Chapter Four
Exception Handling

Exceptions Overview

- An exception indicates a problem that occurs while a program executes.
- The name “exception” suggests that the problem occurs infrequently—if the “rule” is that a statement normally executes correctly, then the problem represents the “exception to the rule.”
- Exception handling helps you create fault-tolerant programs that can resolve (or handle) exceptions.
  - Fault-tolerance allows a program to continue executing as if no problems were encountered.

- More severe problems might prevent a program from continuing normal execution, instead requiring the program to notify the user of the problem, then terminate.
- When the JVM or a method detects a problem, such as an invalid array index or an invalid method argument, it throws an exception—that is, an exception occurs.
- Methods in your own classes can also throw exceptions

The try Statement

- To handle an exception, place any code that might throw an exception in a try statement.
- The try block contains the code that might throw an exception, and the catch block contains the code that handles the exception if one occurs.
- You can have many catch blocks to handle different types of exceptions that might be thrown in the corresponding try block.
- The braces that delimit the bodies of the try and catch blocks are required

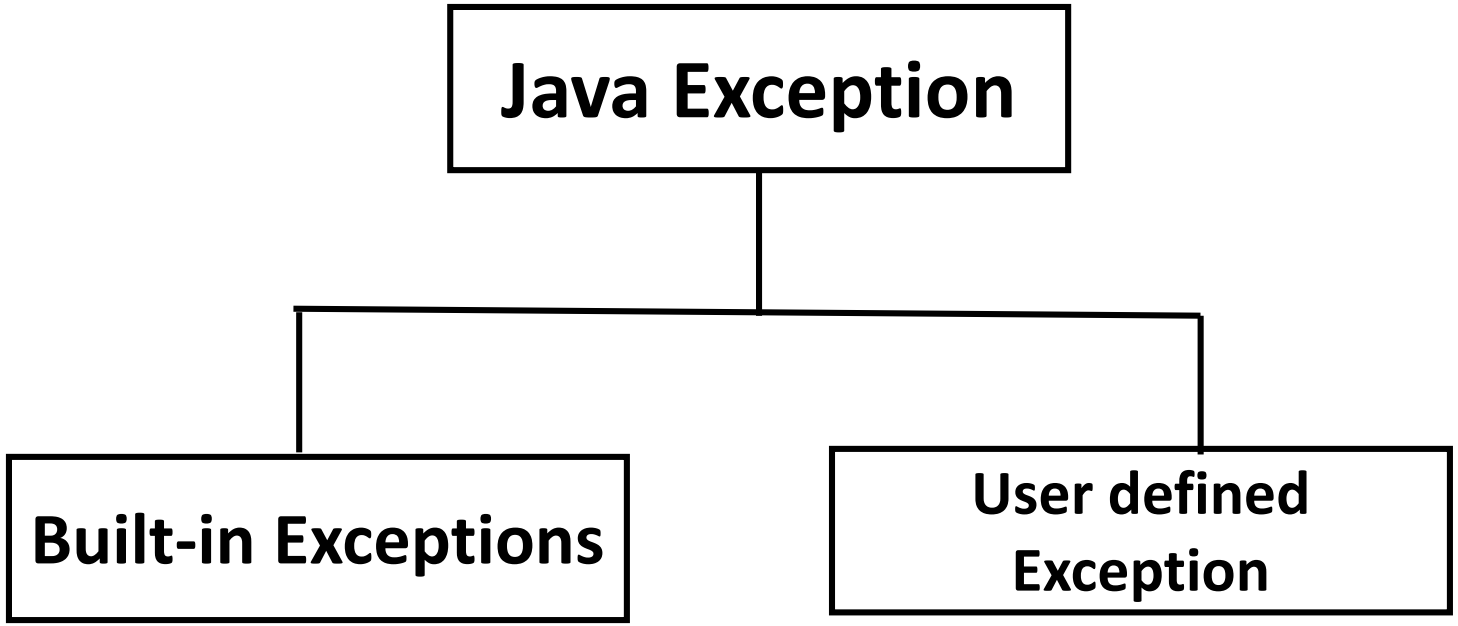
Example

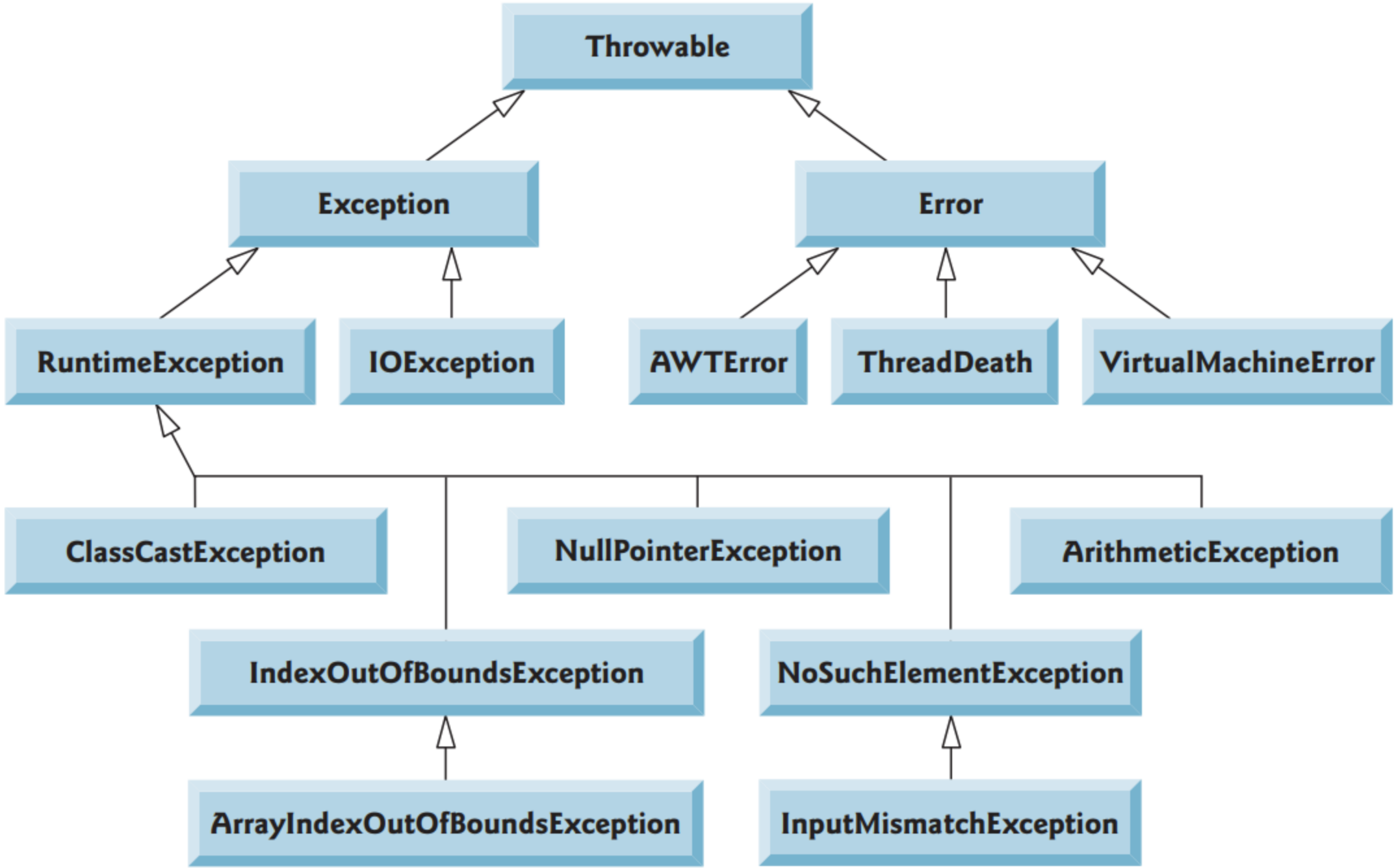
```
package exception;
public class Exception {
   public static void main(String []args){
      int [] responses={1, 2, 5, 4, 3, 5, 2, 1, 3, 3, 1, 4, 3, 3, 3,
      2, 3, 3, 2, 14 };
      int []frequency=new int[7];
      for (int answer=0;answer<responses.length;answer++){
         try{
            ++frequency[responses[answer]];
         }
         catch(ArrayIndexOutOfBoundsException e){
            System.out.println(e);
            System.out.printf("    responses[%d]=%d%n%n",
                  answer,responses[answer]);
         }
      }
      System.out.printf("%s%10s%n","Rating","Frequency");
      for(int rating=1;rating<frequency.length;rating++)

System.out.printf("%6d%10d%n",rating,frequency[rating]);
}
}
```

Executing the catch Block

- From the above example when the program encounters the invalid value 14 in the responses array, it attempts to add 1 to frequency[14], which is outside the bounds of the array—the frequency array has only six elements (with indexes 0–5).
- Because array bounds checking is performed at execution time, the JVM generates an exception—specifically line 19 throws an ArrayIndexOutOfBoundsException to notify the program of this problem.
- At this point the try block terminates and the catch block begins executing—if you declared any local variables in the try block, they're now out of scope (and no longer exist), so they're not accessible in the catch block.
- The catch block declares an exception parameter (e) of type (IndexOutOfRangeException).
- The catch block can handle exceptions of the specified type. Inside the catch block, you can use the parameter's identifier to interact with a caught exception object.

Types of exception



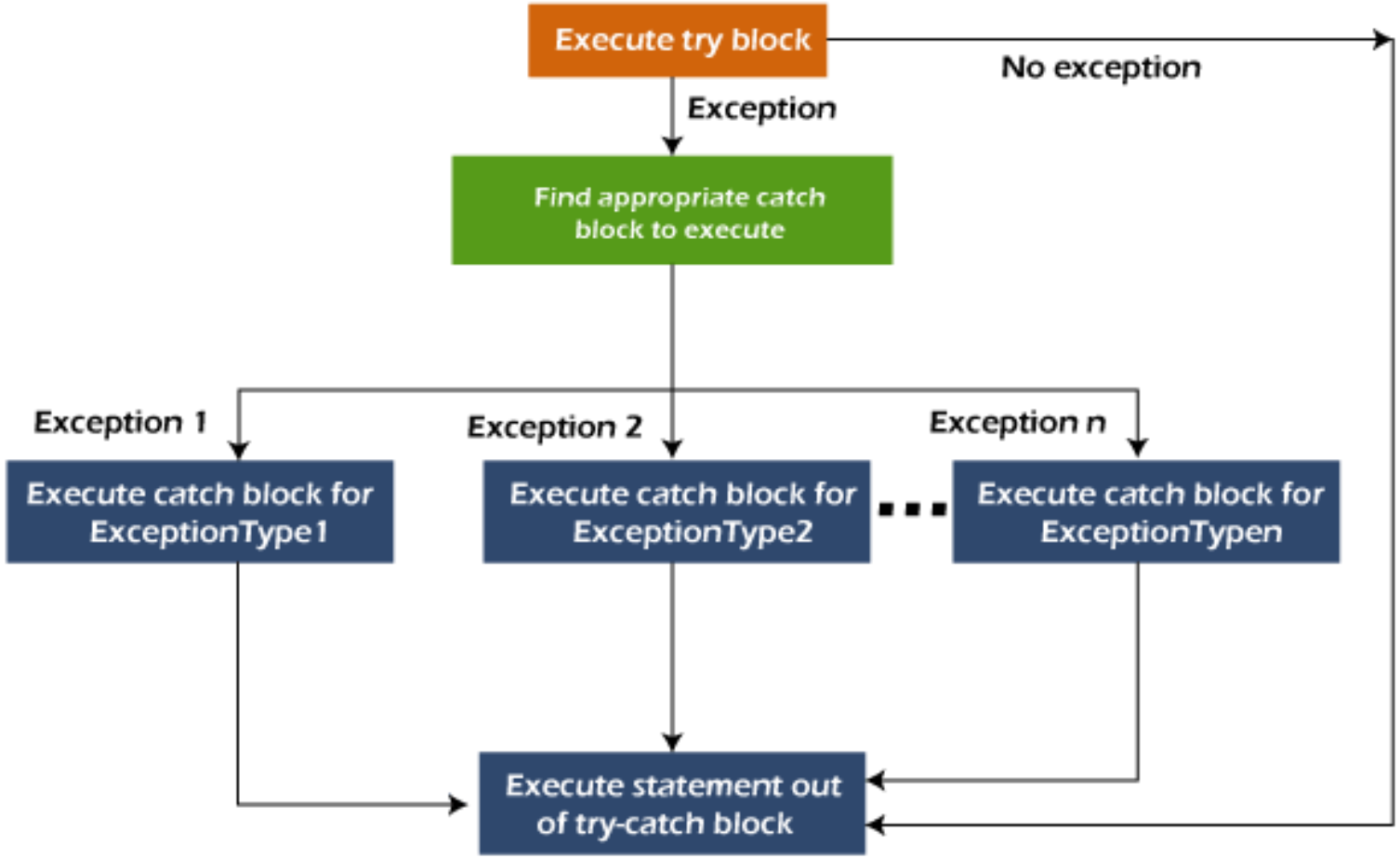

Throwable
Exception
Error
RuntimeException
IOException
AWTError
ThreadDeath
VirtualMachineError
ClassCastException
NullPointerException
ArithmeticException
IndexOutOfBoundsException
NoSuchElementException
ArrayIndexOutOfBoundsException
InputMismatchException

```
package exception;
import java.util.InputMismatchException;
import java.util.Scanner;
public class DivisionByZero {
   public static int quotient (int numerator,int denominator)throws
ArithmeticException{
   return numerator/denominator;
}
   public static void main(String[]args){
      boolean continueloop=true;
      Scanner sc=new Scanner(System.in);
      do{
         try{
              System.out.println("enter integer numerator");
              int numerator=sc.nextInt();
              System.out.println("enter integer denominator");
              int denominator=sc.nextInt();
              int result=quotient(numerator,denominator);
              System.out.printf("%nResult:  %d /%d=%d%n",
numerator,denominator,result);
              continueloop=false;
            }
 catch (InputMismatchException inputMismatchException){
              System.err.printf("%nException:
%s%n",inputMismatchException);
              sc.nextLine(); // discard input so user can try again
              System.out.printf("You must enter integers. Please try
again.%n%n");
              }
          catch (ArithmeticException arithmeticException){
              System.err.printf("%nException: %s%n",
arithmeticException);
              System.out.printf("Zero is an invalid denominator.
Please try again.%n%n");
              }
       }while(continueloop);
   }
}
```

Multi-catch

- It’s relatively common for a try block to be followed by several catch blocks to handle various types of exceptions.
- If the bodies of several catch blocks are identical, you can use the multi-catch feature to catch those exception types in a single catch handler and perform the same task.
- The syntax for a multi-catch is:
  - catch(Type1|Type2|Type3 e);
- Each exception type is separated from the next with a vertical bar (|). The preceding line of code indicates that any of the types (or their subclasses) can be caught in the exception handler.
- Any number of Throwable types can be specified in a multi-catch

Execution steps



The finally Block

- The finally block (which consists of the finally keyword, followed by code enclosed in curly braces), sometimes referred to as the finally clause, is optional.
- If it’s present, it’s placed after the last catch block.
- If there are no catch blocks, the finally block, if present, immediately follows the try block.
- The finally block will execute whether or not an exception is thrown in the corresponding try block.
- The finally block also will execute if a try block exits by using a return, break or continue statement or simply by reaching its closing right brace.

- The one case in which the finally block will not execute is if the application exits early from a try block by calling method System.exit.
  - This method, which we demonstrate, immediately terminates an application.
- If an exception that occurs in a try block cannot be caught by one of that try block's catch handlers, the program skips the rest of the try block and control proceeds to the finally block.
- Then the program passes the exception to the next outer try block—normally in the calling method—where an associated catch block might catch it.
- This process can occur through many levels of try blocks.

- Because a finally block always executes, it typically contains *resource-release code*.
- Suppose a resource is allocated in a try block. If no exception occurs, the catch blocks are *skipped* and control proceeds to the finally block, which frees the resource.
- Control then proceeds to the first statement after the finally block.
- If an exception occurs in the try block, the try block *terminates*.
- If the program catches the exception in one of the corresponding catch blocks, it processes the exception, then the finally block *releases the resource* and control proceeds to the first statement after the finally block.
- If the program doesn't catch the exception, the finally block *still* releases the resource and an attempt is made to catch the exception in a calling method.

Throwing Exceptions Using the throw Statement

- So far, we've caught only exceptions thrown by called methods.
- We can throw exceptions ourselves by using the throw statement.
- Just as with exceptions thrown by the Java API's methods, this indicates to client applications that an error has occurred.
- A throw statement specifies an object to be thrown.
- The operand of a throw can be of any class derived from class Throwable.

Rethrowing Exceptions

- Exceptions are rethrown when a catch block, upon receiving an exception, decides either that it cannot process that exception or that it can only partially process it.
- Rethrowing an exception defers the exception handling (or perhaps a portion of it) to another catch block associated with an outer try statement.
- An exception is rethrown by using the throw keyword, followed by a reference to the exception object that was just caught.
- Exceptions cannot be rethrown from a finally block, as the exception parameter (a local variable) from the catch block no longer exists.

```
public class RethrowException {
 public static void test1() throws Exception {
System.out.println("The Exception in test1() method");
    throw new Exception("thrown from test1() method");}
    catch(Exception e){
       System.out.println("exception in test1");
       throw  e;  }}
  public static void test2() throws Throwable {
    try {
      test1();}
   catch(Exception e) {
      System.out.println("Inside test2() method");
      throw e;
    }
    finally{
       System.out.println("finally in test 2");     }
  }
  public static void main(String[] args) throws Throwable {
    try {
      test2();
    } catch(Exception e) {
      System.out.println("Caught in main");
    }
  }
}
```

```
The Exception in test1() method
exception in test1
Inside test2() method
finally in test 2
Caught in main
```

User defined exceptions

- You can create your own exceptions in Java.
- All exceptions must be a child of Throwable.
  - If you want to write a checked exception that is automatically enforced by the Handle or Declare Rule, you need to extend the Exception class.
  - If you want to write a runtime exception, you need to extend the RuntimeException class.
- To create a user defined exception extend one of the above mentioned classes. To display the message override the **toString()** method or, call the superclass parameterized constructor by passing the message in String format.

AgeDoesnotMatchException.java

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

Example

Student.java

```
import java.util.Scanner;
public class Student {
  private String name;
  private int age;
  public Student(String name, int age){
    try {
      if (age<17||age>24) {
        String msg = "Age is not between 17 and 24";
        AgeDoesnotMatchException ex = new
AgeDoesnotMatchException(msg);
        throw ex;
      }
      else
      {
          this.name = name;
           this.age = age;
      }
    }
    catch(AgeDoesnotMatchException e) {
    System.out.println(e);
    }

  }
public void display(){
    System.out.println("Name of the Student: "+this.name );
    System.out.println("Age of the Student: "+this.age );
  }
  public static void main(String args[]) {
    Scanner sc= new Scanner(System.in);
    System.out.println("Enter the name of the Student: ");
    String name = sc.next();
    System.out.println("Enter the age of the Student should be
17 to 24 (ababebencluding 17 and 24): ");
    int age = sc.nextInt();
    Student obj = new Student(name, age);
    obj.display();
  }

}
```

Sample output:

Enter the name of the Student:
abebe
Enter the age of the Student should be 17 to 24
(ababebencluding 17 and 24):
5
myfirstjavaclass.AgeDoesnotMatchException: Age is not between 17 and 24
Name of the Student: null
Age of the Student: 0

UserException.java

```
class UserException extends
Exception{
int num1;
UserException(int num2) {
num1=num2;
}
public String toString(){
return ("Status code = "+num1) ;
}
}
```

SampleException.java

```
class SampleException{
public static void main(String
args[]){
try{
throw new UserException(400);
}
catch(UserException e){
System.out.println(e) ;
}
}
}
```

Sample output:
Status code=400

EmployeeException.java

```
class EmployeeException extends Exception
{
public EmployeeException(String s)
{
super(s);
}
}
```

SampleEmp.java

```
class SampleEmp
{
void empIDCheck(int EmpID) throws EmployeeException{
if(EmpID<=0 || EmpID>999){
throw new EmployeeException("Invalid Employee ID");
}
}
public static void main(String args[])
{
SampleEmp emp = new SampleEmp();
try
{
emp.empIDCheck(0);
}
catch (EmployeeException e)
{
System.out.println("Exception caught");
System.out.println(e.getMessage());
}
}
}
```

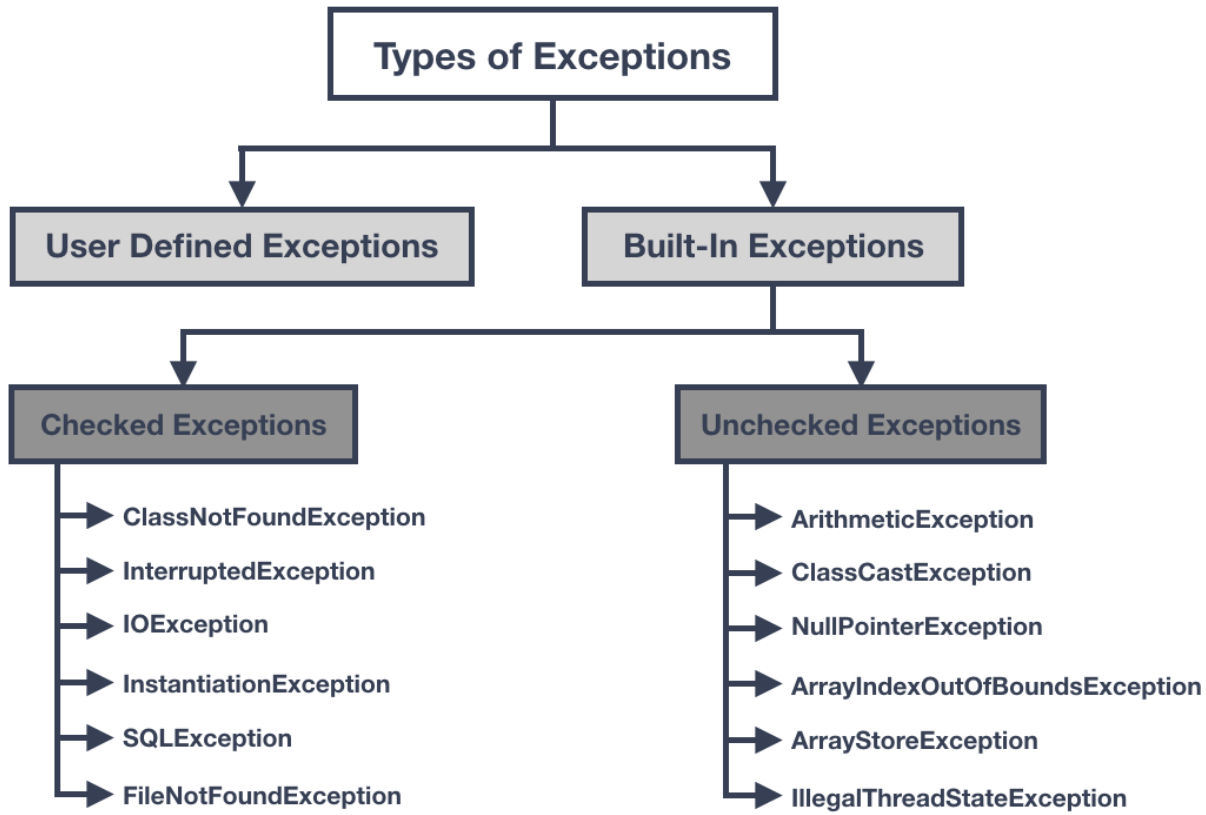
Sample output:
Exception caught
Invalid Employee ID

Errors and Runtime Exceptions

- Exceptions and errors both are subclasses of Throwable class.
- The error indicates a problem that mainly occurs due to the lack of system resources and our application should not catch these types of problems.
- Some of the examples of errors are system crash error and out of memory error.
- Errors mostly occur at runtime that's they belong to an unchecked type.
- Exceptions are the problems which can occur at runtime and compile time.
- It mainly occurs in the code written by the developers.
  - Exceptions are divided into two categories such as checked exceptions and unchecked exceptions.

Error Vs Exception

Sr. No.	Key	Error	Exception
1	Type	Classified as an unchecked type	Classified as checked and unchecked
2	Package	It belongs to java.lang.error	It belongs to java.lang.Exception
3	Recoverable/ Irrecoverable	It is irrecoverable	It is recoverable
4		It can't be occur at compile time	It can occur at run time compile time both
5	Example	OutOfMemoryError ,IOError	NullPointerException , SqlException


Types of Exceptions
User Defined Exceptions
Built-In Exceptions
Checked Exceptions
Unchecked Exceptions
ClassNotFoundException
InterruptedException
IOException
InstantiationException
SQLException
FileNotFoundException
ArithmeticException
ClassCastException
NullPointerException
ArrayIndexOutOfBoundsException
ArrayStoreException
IllegalThreadStateException

```
public class ErrorExample {
  public static void main(String[] args){
    recursiveMethod(10)
  }
  public static void recursiveMethod(int i){
    while(i!=0){
      i=i+1;
      recursiveMethod(i);
    }
  }
}
```

```
Exception in thread "main" java.lang.StackOverflowError
  at ErrorExample.ErrorExample(Main.java:15)
```