EXCEPTION

Exception

- An exception is an event, which occurs during the execution of a program, that disrupts the normal flow of the program's instructions.
- An exception is an abnormal condition that arises in a code sequence at run time.
- It is an object which is thrown at runtime.
- In other words, an exception is a run-time error.

- Runtime errors occur while a program is running if the JVM detects an operation that is impossible to carry out. Example:

```
public class Computer
{
    public static void main(String []args)
    {
        Scanner sc = new Scanner(System.in);
            System.out.println("Enter first no.: ");
            int a = sc.nextInt();
            System.out.println("Enter second no.: ");
            int b = sc.nextInt();
            int c = a/b;
            System.out.println("Result is: " + c);
    }
}
```

Exception occurs at this statement if user enters value 0 for “b” variable

```
Exception in thread "main" java.lang.ArithmeticException: / by zero
        at Pkg1.Computer.main(Computer.java:14)
```

EXCEPTION HANDLING

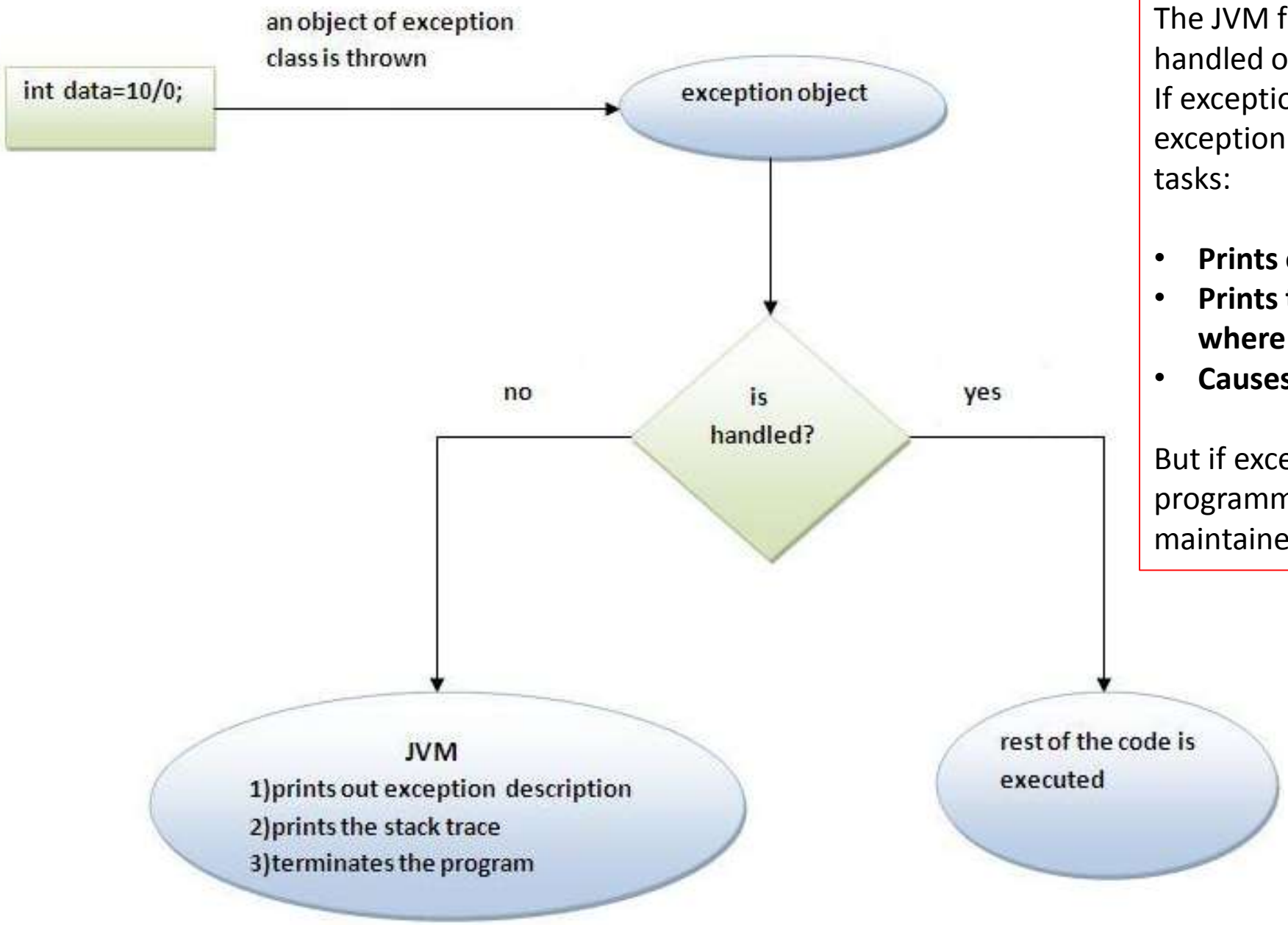
Exception Handling

- Exception Handling is a mechanism to handle runtime errors such as ClassNotFound, IO, SQL, Remote etc.

```
public class Computer
{
    public static void main(String []args)
    {
        Scanner sc = new Scanner(System.in);
        try
        {
            System.out.println("Enter first no.: ");
            int a = sc.nextInt();
            System.out.println("Enter second no.: ");
            int b = sc.nextInt();
            int c = a/b;
            System.out.println("Result is: " + c);
        }
        catch(ArithmeticException e)
        {
            System.out.println("Cannot divide by zero...");
        }
    }
}
```

Exception at this statement is handled & caught if user enters value 0 for “b” variable

Internal working of java try-catch block



The JVM firstly checks whether the exception is handled or not.
If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- **Prints out exception description.**
- **Prints the stack trace (Hierarchy of methods where the exception occurred).**
- **Causes the program to terminate.**

But if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

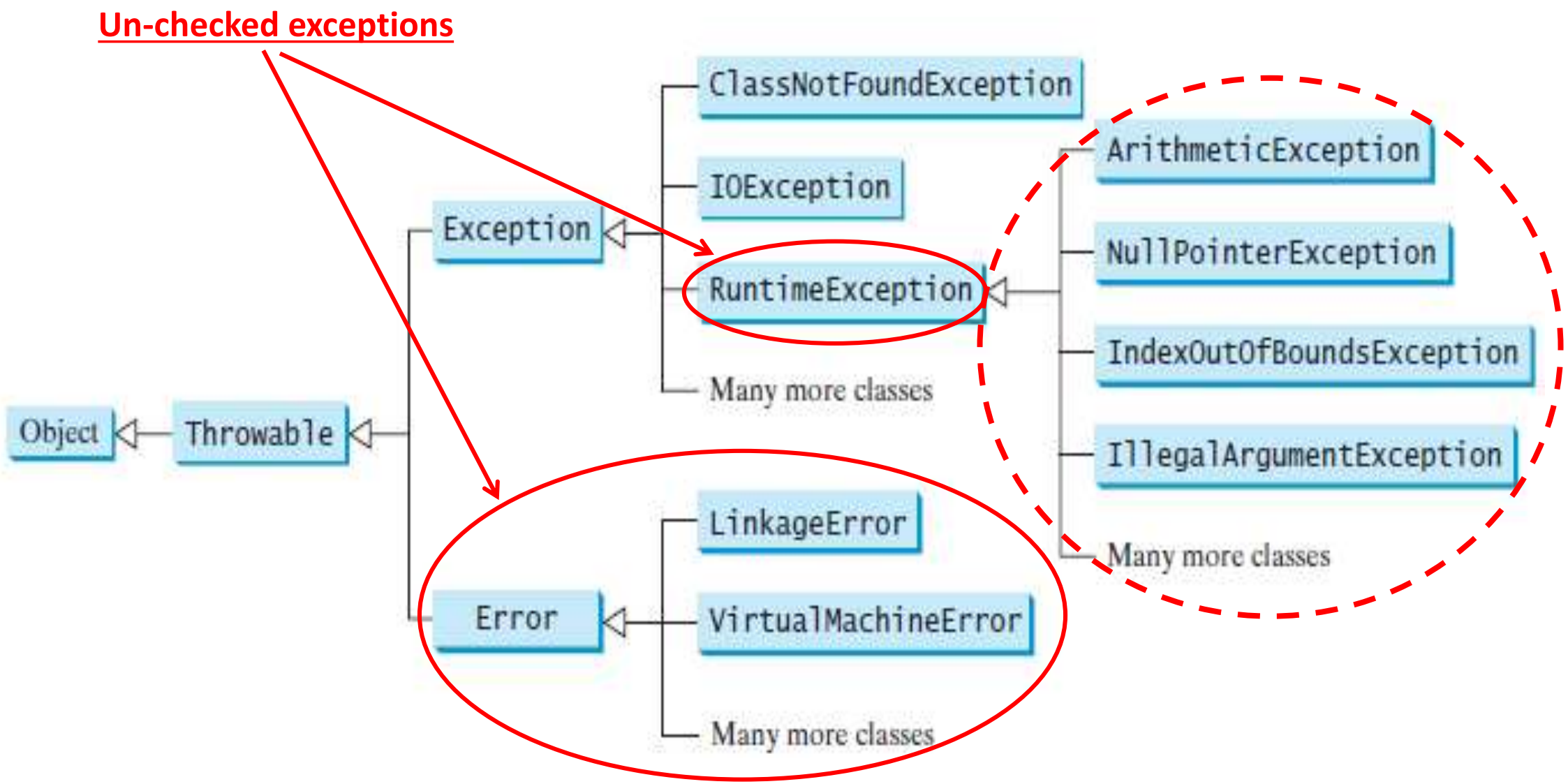
Advantage of Exception Handling

The core advantage of exception handling is **to maintain the normal flow of the application**. Exception normally disrupts the normal flow of the application that is why we use exception handling. Let's take a scenario:

```
statement 1;
statement 2;
statement 3;
statement 4;
statement 5;//exception occurs
statement 6;
statement 7;
statement 8;
statement 9;
statement 10;
```

Suppose there is 10 statements in your program and there occurs an exception at statement 5, rest of the code will not be executed i.e. statement 6 to 10 will not run. If we perform exception handling, rest of the statement will be executed. That is why we use exception handling in java.

Hierarchy of exception classes



Types of Exception

There are mainly two types of exceptions: checked and unchecked where error is considered as unchecked exception. The sun microsystem says there are three types of exceptions:

1. Checked Exception
2. Unchecked Exception
3. Error

Checked Exception

- **Checked Exceptions** are those, that have to be either caught or declared to be thrown in the method in which they are raised.
- **In other words, those exceptions whose handling is verified during compile-time.**

- It is a reminder by compiler to programmer to handle failure scenarios.
- For example, the below I/O statement has to be written in try block:

```
FileInputStream fp = new FileInputStream("C:/a.txt");
```

Un-handled exception. Compiler forces to write it inside try block.

- When to use:
  - Operation where chances of failure are more eg: IO operation, database access, networking operation, etc.

Following are some Examples of Checked Exception in Java library:

IOException

SQLException

DataAccessException

ClassNotFoundException

InvocationTargetException

Unchecked Exception

- **Unchecked exceptions** are those exceptions whose handling is not verified during compile-time, rather they are checked at runtime.
- It is direct sub-class of **RuntimeException**.

- For example, the below statement need not to be written in try block:

```
int val = a/0;
```

Compiler doesn't force to write it inside try block.

- Advantage: maintains code readability.
- They arise due to:
  - Programming errors (like accessing method of null object, accessing element outside array)

Here are few **examples of Unchecked Exception in Java** library:

NullPointerException

ArrayIndexOutOfBoundsException

IllegalArgumentException

IllegalStateException

Commonly used sub-classes of Exception

- ArithmeticException
- ArrayIndexOutOfBoundsException
- NumberFormatException
- NullPointerException
- IOException

Common scenarios where exceptions may occur

1) Scenario where ArithmeticException occurs

If we divide any number by zero, there occurs an ArithmeticException.

```
int a=50/0;//ArithmeticException
```

2) Scenario where NullPointerException occurs

If we have null value in any variable, performing any operation by the variable occurs an NullPointerException.

```
String s=null;
System.out.println(s.length());//NullPointerException
```

3) Scenario where NumberFormatException occurs

The wrong formatting of any value, may occur NumberFormatException. Suppose I have a string variable that have characters, converting this variable into digit will occur NumberFormatException.

```
String s="abc";
int i=Integer.parseInt(s);//NumberFormatException
```

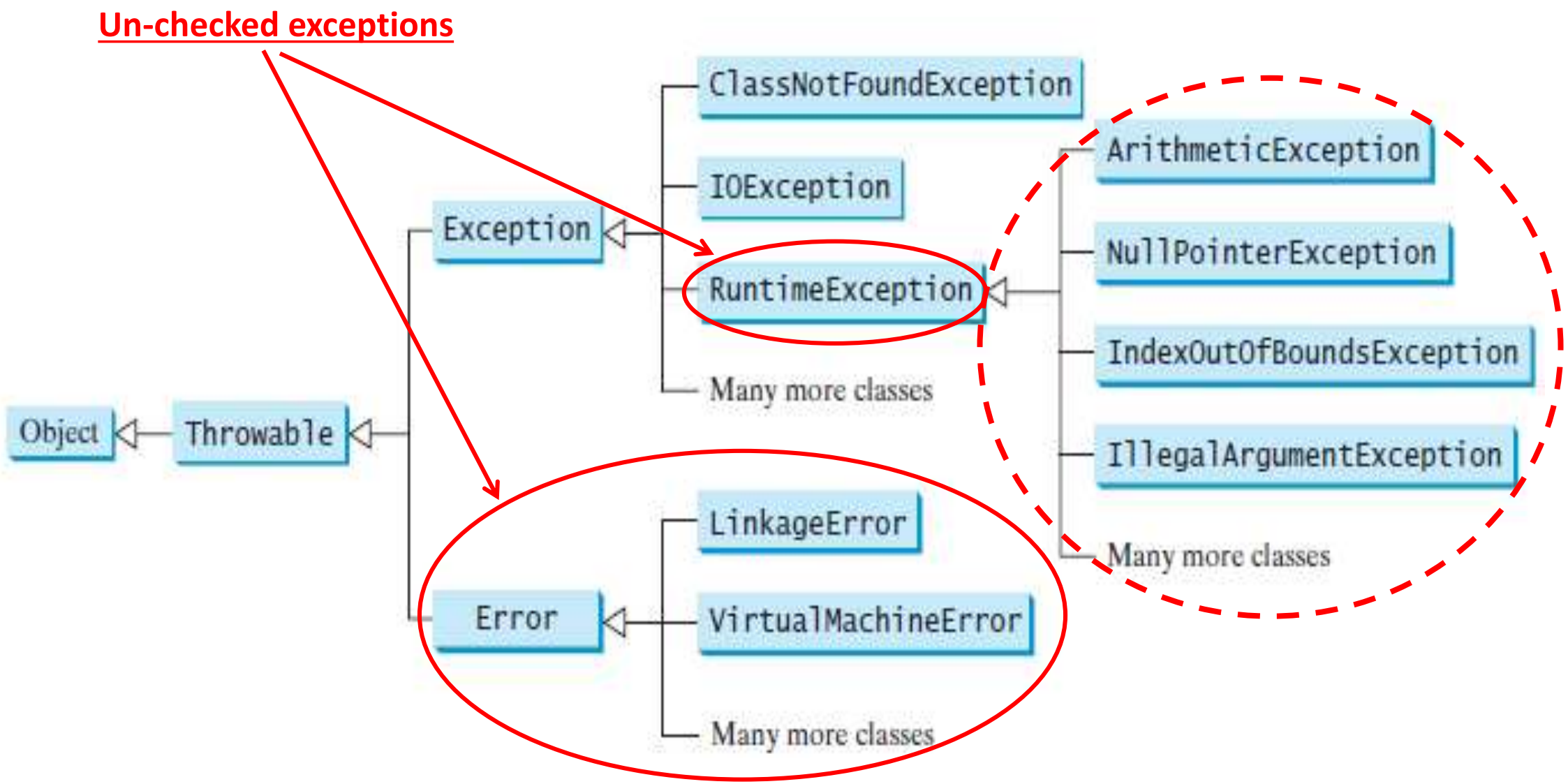
4) Scenario where ArrayIndexOutOfBoundsException occurs

If you are inserting any value in the wrong index, it would result ArrayIndexOutOfBoundsException as shown below:

```
int a[]=new int[5];
a[10]=50; //ArrayIndexOutOfBoundsException
```

Types of exceptions

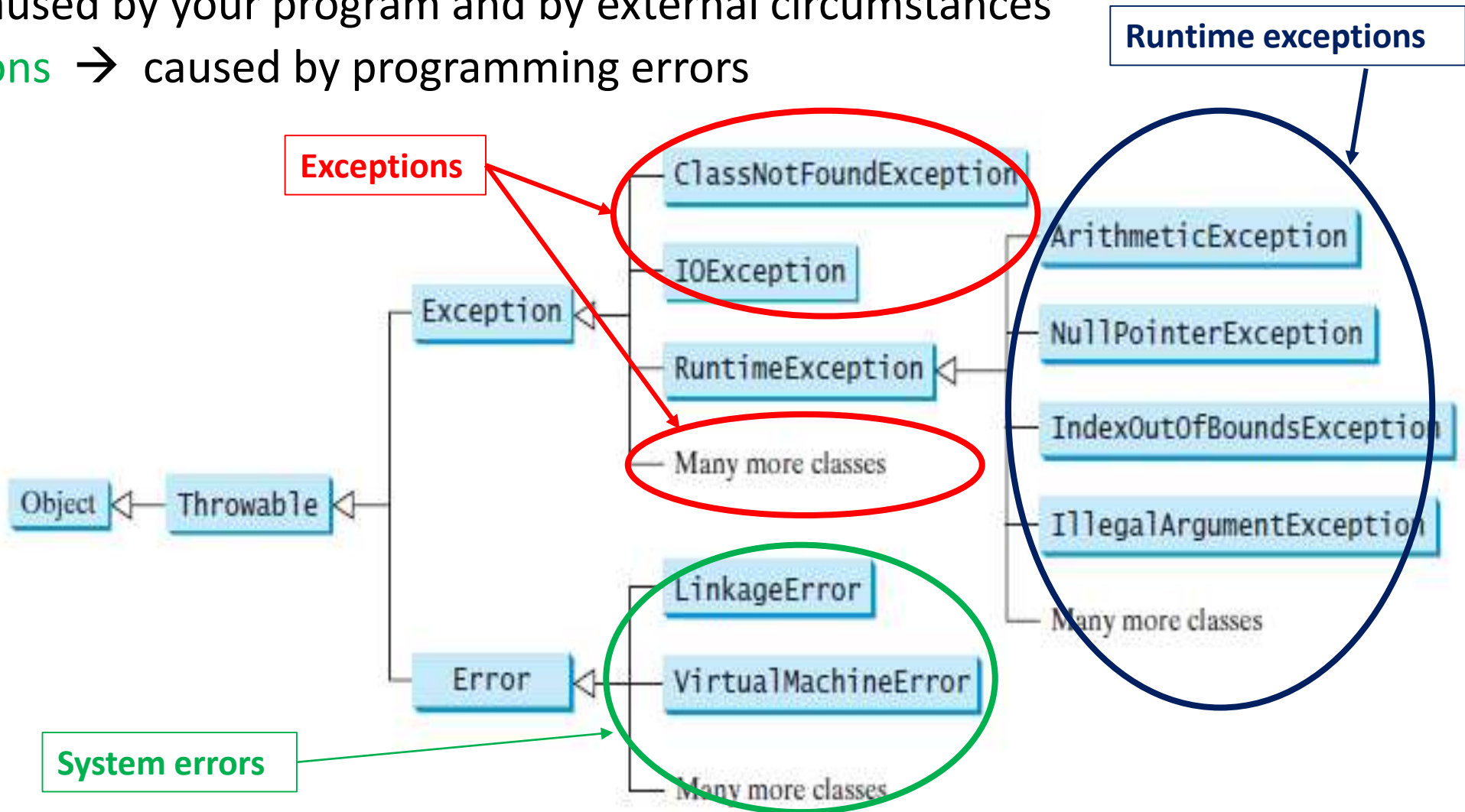
Hierarchy of exception classes



Classification

The exception classes can be classified into three major types:

1) system errors → caused by internal system errors
2) exceptions → caused by your program and by external circumstances
3) runtime exceptions → caused by programming errors



System errors

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

TABLE 14.1 Examples of Subclasses of `Error`

Class	*Reasons for Exception*
`LinkageError`	A class has some dependency on another class, but the latter class has changed incompatibly after the compilation of the former class.
`VirtualMachineError`	The JVM is broken or has run out of the resources it needs in order to continue operating.

Exceptions

- Exceptions are represented in the **Exception** class, which describes errors caused by your program and by external circumstances. These errors can be caught and handled by your program.

TABLE 14.2 Examples of Subclasses of `Exception`

Class	*Reasons for Exception*
`ClassNotFoundException`	Attempt to use a class that does not exist. This exception would occur, for example, if you tried to run a nonexistent class using the **java** command, or if your program were composed of, say, three class files, only two of which could be found.
`IOException`	Related to input/output operations, such as invalid input, reading past the end of a file, and opening a nonexistent file. Examples of subclasses of `IOException` are `InterruptedIOException`, `EOFException` (EOF is short for End of File), and `FileNotFoundException`.

Runtime exceptions

- Runtime exceptions are represented in the **RuntimeException** class, which describes programming errors.
- Example: bad casting, accessing an out-of-bounds array, and numeric errors.
- Runtime exceptions are generally thrown by the JVM.

TABLE 14.3 Examples of Subclasses of `RuntimeException`

Class	*Reasons for Exception*
`ArithmeticException`	Dividing an integer by zero. Note that floating-point arithmetic does not throw exceptions (see Appendix E, Special Floating-Point Values).
`NullPointerException`	Attempt to access an object through a `null` reference variable.
`IndexOutOfBoundsException`	Index to an array is out of range.
`IllegalArgumentException`	A method is passed an argument that is illegal or inappropriate.

Java Exception Handling Keywords

There are 5 keywords used in java exception handling.

1. try
2. catch
3. finally
4. throw
5. throws

Java try block

Java try block is used to enclose the code that might throw an exception. It must be used within the method.

Java try block must be followed by either catch or finally block. or with both.

Syntax of java try-catch

```
try{
//code that may throw exception
}catch(Exception_class_Name ref){}
```

Syntax of try-finally block

```
try{
//code that may throw exception
}finally{}
```

Java catch block

Java catch block is used to handle the Exception. It must be used after the try block only.

You can use multiple catch block with a single try.

It contains statements that are to be executed when the exception represented by the catch block is generated.

If program executes normally, then the statements of catch block will not executed.

If no catch block is found in program, exception is caught by JVM and program is terminated.

Example 1 (multi-catch block)

```
import java.util.*;
class Temp
{
      public static void main(String args[])
      {
            try {
                Scanner sc = new Scanner(System.in);
                System.out.print("Enter 1st no.: ");
                int a = sc.nextInt();
                System.out.print("Enter 2nd no.: ");
                int b = sc.nextInt();
                int c = a/b;
                System.out.println(args[0]);

            }
            catch(ArithmeticException e) {
                System.out.println("Can't divide by zero...");
            }
            catch(InputMismatchException e) {
                System.out.println("Inappropriate input...");
            }

            System.out.println("Hello....");
      }
}
```

Defining Generalized Exception Handler

- A generalized exception handler is one that can handle the exceptions of all types.
- If a class has a generalized as well as specific exception handler, then the generalized exception handler must be the last one.

```
class Test
{
    public static void main(String args[])
    {
        try
        {
            int a= Integer.parseInt(args[0]);
            int b= Integer.parseInt(args[1]);
            int c = a/b;
           System.out.println("Result is: " + c);
        }
        catch (Throwable e) {
            System.out.println(e);
        }
    }
}
```

Example 2 (multi-catch block)

If you have to perform different tasks at the occurrence of different Exceptions, use java multi catch block.

```
public class TestMultipleCatchBlock{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[5]=30/0;
   }
   catch(ArithmeticException e){System.out.println("task1 is completed");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
   catch(Exception e){System.out.println("common task completed");}

   System.out.println("rest of the code...");
  }
}
```

Key points

- At a time only one Exception is occurs and at a time only one catch block is executed.
- All catch blocks must be ordered from most specific to most general i.e. catch for **ArithmeticException** and **ArrayIndexOutOfBoundsException** must come before catch for **Exception** .

```
class TestMultipleCatchBlock1{
  public static void main(String args[]){
   try{
    int a[]=new int[5];
    a[5]=30/0;
   }
   catch(Exception e){System.out.println("common task completed");}
   catch(ArithmeticException e){System.out.println("task1 is completed");}
   catch(ArrayIndexOutOfBoundsException e){System.out.println("task 2 completed");}
   System.out.println("rest of the code...");
  }
}
```

The program will not compile since all the exceptions are caught in 1st catch block. **Hence, "Unreachable catch block" error**.

Nested Try's

```
class NestedTryDemo
{
    public static void main(String args[]){
        try {
            int a = Integer.parseInt(args[0]);
            try {
                int b = Integer.parseInt(args[1]);
                System.out.println(a/b);
            }
            catch (ArithmeticException e){
                System.out.println("Div by zero error!");
            }
        }
        catch (ArrayIndexOutOfBoundsException e) {
                System.out.println("Need 2 parameters!");
        }
    }
}
```

Java “**throw**” keyword

- It is used for explicit exception throwing.

throw <Exception-object>;

- ‘throw’ keyword can be used:
  - to throw user defined exception
  - to customize the message to be displayed by predefined exceptions
  - to re-throw a caught exception

Note: System-generated exceptions are automatically thrown by the Java run-time system.

Example (throwing exception)

```java
import java.util.Scanner;

public class Test
{
    public static void main(String args[])
    {
        try
        {
            Scanner sc = new Scanner(System.in);
            System.out.print("Enter an integer: ");
            int val = sc.nextInt();
            if(val<0)
            {
                IllegalArgumentException obj = new IllegalArgumentException();
                throw obj;
            }
        }
        catch(IllegalArgumentException e)
        {
            System.out.print("Value entered is not feasible...");
        }
    }
}
```

Or it can be written as:

```java
if(val<0)
{
    throw new IllegalArgumentException();
}
```

Example (throwing exception)

```java
import java.util.Scanner;
public class Test
{
    public static void main(String args[])
    {
        try
        {
            fun();
        }
        catch(IllegalArgumentException e)
        {
            System.out.print("Value entered is not feasible...");
        }
    }
    static void fun()
    {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter an integer: ");
        int val = sc.nextInt();
        if(val<0)
        {
            throw new IllegalArgumentException();
        }
    }
}
```

Exception not handled by fun() method which is thrown in it.

So, it has to be handled in the calling method (i.e method which invoked fun() method)

Exercise 1

Write an application named **BadScript** in which you declare an array of 5 names. Write a try block in which you prompt the user for an integer and display the name in the requested position. Create a catch block that catches the potential **ArrayIndexOutOfBoundsException** thrown when the user enters a number that is out of range. The catch block should also display an appropriate error message.

Exercise 2

Write a program to take String input from user which contains a number of double data-type. Extract the no. from the String to calculate its square and display it. If the input String doesn’t contain a no. then exception named **NumberFormatException** is generated at run-time, which should be handled with an appropriate catch block. The catch block should display the no. which caused the exception using appropriate error message.

Keep on taking input from the user until he enters a value.

Example (re-throwing exception)

```java
import java.util.Scanner;
public class Test
{
    public static void main(String args[])
    {
        try {
            fun();
        }
        catch(IllegalArgumentException e) {
            System.out.print("Value entered is not feasible...");
        }
    }
    static void fun()
    {
        try {
            Scanner sc = new Scanner(System.in);
            System.out.print("Enter an integer: ");
            int val = sc.nextInt();
            if(val<0) {
                throw new IllegalArgumentException();
            }
        }
        catch(IllegalArgumentException e) {
            throw e;
        }
    }
}
```

Re-throwing exception

Example (without exception handling)

LISTING 14.3 QuotientWithMethod.java

```java
import java.util.Scanner;

public class QuotientWithMethod {
  public static int quotient(int number1, int number2) {
    if (number2 == 0) {
      System.out.println("Divisor cannot be zero");
      System.exit(1);
    }

    return number1 / number2;
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    int result = quotient(number1, number2);
    System.out.println(number1 + " / " + number2 + " is "
      + result);
  }
}
```

Consider a scenario where this method is to be implemented by any other java developer

Method quitting program to avoid runtime exception

Invoking method to perform calculation

Example (with exception handling)

```java
import java.util.Scanner;

public class QuotientWithException {
  public static int quotient(int number1, int number2) {
    if (number2 == 0)
      throw new ArithmeticException("Divisor cannot be zero");

    return number1 / number2;
  }

  public static void main(String[] args) {
    Scanner input = new Scanner(System.in);

    // Prompt the user to enter two integers
    System.out.print("Enter two integers: ");
    int number1 = input.nextInt();
    int number2 = input.nextInt();

    try {
      int result = quotient(number1, number2);
      System.out.println(number1 + " / " + number2 + " is "
        + result);
    }
    catch (ArithmeticException ex) {
      System.out.println("Exception: an integer " +
        "cannot be divided by zero ");
    }

    System.out.println("Execution continues ...");
  }
}
```

If an ArithmeticException occurs

Method throws exception object instead of quitting program

Invoking method to perform calculation

Chained 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

Chained exceptions

- Sometimes, you may need to throw a new exception (with additional information) along with the original exception. This is called **chained exceptions**.

```java
public class ChainedExceptionDemo {
  public static void main(String[] args) {
    try {
      method1();
    }
    catch (Exception ex) {
      ex.printStackTrace();
    }
  }

  public static void method1() throws Exception {
    try {
      method2();
    }
    catch (Exception ex) {
      throw new Exception("New info from method1", ex);
    }
  }

  public static void method2() throws Exception {
    throw new Exception("New info from method2");
  }
}
```

Java "**throws**" keyword

The **Java throws keyword** is used to declare an exception. It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

Exception Handling is mainly used to handle the checked exceptions. If there occurs any unchecked exception such as NullPointerException, it is programmers fault that he is not performing check up before the code being used.

- Syntax:

```
type method-name(parameter-list) throws exception-list
{
        // body of method
}
```

- This is necessary for all exceptions, except those of type Error or Runtime Exception, or any of their subclasses.

- All other exceptions that a method can throw must be declared in the throws clause. If they are not, a compile-time error will result.

- If method throws checked exception, then either provide try-catch block in that method or write “**throws**” keyword at the method header.

- If “**throws**” keyword is used at the method header, then the calling method must handle the exception which could be thrown from the called method.

Example

```
import java.io.IOException;
import java.util.Scanner;
class Test
{
    public static void main(String args[])
    {
        try {
            check();
        }
        catch(IOException e) {
            System.out.println("You are not eligible voter...");
        }
    }

    static void check() throws IOException
    {
        Scanner sc = new Scanner(System.in);
        System.out.print("Enter the age: ");
        int age = sc.nextInt();
        if(age < 18) {
            IOException obj = new IOException();
            throw obj;
        }
        System.out.println("You are eligible voter...");
    }
}
```

Java "**finally**" keyword

- **Java finally block is a block that is used *to execute important code* such as closing connection, stream etc.**
- "**finally**" creates a block of code that will be executed after a try/catch block has completed and before the code following the try/catch block.
- The "**finally**" block will execute whether or not an exception is thrown.
- If an exception is thrown, the "**finally**" block will execute even if no catch statement matches the exception.
- If a "**finally**" block is associated with a try, the "**finally**" block will be executed upon conclusion of the try.
- The "**finally**" clause is optional. However, each try statement requires at least one catch or a "**finally**" clause.
- **Note:** **1) For each try block there can be zero or more catch blocks, but only one finally block.**

  2) The finally block will not be executed if program exits(either by calling System.exit() or by causing a fatal error that causes the process to abort).

Example

```
public class Computer
{
    public static void main(String []args)
    {
        try
        {
            int c = 5/0;
            System.out.println("Statement inside try-catch");
        }
        catch(ArithmeticException e)
        {
            System.out.println("Exception occured");
        }
        finally
        {
            System.out.println("Statement inside finally");
        }
        System.out.println("Statement outside try-catch");
    }
}
```

Creating Custom/user-defined Exception

Defining Custom Exceptions

- We can create our own Exception sub-classes by inheriting Exception class.
- The Exception class does not define any methods of its own.
- It inherits those methods provided by Throwable.
- Thus, all exceptions, including those that we create, have the methods defined by Throwable available to them.

Constructors for creating Exception:

- Exception( )
- Exception(String msg)

- A custom exception class is represented by a subclass of *Exception / Throwable.*

- It contains the above mentioned constructor to initialize custom exception object.

Example

```
public class ExceptionTest
{
    public static void main(String []args){
        int i=Integer.parseInt(args[0]);
        int j=Integer.parseInt(args[1]);
        ExceptionTest t=new ExceptionTest();
        try{
            t.show(i);      t.show(j);
        }
        catch(Throwable e) {
            System.out.println("catched exception is "+e);
        }
    }
    public void show(int i) throws Myexception {
        if(i>100)
            throw new Myexception(i);
        else
            System.out.println(+i+" is less then 100 it is ok");
    }
}
```

```
class Myexception extends Throwable
{
    public Myexception(int i)
    {
        System.out.println("you  have entered ." +i +"  : It
        exceeding the limit");
    }
}
```

Here object of custom class is created and thrown

Custom Exception (Example 1)

```
public class Computer
{
    public static void main(String []args)
    {
        try
        {
            Computer c = new Computer();
            c.fun();
        }
        catch(InvalidRange e)
        {
            System.out.println("This is an exception...");
        }
    }
    void fun()
    {
        Scanner sc = new Scanner(System.in);
        int a = sc.nextInt();
        if(a<0)
            throw new InvalidRange(a);
    }
}
```

Custom/ user-defined exception

```
class InvalidRange extends RuntimeException
{
    InvalidRange(int i)
    {
        System.out.println("Entered value is: " + i);
    }
}
```

Custom Exception (Example 2)

```java
public class ExceptionHandling
{
    public static void main(String args[])
    {
        Scanner input  = new Scanner(System.in);
        try
        {
            System.out.print("Enter radius : ");
            double radius = input.nextDouble();
            if(radius<0)
                throw new MyException(radius);
            System.out.println("Area is: " + 3.14 * radius * radius);
        }
        catch(MyException e)
        {
            System.out.println("Exception occured...");
            System.out.println(e);
        }
    }
}
```

Custom/ user-defined exception

```java
class MyException extends Exception
{
    double r;
    MyException(double radius)
    {
        r = radius;
    }
    public String toString()
    {
        return ("Invalid no.: " + r);
    }
}
```

Assertion

Assertion

Assertion is a statement in java. It can be used to test your assumptions about the program.

While executing assertion, it is believed to be true. If it fails, JVM will throw an error named AssertionError. It is mainly used for testing purpose. They are not usually used for released code.

Advantage of Assertion:

It provides an effective way to detect and correct programming errors.

Syntax of using Assertion:

There are two ways to use assertion. First way is:

```
assert expression;
```

and second way is:

```
assert expression1 : expression2;
```

Example

```
import java.util.Scanner;


class AssertionExample{
 public static void main( String args[] ){

  Scanner scanner = new Scanner( System.in );
  System.out.print("Enter ur age ");

  int value = scanner.nextInt();
  assert value>=18:" Not valid";

  System.out.println("value is "+value);
 }
}
```

If you use assertion, It will not run simply because assertion is disabled by default. To enable the assertion, **-ea** or **-enableassertions** switch of java must be used.

Compile it by: **javac AssertionExample.java**

Run it by: **java -ea AssertionExample**

```
Output: Enter ur age 11
        Exception in thread "main" java.lang.AssertionError: Not valid
```

Example

```java
import java.util.*;
public class Test
{
    public static void main(String args[])
    {
        int length;
        final double rate = 5.4;

        Scanner sc = new Scanner(System.in);
        System.out.println("Enter length: ");
        length = sc.nextInt();

        double result = length * 12 * rate;

        assert result<650;      //error will be thrown if this statement goes wrong

        double calculate = result + 3.14 * 2;
        System.out.println("Final calculation is: " + calculate);
    }
}
```

One important point to understand about assertions is that you must not rely on them to perform any action actually required by the program. The reason is that normally, released code will be run with assertions disabled.

```
public class Test
{
    static int val = 3;

    static int getnum()
    {
        return val--;
    }

    public static void main(String args[])
    {
        int n = 0;
        for(int i=0; i < 10; i++) {
            assert (n = getnum()) > 0;  // This is not a good idea!
            System.out.println("n is " + n);
        }
    }
}
```

This statement will not execute if assertion is not enabled using "-ea" switch

Assertions are a good addition to Java because they streamline the type of error checking that is common during development. For example, prior to **assert**, if you wanted to verify that **n** was positive in the preceding program, you had to use a sequence of code similar to this:

```
if(n < 0) {
   System.out.println("n is negative!");
   return; // or throw an exception
}
```

With **assert**, you need only one line of code. Furthermore, you don't have to remove the **assert** statements from your released code.

Key points

Below are two rules to note when overriding methods related to exception-handling.

- **Rule#1 :** If the super-class overridden method does not throws an exception, subclass overriding method can only throws the unchecked exception, throwing checked exception will lead to compile-time error.

```
/* Java program to demonstrate overriding when
  superclass method does not declare an exception
*/

class Parent
{
    void m1() { System.out.println("From parent m1()");}

    void m2() { System.out.println("From parent  m2()"); }
}

class Child extends Parent
{
    @Override
    // no issue while throwing unchecked exception
    void m1() throws ArithmeticException
    { System.out.println("From child m1()");}

    @Override
    // compile-time error
    // issue while throwin checked exception
    void m2() throws Exception{ System.out.println("From child m2");}

}
```

```java
class A {
   public void foo() throws IOException {..}
}

class B extends A {
   @Override
   public void foo() throws SocketException {..} // allowed

   @Override
   public void foo() throws SQLException {..} // NOT allowed
}
```

`SocketException extends IOException`, but `SQLException` does not.

This is because of polymorphism:

```java
A a = new B();
try {
    a.foo();
} catch (IOException ex) {
    // forced to catch this by the compiler
}
```

If `B` had decided to throw `SQLException`, then the compiler could not force you to catch it, because you are referring to the instance of `B` by its superclass - `A`. On the other hand, any subclass of `IOException` will be handled by clauses (catch or throws) that handle `IOException`

- **Rule#2 :** If the super-class overridden method does throws an exception, subclass overriding method can only throw same, subclass exception. Throwing parent exception in Exception hierarchy will lead to compile time error.Also there is no issue if subclass overridden method is not throwing any exception.

```java
/* Java program to demonstrate overriding when
  superclass method does declare an exception
*/

class Parent
{
    void m1() throws RuntimeException
    { System.out.println("From parent m1()");}

}

class Child1 extends Parent
{
    @Override
    // no issue while throwing same exception
    void m1() throws RuntimeException
    { System.out.println("From child1 m1()");}

}
```

```java
class Child2 extends Parent
{
    @Override
    // no issue while throwing subclass exception
    void m1() throws ArithmeticException
    { System.out.println("From child2 m1()");}

}
class Child3 extends Parent
{
    @Override
    // no issue while not throwing any exception
    void m1()
    { System.out.println("From child3 m1()");}

}
class Child4 extends Parent
{
    @Override
    // compile-time error
    // issue while throwing parent exception
    void m1() throws Exception
    { System.out.println("From child4 m1()");}

}
```