle runtime exceptions by "fixing the bugs" that give rise to them .

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  
        Manager first = (Manager) employees.get(0);  
    }  
}
```

Throwing Runtime Exceptions

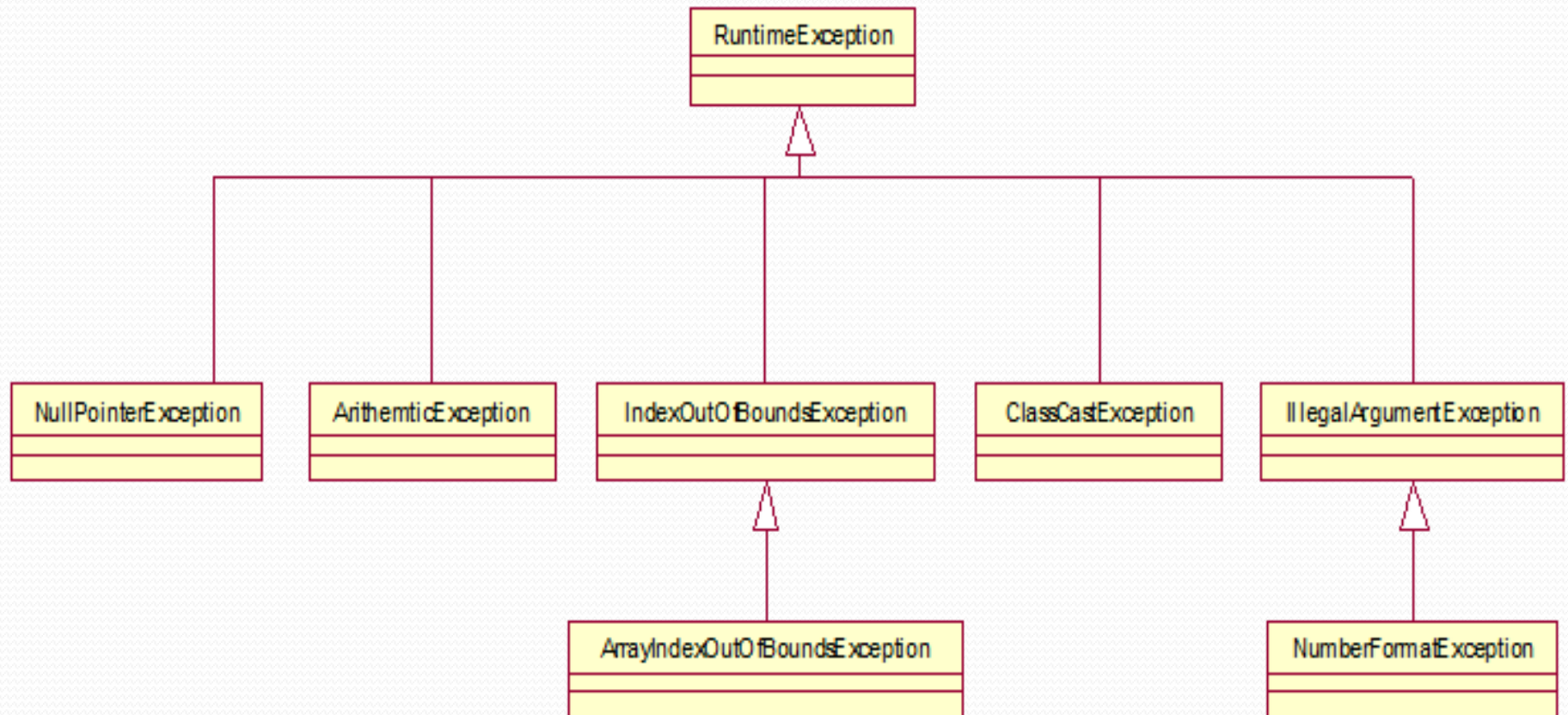
Because exceptions of type `RuntimeException` are unchecked, they can also be used by developers to indicate a problem that needs to be corrected (useful during development, not for production code).

Another example is `IllegalArgumentException`.

Example:

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

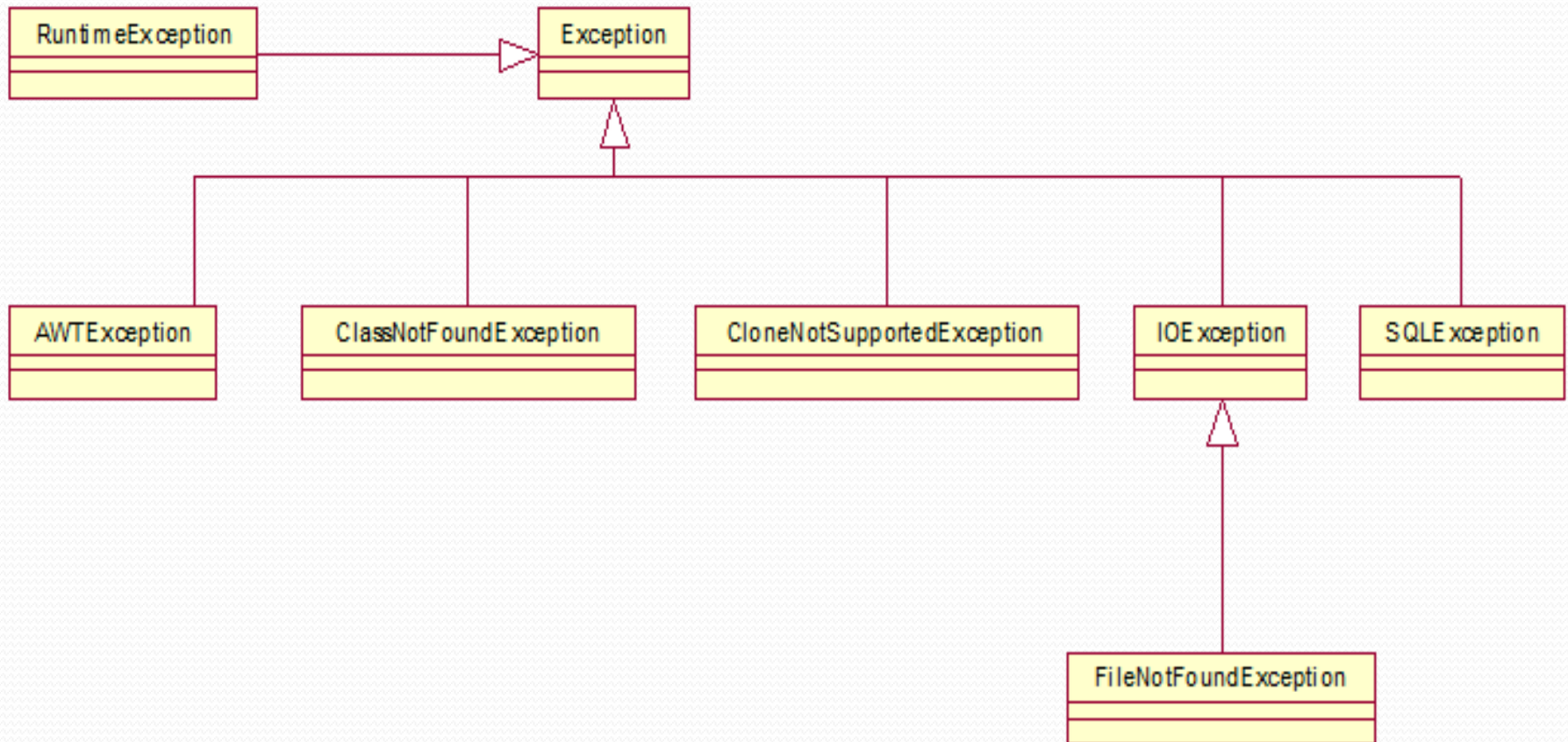
Hierarchy of RuntimeException



Checked Exceptions

- 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`
- 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).
- The JVM expects 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

- 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.

- Examples:

- The `clone` method in `Object`:

```
protected clone() throws CloneNotSupportedException
```

- The constructor of the class `FileWriter` (which is used for writing text to a file)

```
public FileWriter(File file) throws FileNotFoundException
```


Four Ways to Deal with Checked Exceptions

1. Do not attempt to handle directly; instead, declare that *your* method **throws** this kind of exception too
2. Surround the calling code in a **try** block, and then do one of the following:
 - a. write exception-handling code in a **catch** block
 - b. 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
 - c. **throw** a new kind of exception from within the **catch** block

Demo: `lesson12.checkedexceptions`

Question: What is the difference between **throw** and **throws**?

What Happens in Each Case

1. Whenever a checked 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:
 - 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.
 - If `try/catch` blocks have been provided, and the catch block's parameter matches `XXException`, then:
 - the program skips the remainder of code in the `try` block
 - the program executes the code in the `catch` block

(continued)

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

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, close files)

finally Exercise

Demo: `lesson12.finallyexercise`

```
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

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.)

Using/Creating Exception Classes

- Sometimes in designing/coding an application, you may wish to indicate that an error condition has arisen, and you cannot find one of Java's pre-defined exception classes that serve your purpose.

User-Defined Exception Classes

- Any class which extends Exception class will be a user defined **Checked** exception class. Any class which extends RuntimeException will be **Unchecked** exception class.
- 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);  
    }  
}
```

User-Defined Exception Classes

//throw one of your exceptions like this

```
throw new MyException("An exception has occurred");
```

//catch one like this:

```
try {
```

```
    . . .
```

```
}
```

```
catch(MyException e) {
```

```
    System.out.println(e.getMessage()); //read stored msg
```

```
}
```

Best Practices:

When to Handle, When to Throw

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 and log the exceptions.

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

```
//Triangle constructor
public Triangle(double side1, double side2, double side3)
    throws IllegalArgumentException {
    double[] arr = sort(side1,side2,side3);
    double x = arr[0];
    double y = arr[1];
    double z = arr[2];
    if(x + y < z) {
        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(IllegalTriangleException 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());
    }
}
```

Summary of Exception Types

- *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).
- *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.
- *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).

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.

