Exceptions

- Exception is an abnormal condition that arises when executing a program.
- In the languages that do not support exception handling, errors must be checked and handled manually, usually through the use of error codes.
- In contrast, Java:

1) provides syntactic mechanisms to signal, detect and handle errors
2) ensures a clean separation between the code executed in the absence of errors and the code to handle various kinds of errors
3) brings run-time error management into object-oriented programming

Uncaught Exception

- What happens when exceptions are not handled?
- ```
  class Exc0 {
      public static void main(String args[]) {
      int d = 0;
      int a = 42 / d;
      }
  }
  ```
- When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and throws this object.
- This will cause the execution of Exc0 to stop – once an exception has been thrown it must be caught by an exception handler and dealt with.
  ```

Default Exception Handler

- As we have not provided any exception handler, the exception is caught by the default handler provided by the Java run-time system.
- This default handler:
- 1) displays a string describing the exception,
- 2) prints the stack trace from the point where the exception occurred
- 3) terminates the program
- java.lang.ArithmeticException: / by zero
- at Exc0.main(Exc0.java:4)
- Any exception not caught by the user program is ultimately processed by the default handler.

Own Exception Handling

- Default exception handling is basically useful for debugging.
- Normally, we want to handle exceptions ourselves because:

1) if we detected the error, we can try to fix it

2) we prevent the program from automatically terminating

- Exception handling is done through the try and catch block.

Try and Catch

- Try and catch:
- 1) try surrounds any code we want to monitor for exceptions
- 2) catch specifies which exception we want to handle and how.
- When an exception is thrown in the try block:
- try {

```
d = 0;
a = 42 / d;
System.out.println("This will not be printed.");
```

- }

Try and Catch

- control moves immediately to the catch block:
- catch (ArithmeticException e) {

  System.out.println("Division by zero.");
- }
- The exception is handled and the execution resumes.
- The scope of catch is restricted to the immediately preceding try statement -
- it cannot catch exceptions thrown by another try statements.

Exception Handling

- An **exception** is an object that describes an exceptional condition (error) that has occurred when executing a program.
- Exception handling involves the following:

1) when an error occurs, an object (exception) representing this error is created and thrown in the method that caused it.
2) that method may choose to handle the exception itself or pass it on.
3) either way, at some point, the exception is caught and processed

Exception Sources

- Exceptions can be:

1. generated by the Java run-time system

- Fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment.

2. manually generated by programmer's code

- Such exceptions are typically used to report some error conditions to the caller of a method.

Most common types of exceptions

1. NullPointerException: generated when the reference used to invoke a method has value null, or when we try to access an instance variable through a null reference.

- Some methods throw explicitly this type of exception when they are passed a parameter that is null.

2. ArrayIndexOutOfBoundsException: generated when we access an element of an array using an index that is less than zero, or bigger than the length of the array minus one.
3. IOException: generated by methods that access input/output devices when an error situation occurs.

Most common types of exceptions

4. FileNotFoundException: generated when we try to open a non-existent file.
5. NumberFormatException: generated by methods that perform conversions from strings to numbers.

- For example, Integer.parseInt generates an exception of this type if the string passed as parameter does not contain a number.

Exception Constructs

- Five constructs are used in exception handling:

1) try – a block surrounding program statements to monitor for exceptions
2) catch – together with try, catches specific kinds of exceptions and handles them in some way
3) finally – specifies any code that absolutely must be executed whether or not an exception occurs
4) throw – used to throw a specific exception from the program
5) throws – specifies which exceptions a given method can throw

Exception-Handling Block

- General form:
- try {
- // block of code to monitor for errors
- }
- catch (*ExceptionType1 exOb) {*
- // exception handler for *ExceptionType1*
- }
- catch (*ExceptionType2 exOb) {*
- // exception handler for *ExceptionType2*
- }
- finally {
- // block of code to be executed before try block ends
- }

Exception-Handling Block

- where:

1) try { … } is the block of code to monitor for exceptions
2) catch(Exception ex) { … } is exception handler for the exception Exception
3) finally { … } is the block of code to execute before the try block ends

Exception Hierarchy

- All exceptions are sub-classes of the build-in class Throwable.
- Throwable contains two immediate sub-classes:

1) Exception – exceptional conditions that programs should catch

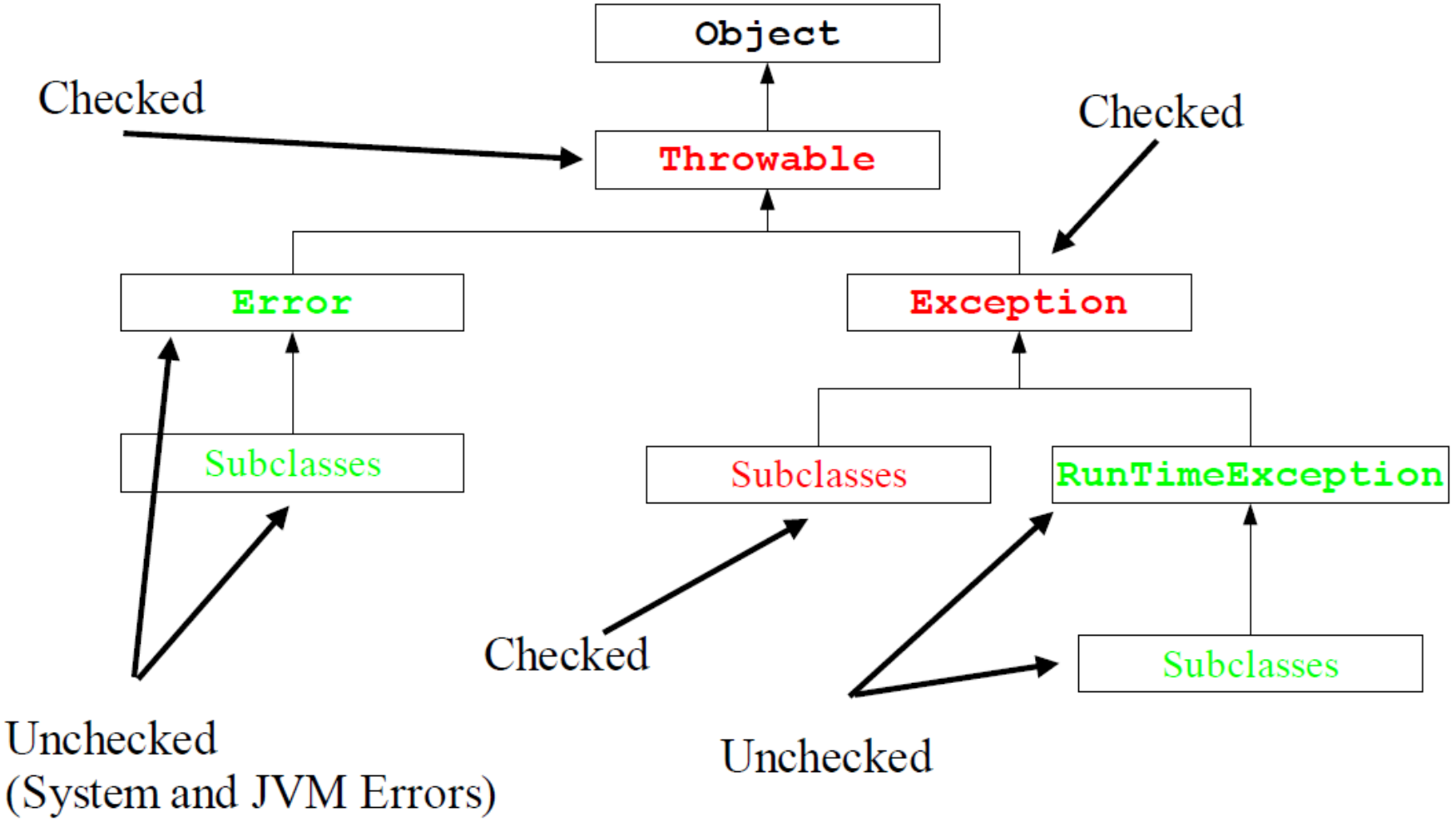
- The class includes:

  a) RuntimeException – defined automatically for user programs to include: division by zero, invalid array indexing, etc.

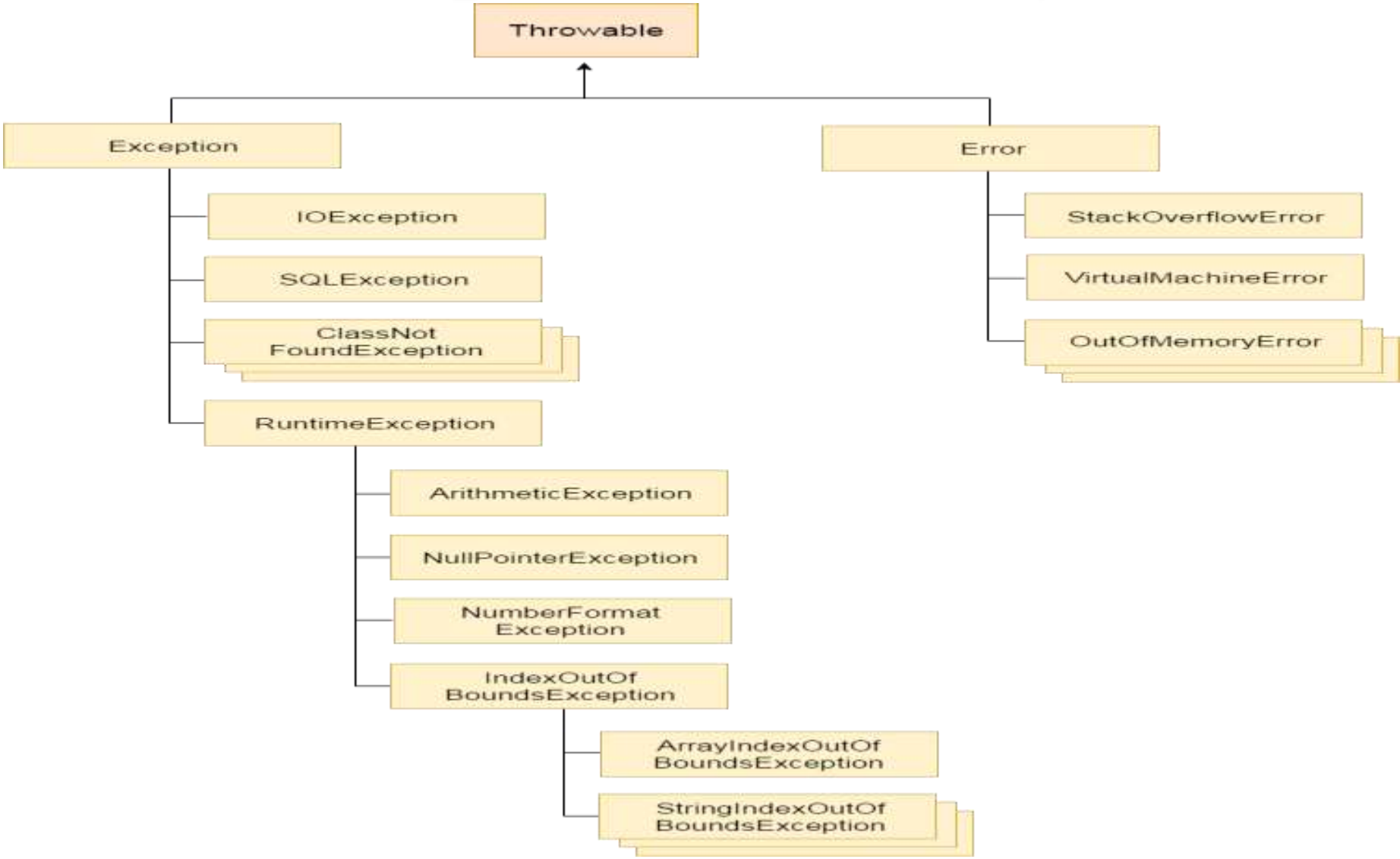
  b) use-defined exception classes

- 2) Error – exceptions used by Java to indicate errors with the runtime environment; user programs are not supposed to catch them

Java's Exception Class Hierarchy



Exception Hierarchy



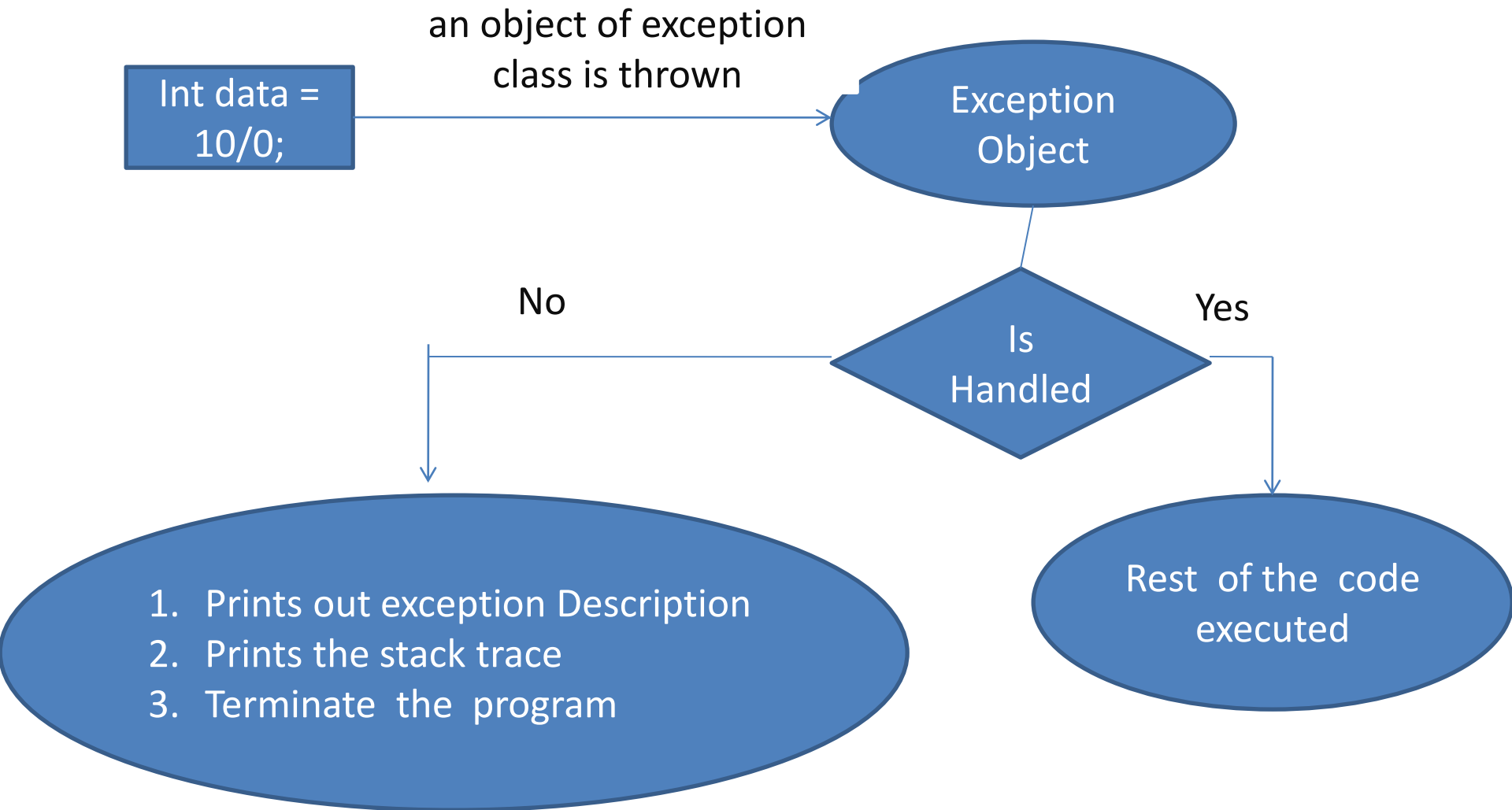
Difference between checked and unchecked exceptions

- Checked Exception
- The classes that extend Throwable class except RuntimeException and Error are known as checked exceptions e.g.IOException, SQLException etc. Checked exceptions are checked at compile-time.
- Unchecked Exception
- The classes that extend RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. Unchecked exceptions are not checked at compile-time rather they are checked at runtime.
- Error
- Error is irrecoverable e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

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:
  1. Prints out exception description.
  2. Prints the stack trace (Hierarchy of methods where the exception occurred).
  3. 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.

Internal working of java try-catch block



Try and Catch

- Resumption occurs with the next statement after the try/catch block:
- try { … }
- catch (ArithmeticException e) { … }
- System.out.println("After catch statement.");
- Not with the next statement after a = 42/d; which caused the exception!
- a = 42 / d;
- System.out.println("This will not be printed.");

Exception Display

- All exception classes inherit from the Throwable class.
- Throwable overrides toString() to describe the exception textually:
- try { … }

  catch (ArithmeticException e) {

  System.out.println("Exception: " + e);

  }
- The following text will be displayed:
- Exception: java.lang.ArithmeticException: / by zero

Multiple Catch Clauses

- When more than one exception can be raised by a single piece of code,
- several catch clauses can be used with one try block:

1) each catch catches a different kind of exception
2) when an exception is thrown, the first one whose type matches that of the exception is executed
3) after one catch executes, the other are bypassed and the execution continues after the try/catch block

Example: Multiple Catch

- Two different exception types are possible in the following code: division by zero and array index out of bound:
- class MultiCatch {

```
public static void main(String args[]) {
try {
        int a = args.length;
        System.out.println("a = " + a);
        int b = 42 / a;
        int c[] = { 1 };
        c[42] = 99;
}
```

Example: Multiple Catch

- Both exceptions can be caught by the following catch clauses:
- catch(ArithmeticException e) {

```
                System.out.println("Divide by 0: " + e);
} catch(ArrayIndexOutOfBoundsException e) {
    System.out.println("Array index oob: " + e);
}
    System.out.println("After try/catch blocks.");
}
```

- }

Order of Multiple Catch Clauses

- Order is important:

1) catch clauses are inspected top-down

2) a clause using a super-class will catch all sub-class exceptions

- Therefore, specific exceptions should appear before more general ones.
- In particular, exception sub-classes must appear before super-classes.

Example: Multiple Catch

- A try block with two catch clauses:
- class SuperSubCatch {

```
public static void main(String args[]) {
try {
        int a = 0;
        int b = 42 / a;
```

- This exception is more general but occurs first:

```
} catch(Exception e) {
System.out.println("Generic Exception catch.");
```

- }

Example: Multiple Catch

- This exception is more specific but occurs last:
- catch(ArithmeticException e) {
  System.out.println("This is never reached.");
  }
  }
- }
- The second clause will never get executed. A compile-time error (unreachable code) will be raised.

Nested try Statements

- The try statements can be nested:

1) if an inner try does not catch a particular exception
2) the exception is inspected by the outer try block
3) this continues until:
    a) one of the catch statements succeeds or
    b) all the nested try statements are exhausted
4) in the latter case, the Java run-time system will handle the exception

Example: Nested try

- An example with two nested try statements:
- class NestTry {

```
    public static void main(String args[]) {
```

- Outer try statement:

```
    try {
        int a = args.length;
```

- Division by zero when no command-line argument is present:

```
        int b = 42 / a;
        System.out.println("a = " + a);
```

Example: Nested try

- Inner try statement:

```
try {
```

- Division by zero when one command-line argument is present:

```
if (a==1) a = a/(a-a);
```

- Array index out of bound when two command-line arguments are present:

```
if (a==2) {
int c[] = { 1 };
c[42] = 99;
}
```

Example: Nested try

- Catch statement for the inner try statement, catches the array index out of bound exception:
- } catch(ArrayIndexOutOfBoundsException e) {
  System.out.println(e);
- }
- Catch statement for the outer try statement, catches both division-by-zero exceptions for the inner and outer try statements:
- } catch(ArithmeticException e) {
  System.out.println("Divide by 0: " + e);
- }
- }

Throwing Exceptions

- So far, we were only catching the exceptions thrown by the Java system.
- In fact, a user program may throw an exception explicitly:

```
throw ThrowableInstance;
```

- ThrowableInstance must be an object of type Throwable or its subclass.

throw Follow-up

- Once an exception is thrown by:

  throw ThrowableInstance;

1) the flow of control stops immediately
2) the nearest enclosing try statement is inspected if it has a catch statement that matches the type of exception:

1) if one exists, control is transferred to that statement
2) otherwise, the next enclosing try statement is examined
3) if no enclosing try statement has a corresponding catch clause, the default exception handler halts the program and prints the stack

Creating Exceptions

- Two ways to obtain a Throwable instance:
- 1) creating one with the new operator
- All Java built-in exceptions have at least two constructors: one without parameters and another with one String parameter:
- throw new NullPointerException("demo");
- 2) using a parameter of the catch clause
- try { … } catch(Throwable e) { … e … }

Creating Exceptions

```
public class TestThrow1{
   static void validate(int age){
     if(age<18)
      throw new ArithmeticException("not valid");
     else
      System.out.println("welcome to vote");
   }
   public static void main(String args[]){
     validate(13);
     System.out.println("rest of the code...");
  }
}
```

Example: throw

- class ThrowDemo {
- The method demoproc throws a NullPointerException exception which is immediately caught in the try block and re-thrown:
- static void demoproc() {

```
        try {
                throw new NullPointerException("demo");
        } catch(NullPointerException e) {
                System.out.println("Caught inside demoproc.");
                throw e;
```

- }
- }

Example: throw

- The main method calls demoproc within the try block which catches and handles the NullPointerException exception:
- public static void main(String args[]) {

```
        try {
                demoproc();
        } catch(NullPointerException e) {
                System.out.println("Recaught: " + e);
        }
        }       //OutPut : Caught inside demoproc.
} Recaught: java.lang.NullPointerException: demo
```

throws Declaration

- 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.

throws Declaration

- If a method is capable of causing an exception that it does not handle, it must specify this behavior by the throws clause in its declaration:
- type name(parameter-list) throws exception-list {
- …
- }
- where exception-list is a comma-separated list of all types of
- exceptions that a method might throw.
- All exceptions must be listed except Error and RuntimeException or any of their subclasses, otherwise a compile-time error occurs.

throws

1. throws keyword is required only for checked exception and usage of throws keyword for unchecked exception is meaningless.
2. throws keyword is required only to convince compiler and usage of throws keyword does not prevent abnormal termination of program.
3. By the help of throws keyword we can provide information to the caller of the method about the exception.

Example: throws

- Corrected program: throwOne lists exception, main catches it:
- class ThrowsDemo {

```
        static void throwOne() throws IllegalAccessException {
                System.out.println("Inside throwOne.");
                throw new IllegalAccessException("demo");
        }
        public static void main(String args[]) {
                try {
                        throwOne();
                } catch (IllegalAccessException e) {
                        System.out.println("Caught " + e);
```

- } inside throwOne

```
}       }                       //caught java.lang.IllegalAccessException:
```

demo

Throw Vs throws

	Throw	**throws**
1.	Throw is used to explicitly throw an exception	Throws is used to declare an exception
2.	Checked exception can not be propagated without throws	Checked exception can be propagated with exception
3.	Throw followed by an instance	Throws followed by a class
4.	Throw used within the method	Throws is used with the method signature
5.	We cannot throw multiple exception	We can declare multiple exception eg. public void method() throws IOExeptions,SQlException

Motivating finally

- When an exception is thrown:

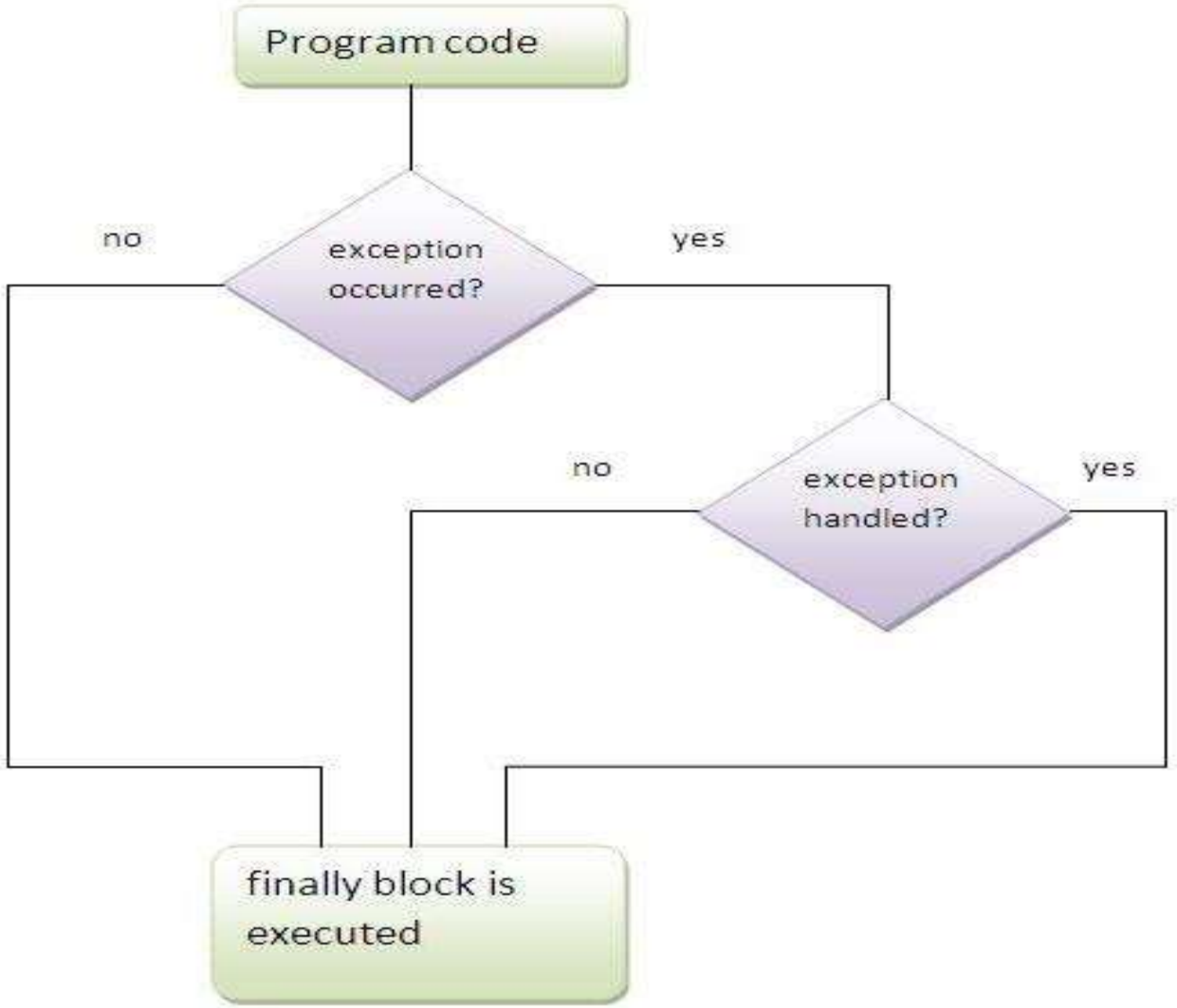
  1) the execution of a method is changed

  2) the method may even return prematurely.
- This may be a problem is many situations.
- For instance, if a method opens a file on entry and closes on exit; exception handling should not bypass the proper closure of the file.
- The finally block is used to address this problem.

finally Clause

- The try/catch statement requires at least one catch or finally clause, although both are optional:
- try { … }
- catch(Exception1 ex1) { … } …
- finally { … }
- Executed after try/catch whether of not the exception is thrown.
- Any time a method is to return to a caller from inside the try/catch block via:
- 1) uncaught exception or
- 2) explicit return
- the finally clause is executed just before the method returns.

finally Clause



Example: finally

- Three methods to exit in various ways.
- class FinallyDemo {
- procA prematurely breaks out of the try by throwing an exception, the finally clause is executed on the way out:

```
static void procA() {
        try {
                System.out.println("inside procA");
                throw new RuntimeException("demo");
        } finally {
                System.out.println("procA's finally");
        }
}
```

Example: finally

- procB's try statement is exited via a return statement, the finally
- clause is executed before procB returns:

```
static void procB() {
try {
        System.out.println("inside procB");
        return;
} finally {
        System.out.println("procB's finally");
}
```

- }

Example: finally

- In procC, the try statement executes normally without error, however the finally clause is still executed:
- static void procC() {

```
    try {
        System.out.println("inside procC");
    } finally {
        System.out.println("procC's finally");
    }
```

- }

Example: finally

- Demonstration of the three methods:
- public static void main(String args[]) {

```
        try {
                procA();
        } catch (Exception e) {
                System.out.println("Exception caught");
        }
        procB();
        procC();
        }
```

- }

Example: finally

- inside procA
- procA's finally
- Exception caught
- inside procB
- procB's finally
- inside procC
- procC's finally

Own Exception

- If you are creating your own Exception that is known as custom exception or user-defined exception.
- Java custom exceptions are used to customize the exception according to user need.
- By the help of custom exception, you can have your own exception and message.
- The Exception class does not define any methods of its own.
- It does, of course, inherit those methods provided by Throwable.
- Thus, all exceptions, including those that you create, have the methods defined by Throwable available to them.

Own Exception

```
class MyException extends Exception {
        private int detail;
        MyException(int a) {
                detail = a;
        }
        public String toString() {
                return "MyException[" + detail + "]";
        }
}
class ExceptionDemo {
        static void compute(int a) throws MyException {
        System.out.println("Called compute(" + a + ")");
```

Own Exception

```
        if(a > 10)
                throw new MyException(a);
                System.out.println("Normal exit");
        }
        public static void main(String args[]) {
        try {
                compute(1);
                compute(20);
        } catch (MyException e) {
        System.out.println("Caught " + e);
        }
    }
}
```

Own Exception

- OutPut
- Called compute(1)
- Normal exit
- Called compute(20)
- Caught MyException[20]