LESSON 12 EXCEPTION-HANDLING:

Nature is Structured in Layers

WHOLENESS STATEMENT

Exception handling can be organized in layers with each handler re-establishing a safe state at a given level so the user can continue executing or terminate gracefully. The Universe is also structured in layers with different laws of nature organizing and establishing the order at each level in creation.

EXCEPTIONS

An Exception is an abnormal condition in which the normal execution of code gets hampered.

An exception is an error that occurs at run time.

Example

```
int x = 10;
int y = 0;
System.out.println("Result :" + (x/y));
```

Exception in thread "main" java.lang.ArithmeticException: / by zero

```
String[] a = {"Raju","Sutha","Jai"};
System.out.println(a[3]);
```

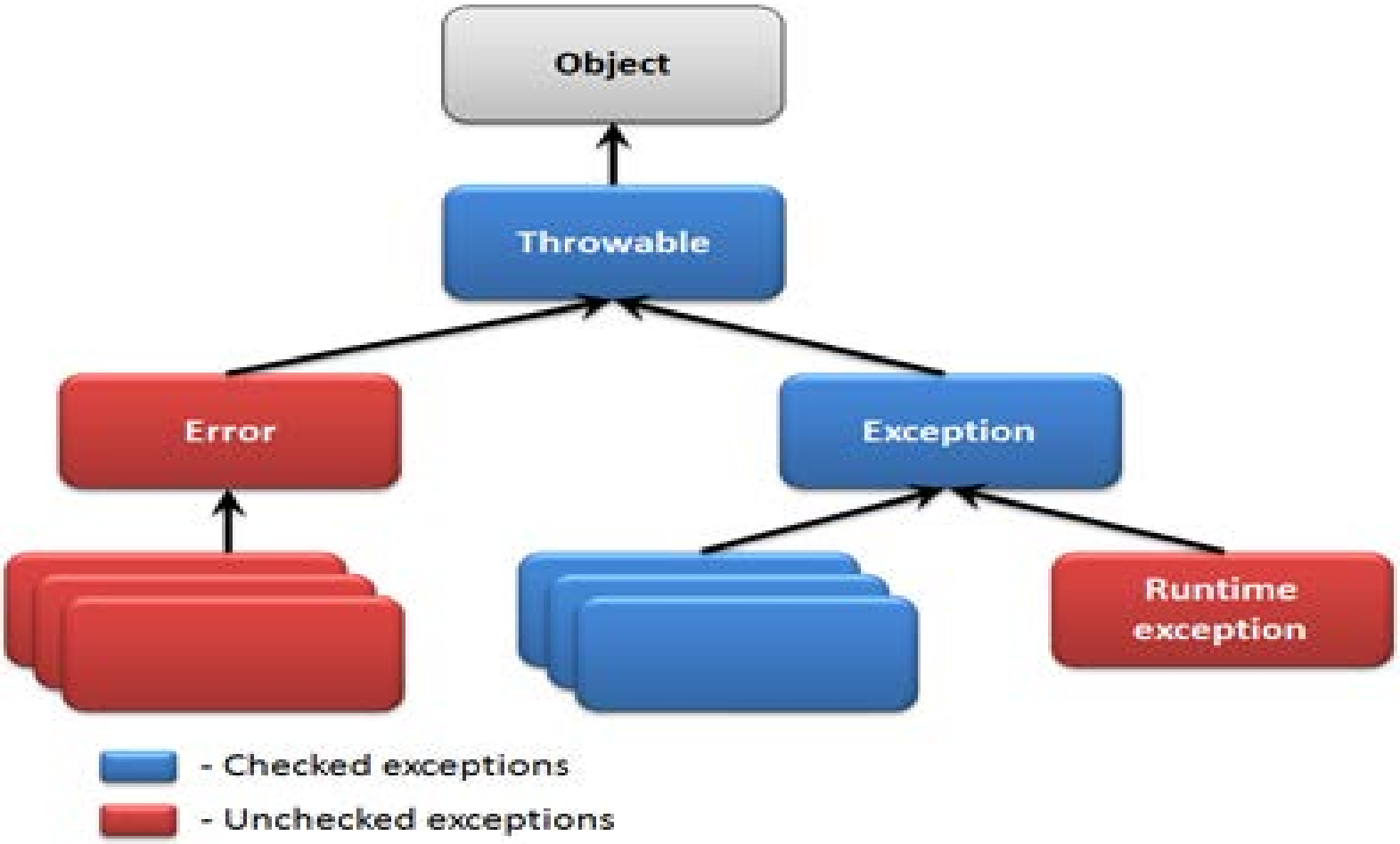
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException

Exception Handling

- Exception handling *enables a program to deal with exceptional situations and continue its normal execution.*
- *Runtime errors* occur while a program is running if the JVM detects an operation that is impossible to carry out.
- In Java, runtime errors are thrown as exceptions. An *exception* is an object that represents an error that prevents execution from proceeding normally. If the exception is not handled, the program will terminate abnormally.
- How can you handle the exception so that the program can continue to run or else terminate gracefully? This section deals this problem.

Exception Hierarchy

The exception classes can be classified into three major types: system errors, exceptions, and runtime exceptions.

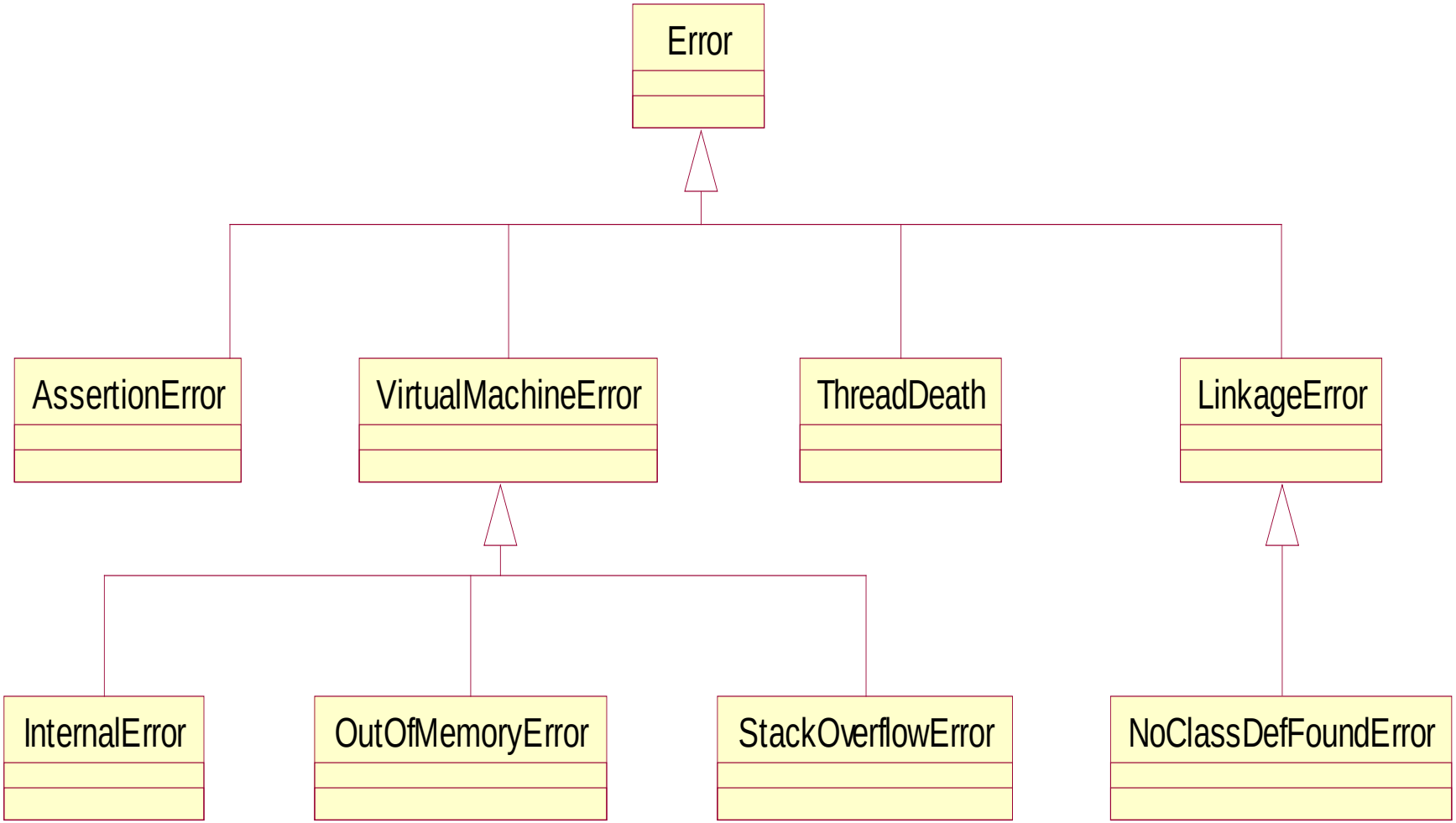


Errors

- 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.
- 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.]
- 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*).

```
// Create StackOverFlow Error due to repetitive self calls
public class ErrorDemo {
public void method1()
{
this.method2();
}
public void method2()
{
this.method1();
}
public static void main(String[] args) {
ErrorDemo ed = new ErrorDemo();
ed.method1();
}
}
```

The Error Hierarchy



Exceptions

- Exceptions are represented in the Exception class, which describes errors caused by your program. These errors can be caught and handled by your program.
- Unlike Errors, Exceptions can be handled to prevent the program to automatically terminate.
- It can be handled using try and catch blocks.
- An important subclass of Exception is RuntimeException, which is used to represent various common types of run-time errors.

Checked Exceptions vs. Unchecked Exceptions

RuntimeException and their subclasses are known as *unchecked exceptions*. It may occur at run time. Programmer need to handle it.

All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions. Throws class is an example of this category.

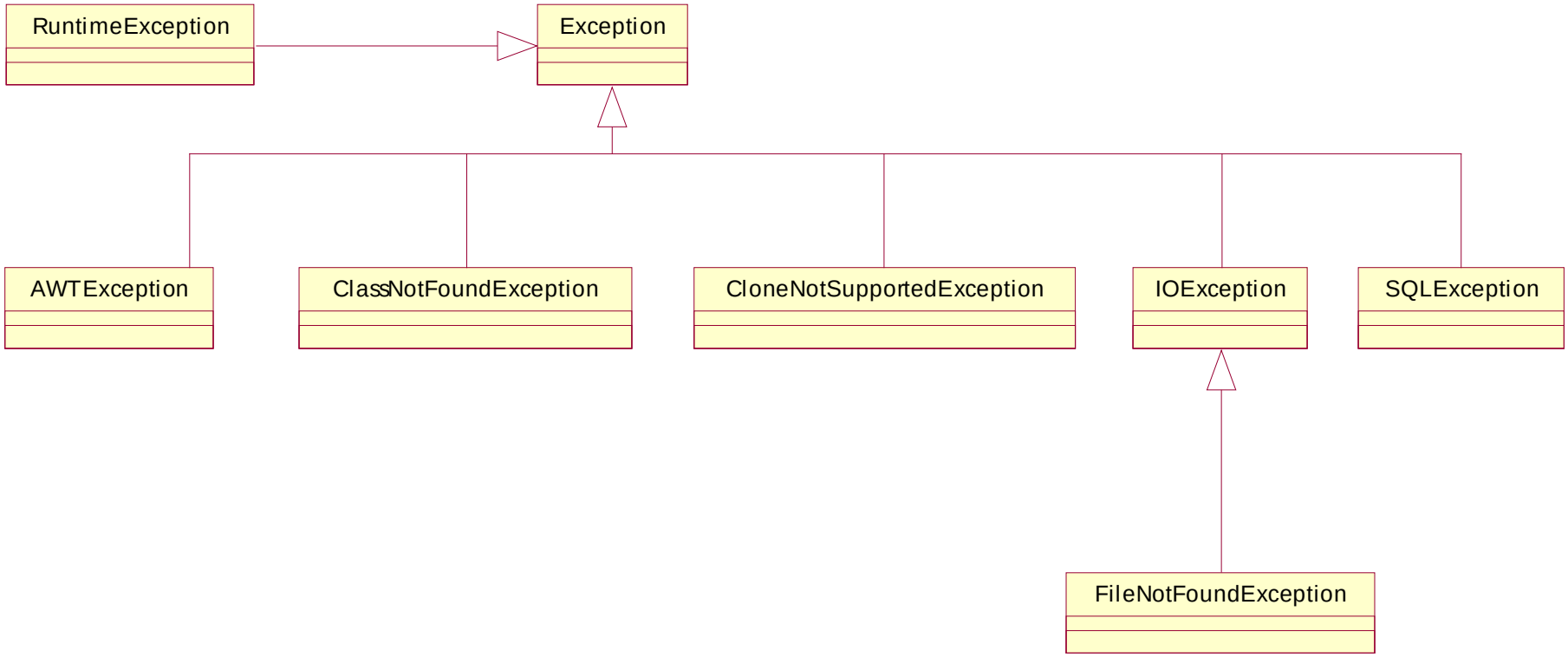
```
public static void main(String[] args) throws FileNotFoundException
{
File file = new File("scores1.txt");

    // Create a Scanner for the file
    Scanner input = new Scanner(file);
}
```

Checked Exceptions

- Method or constructor may throw an exception. The compiler will confirm at compile time that the method includes code that might **throw** an exception.
- Examples:
  - CloneNotSupportedException
  - FileNotFoundException
  - ClassNotFoundException
  - SQLException
- JVM expects the developer to *handle* any exception of this type
  - will *issue a compiler error if a program fails to do so*

Hierarchy of Checked Exception



Checked Exception defined in java.lang

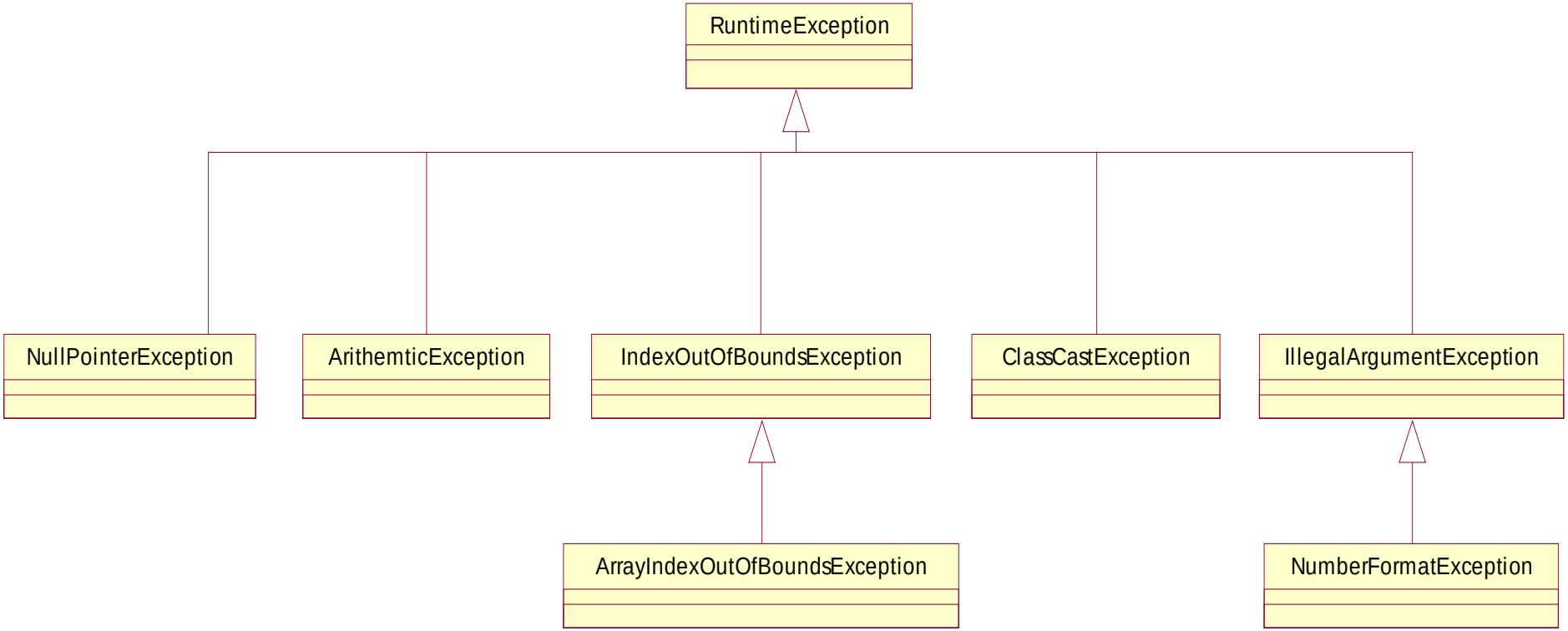
Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

Unchecked Exceptions

- Belong to the hierarchy RuntimeException
- Programmer needs to fix it
- Examples:
  - 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.

- Unchecked exceptions can occur anywhere in the program.
- It can be handled with the help of try-catch block.

Hierarchy of Runtime Exception



Unchecked Exception defined in java.lang

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found. (Added by J2SE 5.)
UnsupportedOperationException	An unsupported operation was encountered.

Exception Handling Fundamentals

- Java exception handling is managed via five keywords: **try**, **catch**, **throw**, **throws**, and **finally**.
- Program statements that you want to monitor for exceptions are contained within a **try** block.
- Your code can catch this exception using **catch.**
- To manually throw an exception, use the keyword **throw**.
- an exception that is thrown out of a method must be specified as such by a **throws** clause.
- Example : public void **shutdown**() throws **IOException // Methods**

{ throw new IOException("Unable to shutdown"); }// Line of code

- Any code that absolutely must be executed upon exiting from a **try** block is put in a **finally** block.

Using try and catch - Syntax

```
try {
// code might throw exceptions
}
catch (Excep class1 exOb) {
// handler for Excep class1
}
catch (Excepclass2 exOb) {
// handler for Excepclass2
}...
finally(optional){
// Release some resources here
}
```

The *finally* Keyword

- *The* **finally** *clause is always executed regardless whether an exception occurred or not.*
- A finally block is guaranteed to run after any try/catch block.
  - even if a return or break occurs
  - even if another exception is thrown inside those blocks
- A finally clause is used to cleanup resources (like database connections, closing files)

```
public class ExampleDemo1 {
                        public static void main(String[] args) {
                        int nums[] = new int[4];
                        try {
                        //1
                        System.out.println("Before exception is generated.");
                        // Generate an index out-of-bounds exception.
                        //2
                        nums[4] = 10;
                        //3
                        System.out.println("End of try");
                        }
                        catch (ArrayIndexOutOfBoundsException exc) {
                        // catch the exception
                        //4
                        System.out.println("Index out-of-bounds!");
                        }
                        finally
                        {
                        //5
                        System.out.println("End of try catch block");
                        }
                    }
}
```

- Statements 1,2,4 & 5 will execute.

Throwing an Exception

- A program that detects an error can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*.

Example

IllegalArgumentException ex =

new IllegalArgumentException(**"Wrong Argument"**);

throw ex;

Or,

if you prefer, you can use the following:

throw new IllegalArgumentException(**"Wrong Argument"**);

Example : ThrowDemo.java

```
public class ThrowExample {
static int Multiply(int num1, int num2){
      if (num2 == 0)
         throw new ArithmeticException("Second parameter
should not be zero");
      else
         System.out.println("Both parameters are correct!!");
      return num1/num2;
   }
   public static void main(String args[]){
      int res=Multiply(12,2);
      System.out.println(res);
      System.out.println("End of the Program");
   }
}
```

```
public class BalanceThrow {
public static void main(String[] args) {
int balance = 100, withdraw = 1000;
if (balance < withdraw)
{
UnsupportedOperationException e = new
UnsupportedOperationException("No money, Sorry");
throw e;
} else
{ System.out.println("Draw Money & enjoy, Have a good day.");
}
System.out.println("End");
}
}
```

Using throws

- In some cases, if a method generates an exception that it does not handle
- it must declare that exception in a **throws** clause.
- Here is the general form of a method that includes a **throws** clause.

  ret-type methName(*param-list*) throws *except-list* {

  // body

  }
- throws clause basically provides a broader design scope, wherein the originator is not handling the exception in it's own way, but allowing the exception to be handled differently by different methods calling it.
- See : ThrowsDemo

```
public class PriceListThrows {
private static final double[] price = {15.99, 27.88, 34.56, 45.89};
public static void displayPrice(int item) throws IndexOutOfBoundsException
{
System.out.println("The price is $" + price[item]);
}
public static void main(String[] args) {
displayPrice(0);
displayPrice(1);
displayPrice(2);
displayPrice(3);
displayPrice(5);
}
}
```

User Defined Exception Classes

```
public class MyException extends Exception {
  public MyException() {
        super();
  }
  public MyException(String msg) {
        super(msg);
  }
}
```

See : UserException

Assertions

- An assertion is a Java statement that enables you to assert an assumption about your program.
- An assertion contains a Boolean expression that should be true during program execution.
- Assertions can be used to assure program correctness and avoid logic errors.
- Syntax : assert *assertion*; or
- assert *assertion* : *detailMessage*;

Executing Assertions Example

```
public class AssertionDemo {
  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);
    System.out.print("Enter a number between 0 and 10: ");
     int number = input.nextInt();
 // assert that the value is >= 0 and <= 10
 assert (number >= 0 && number <= 10) : "bad number: " + number;
 System.out.printf("You entered %d%n", number);}
}
```

When an assertion statement is executed, Java evaluates the assertion. If it is false, an AssertionError will be thrown.

Best Practices

- When To Throw
  - at the exact point during execution where a problem arises
- When To Handle
  - by a class that has among its responsibilities the proper knowledge about what should be done
- When To Throws
  - *Exception* could be thrown during the execution of the method.

Pitfalls

- The following is not good (illegal?) because IOException is a superclass of FileNotFoundException. Why?

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

Corrected

- The following is OK because all catch clauses are now reachable. Why?

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

Nesting is OK

- The following is sometimes necessary:

```
try {
        . . .
} catch(AnException ex1) {
        . . .
    try {
        . . .
    } catch(AnotherException ex2) {
        . . .
    }
}
```

Demo Code

- AgeInputVerification
- BalanceThrow
- CheckedExpDemo
- DemoStackTrace
- FinallyDemo
- MultiCatch
- MultipleCatch
- ThrowDemo
- Throwsdemo
- UserException

Importance of Logging

- 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.*

- 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

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);
                }
        }
```

- See Sakai : **Resources / LecturePPT / DemoCode / Lesson12 / logging**

Main Point

To use Exceptions effectively, a message should be *logged* so that the support team can review it later; the Exception should be *thrown* up the call stack until a class that knows how to handle the Exception is reached; this final class should *catch* and *handle* the Exception in an appropriate way. 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 not be applicable at another level.

CONNECTING THE PARTS OF KNOWLEDGE WITH THE WHOLENESS OF KNOWLEDGE

1. **If a Java method has a *throws* clause in its declaration, the compiler requires methods that call it to handle potential exceptions.**

2. **Exceptions should be handled as follows: *log* them as soon as the exception is thrown; re-throw the exception up the call stack to a controller method that understands how to handle it; the exception is then caught and appropriate action is taken.**

3. **Transcendental Consciousness**

is 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.