# Chapter 12 Exception Handling




# Motivations

- When a program runs into a runtime error, the program terminates abnormally.
- How can you handle the runtime error so that the program can continue to run or terminate gracefully?




# Exception-Handling Overview

LISTING 12.1 Quotient.java

```
1  import java.util.Scanner;
2
3  public class Quotient {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6
7      // Prompt the user to enter two integers
8      System.out.print("Enter two integers: ");
9      int number1 = input.nextInt();
10     int number2 = input.nextInt();
11
12     System.out.println(number1 + " / " + number2 + " is " +
13       (number1 / number2));
14   }
15 }
```

```
Enter two integers: 5 2 Enter
5 / 2 is 2
```

```
Enter two integers: 3 0 Enter
Exception in thread "main" java.lang.ArithmeticException: / by zero
at Quotient.main(Quotient.java:11)
```




# Exception-Handling Overview

LISTING 12.2 QuotientWithIf.java

```
1  import java.util.Scanner;
2
3  public class QuotientWithIf {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6
7      // Prompt the user to enter two integers
8      System.out.print("Enter two integers: ");
9      int number1 = input.nextInt();
10     int number2 = input.nextInt();
11
12     if (number2 != 0)
13       System.out.println(number1 + " / " + number2
14         + " is " + (number1 / number2));
15     else
16       System.out.println("Divisor cannot be zero ");
17   }
18 }
```

```
Enter two integers: 5 0 Enter
Divisor cannot be zero
```

# Exception-Handling Overview

**LISTING 12.3** QuotientWithMethod.java

```
1  import java.util.Scanner;
2
3  public class QuotientWithMethod {
4    public static int quotient(int number1, int number2) {
5      if (number2 == 0) {
6        System.out.println("Divisor cannot be zero");
7        System.exit(1);
8      }
9
10     return number1 / number2;
11   }
12
13   public static void main(String[] args) {
14     Scanner input = new Scanner(System.in);
15
16     // Prompt the user to enter two integers
17     System.out.print("Enter two integers: ");
18     int number1 = input.nextInt();
19     int number2 = input.nextInt();
20
21     int result = quotient(number1, number2);
22     System.out.println(number1 + " / " + number2 + " is "
23       + result);
24   }
25 }
```

```
Enter two integers: 5 3 Enter
5 / 3 is 1
```

```
Enter two integers: 5 0 Enter
Divisor cannot be zero
```



# Exception Advantages

**LISTING 12.4** QuotientWithException.java

```
1  import java.util.Scanner;
2
3  public class QuotientWithException {
4    public static int quotient(int number1, int number2) {
5      if (number2 == 0)
6        throw new ArithmeticException("Divisor cannot be zero");
7
8      return number1 / number2;
9    }
10
11   public static void main(String[] args) {
12     Scanner input = new Scanner(System.in);
13
14     // Prompt the user to enter two integers
15     System.out.print("Enter two integers: ");
16     int number1 = input.nextInt();
17     int number2 = input.nextInt();
18
19     try {
20       int result = quotient(number1, number2);
21       System.out.println(number1 + " / " + number2 + " is "
22         + result);
23     }
24     catch (ArithmeticException ex) {
25       System.out.println("Exception: an integer " +
26         "cannot be divided by zero ");
27     }
28
29     System.out.println("Execution continues ...");
30   }
31 }
```

If an ArithmeticException occurs

```
Enter two integers: 5 3 Enter
5 / 3 is 1
Execution continues ...
```

```
Enter two integers: 5 0 Enter
Exception: an integer cannot be divided by zero
Execution continues ...
```



# Exception Advantages

- It enables a method to throw an exception to its caller.
- Without this capability, a method must handle the exception or terminate the program.



# Handling InputMismatchException

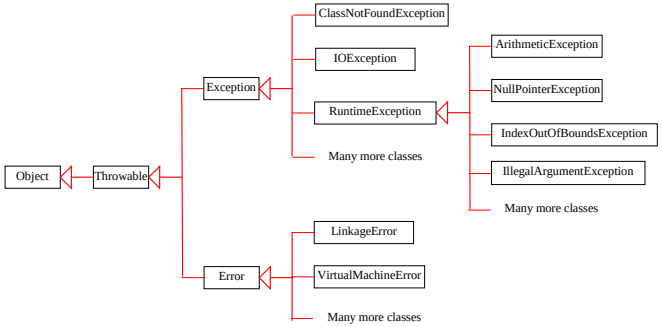
**LISTING 12.5** InputMismatchExceptionDemo.java

```
1  import java.util.*;
2
3  public class InputMismatchExceptionDemo {
4    public static void main(String[] args) {
5      Scanner input = new Scanner(System.in);
6      boolean continueInput = true;
7
8      do {
9        try {
10         System.out.print("Enter an integer: ");
11         int number = input.nextInt();
12
13         // Display the result
14         System.out.println(
15           "The number entered is " + number);
16
17         continueInput = false;
18       }
19       catch (InputMismatchException ex) {
20         System.out.println("Try again. (" +
21           "Incorrect input: an integer is required)");
22         input.nextLine(); // Discard input
23       }
24     } while (continueInput);
25   }
26 }
```

If an InputMismatchException occurs

By handling InputMismatchException, your program will continuously read an input until it is correct.

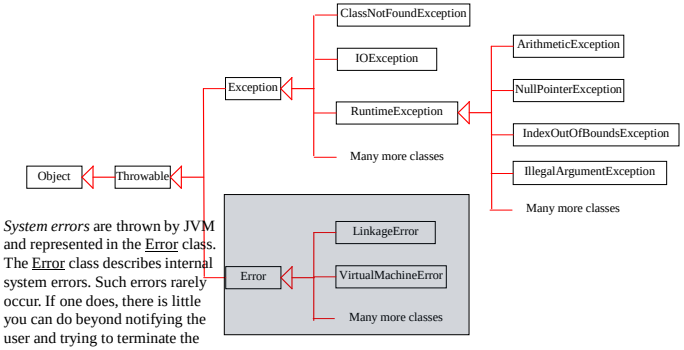
# Exception Types








# System Errors



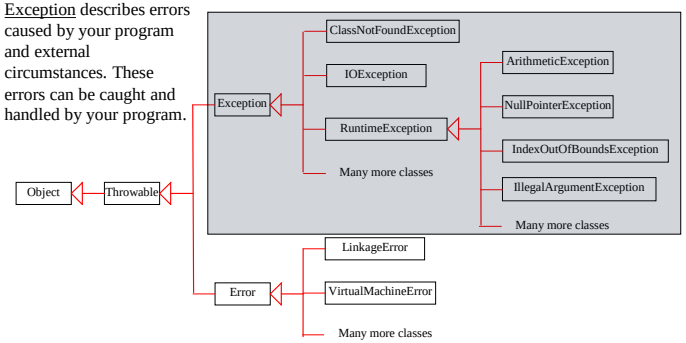

*System errors* are thrown by JVM and represented in the Error class. The Error class describes internal system errors. Such errors rarely occur. If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully.





# Exceptions

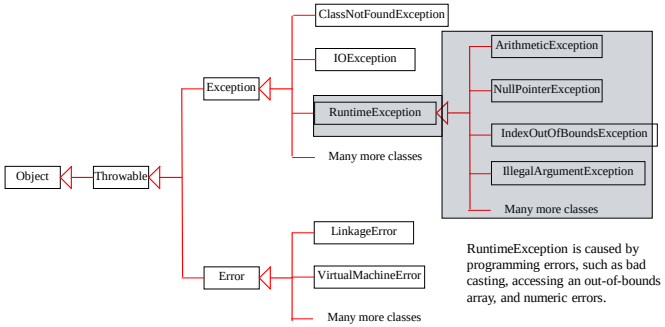
Exception describes errors caused by your program and external circumstances. These errors can be caught and handled by your program.








# Runtime Exceptions




RuntimeException is caused by programming errors, such as bad casting, accessing an out-of-bounds array, and numeric errors.

## Checked Exceptions vs. Unchecked Exceptions

- RuntimeException, Error and their subclasses are known as *unchecked exceptions*.
- All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with the exceptions.



## Unchecked Exceptions

- In most cases, unchecked exceptions reflect programming logic errors that are not recoverable.
- For example, 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.
- These are the logic errors that should be corrected in the program.

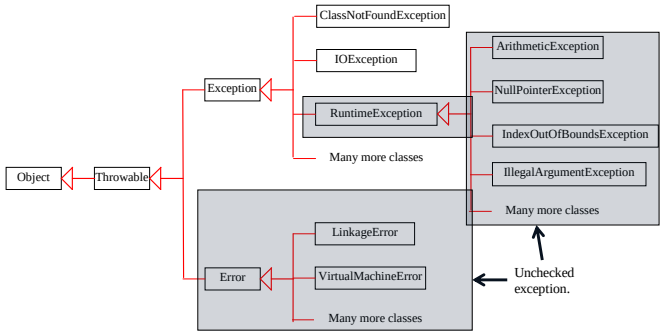


## Unchecked Exceptions

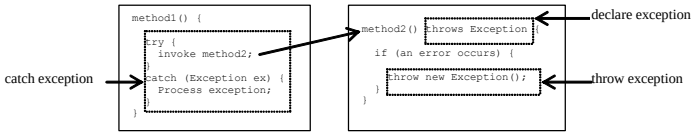
- Unchecked exceptions can occur anywhere in the program.
- To avoid cumbersome overuse of try-catch blocks, Java does not mandate you to write code to catch unchecked exceptions.



## Unchecked Exceptions



# Declaring, Throwing, and Catching Exceptions






# Declaring Exceptions

Every method must state the types of checked exceptions it might throw. This is known as *declaring exceptions*.

```
public void myMethod()
    throws IOException

public void myMethod()throws
IOException,OtherException
```



# Throwing Exceptions

When the program detects an error, the program can create an instance of an appropriate exception type and throw it. This is known as *throwing an exception*. Here is an example,

```
throw new TheException();

TheException ex = new TheException();
throw ex;
```



# Throwing Exceptions Example

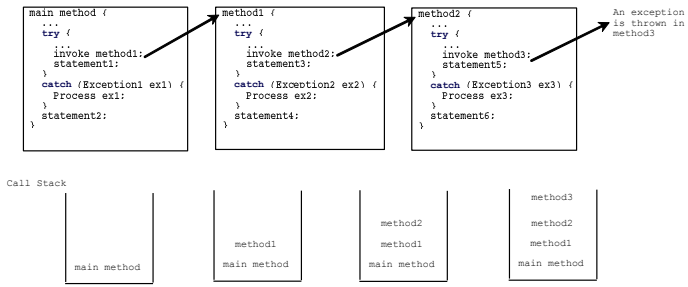
```
/** Set a new radius */
 public void setRadius(double newRadius)
     throws IllegalArgumentException {
   if (newRadius >= 0)
     radius =  newRadius;
   else
     throw new IllegalArgumentException(
       "Radius cannot be negative");
 }
```

# Catching Exceptions

```
try {
  statements;  // Statements that may throw exceptions
}
catch (Exception1 exVar1) {
  handler for exception1;
}
catch (Exception2 exVar2) {
  handler for exception2;
}
...
catch (ExceptionN exVar3) {
  handler for exceptionN;
}
```



# Catching Exceptions






# Catch or Declare Checked Exceptions

Java forces you to deal with checked exceptions. If a method declares a checked exception (i.e., an exception other than Error or RuntimeException), you must invoke it in a try-catch block or declare to throw the exception in the calling method. For example, suppose that method p1 invokes method p2 and p2 may throw a checked exception (e.g., IOException), you have to write the code as shown in (a) or (b).

```
void p1() {
  try {
    p2();
  }
  catch (IOException ex) {
    ...
  }
}
```

(a)

```
void p1() throws IOException {
  p2();
}
```

(b)



# Example: Declaring, Throwing, and Catching Exceptions

Objective: This example demonstrates declaring, throwing, and catching exceptions by modifying the setRadius method in the Circle class defined in previous Chapter. The new setRadius method throws an exception if radius is negative.

## LISTING 12.7 CircleWithException.java

```
1  public class CircleWithException {
2    /** The radius of the circle */
3    private double radius;
4
5    /** The number of the objects created */
6    private static int numberOfObjects = 0;
7
8    /** Construct a circle with radius 1 */
9    public CircleWithException() {
10     this(1.0);
11   }
12
13   /** Construct a circle with a specified radius */
14   public CircleWithException(double newRadius) {
15     setRadius(newRadius);
16     numberOfObjects++;
17   }
18
19   /** Return radius */
20   public double getRadius() {
21     return radius;
22   }
23
```



```
24   /** Set a new radius */
25   public void setRadius(double newRadius)
26       throws IllegalArgumentException {
27     if (newRadius >= 0)
28       radius = newRadius;
29     else
30       throw new IllegalArgumentException(
31         "Radius cannot be negative");
32   }
33
34   /** Return numberOfObjects */
35   public static int getNumberOfObjects() {
36     return numberOfObjects;
37   }
38
39   /** Return the area of this circle */
40   public double findArea() {
41     return radius * radius * 3.14159;
42   }
43 }
```



## LISTING 12.8 TestCircleWithException.java

```
1  public class TestCircleWithException {
2    public static void main(String[] args) {
3      try {
4        CircleWithException c1 = new CircleWithException(5);
5        CircleWithException c2 = new CircleWithException(-5);
6        CircleWithException c3 = new CircleWithException(0);
7      }
8      catch (IllegalArgumentException ex) {
9        System.out.println(ex);
10     }
11
12     System.out.println("Number of objects created: " +
13       CircleWithException.getNumberOfObjects());
14   }
15 }
```

```
java.lang.IllegalArgumentException: Radius cannot be negative
Number of objects created: 1
```

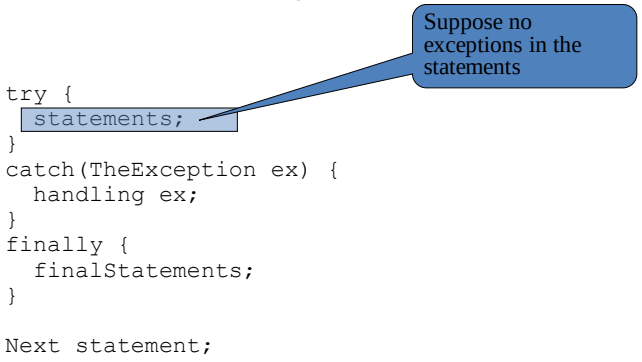


## The `finally` Clause

The **finally** clause is always executed regardless whether an exception occurred or not.

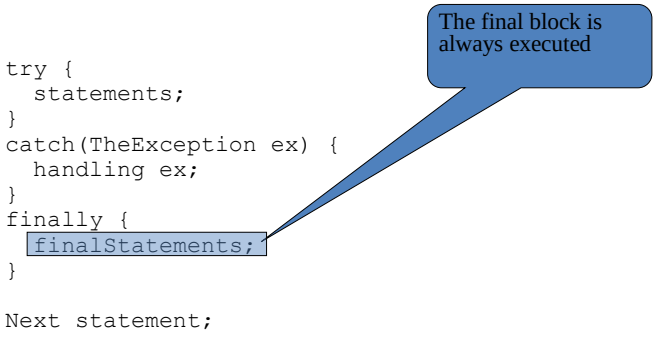
```
try {
  statements;
}
catch(TheException ex) {
  handling ex;
}
finally {
  finalStatements;
}
```

## Trace a Program Execution



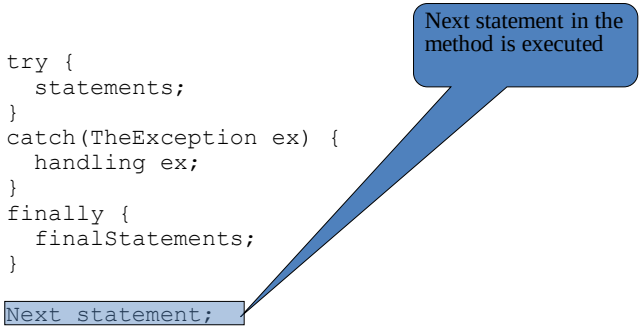



## Trace a Program Execution



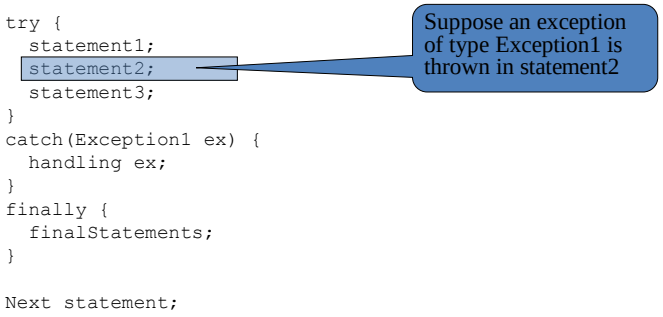



## Trace a Program Execution

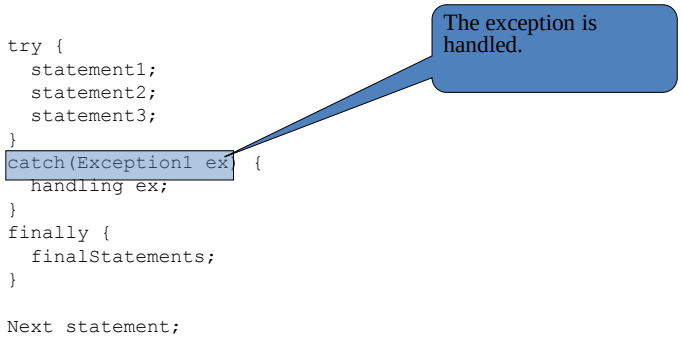





## Trace a Program Execution

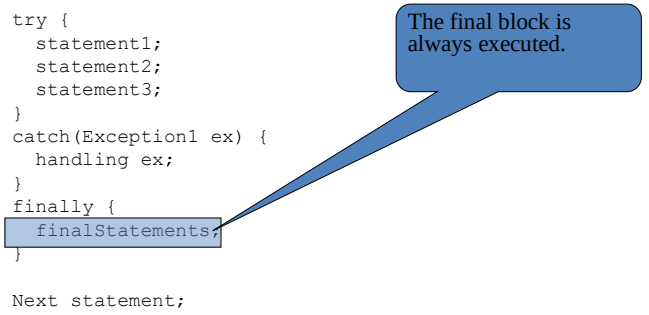


## Trace a Program Execution



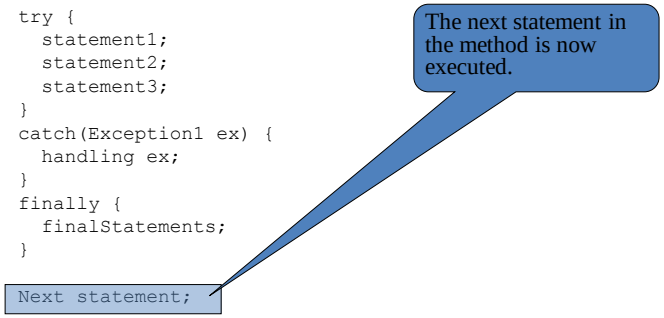



## Trace a Program Execution



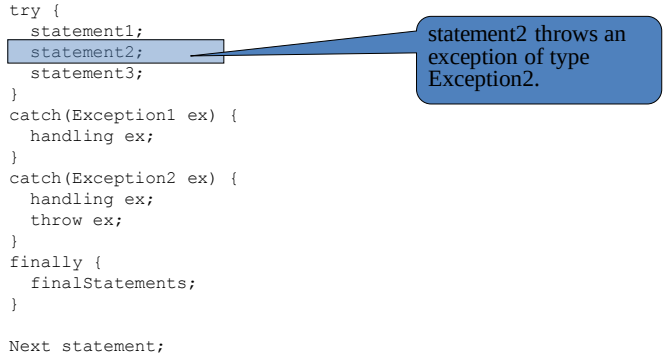



## Trace a Program Execution






## Trace a Program Execution



## Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Handling exception



## Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Execute the final block



## Trace a Program Execution

```
try {
  statement1;
  statement2;
  statement3;
}
catch(Exception1 ex) {
  handling ex;
}
catch(Exception2 ex) {
  handling ex;
  throw ex;
}
finally {
  finalStatements;
}

Next statement;
```

Rethrow the exception and control is transferred to the caller



## Cautions When Using Exceptions

- Exception handling separates error-handling code from normal programming tasks, thus making programs easier to read and to modify.
- Be aware, however, that exception handling usually requires more time and resources because it requires instantiating a new exception object, rolling back the call stack, and propagating the errors to the calling methods.

## When to Throw Exceptions

- An exception occurs in a method.
- If you want the exception to be processed by its caller, you should create an exception object and throw it.
- If you can handle the exception in the method where it occurs, there is no need to throw it.



## When to Use Exceptions

When should you use the try-catch block in the code? You should use it to deal with unexpected error conditions. Do not use it to deal with simple, expected situations. For example, the following code

```
try {
  System.out.println(refVar.toString());
}
catch (NullPointerException ex) {
  System.out.println("refVar is null");
}
```



## When to Use Exceptions

is better to be replaced by

```
if (refVar != null)
  System.out.println(refVar.toString());
else
  System.out.println("refVar is null");
```



## Defining Custom Exception Classes

- Use the exception classes in the API whenever possible.
- Define custom exception classes if the predefined classes are not sufficient.
- Define custom exception classes by extending Exception or a subclass of Exception.

# Custom Exception Class Example

**LISTING 12.10 InvalidRadiusException.java**

```
1  public class InvalidRadiusException extends Exception {
2    private double radius;
3
4    /** Construct an exception */
5    public InvalidRadiusException(double radius) {
6      super("Invalid radius " + radius);
7      this.radius = radius;
8    }
9
10   /** Return the radius */
11   public double getRadius() {
12     return radius;
13   }
14 }
```



```
17 class CircleWithCustomException {
18   /** The radius of the circle */
19   private double radius;
20
21   /** The number of objects created */
22   private static int numberOfObjects = 0;
23
24   /** Construct a circle with radius 1 */
25   public CircleWithCustomException() throws InvalidRadiusException {
26     this(1.0);
27   }
28
29   /** Construct a circle with a specified radius */
30   public CircleWithCustomException(double newRadius)
31       throws InvalidRadiusException {
32     setRadius(newRadius);
33     numberOfObjects++;
34   }
35
36   /** Return radius */
37   public double getRadius() {
38     return radius;
39   }
40
41   /** Set a new radius */
42   public void setRadius(double newRadius)
43       throws InvalidRadiusException {
44     if (newRadius >= 0)
45       radius = newRadius;
46     else
47       throw new InvalidRadiusException(newRadius);
48   }
49
50   /** Return numberOfObjects */
51   public static int getNumberOfObjects() {
52     return numberOfObjects;
53   }
54
55   /** Return the area of this circle */
56   public double findArea() {
57     return radius * radius * 3.14159;
58   }
59 }
```



**LISTING 12.11 TestCircleWithCustomException.java**

```
1  public class TestCircleWithCustomException {
2    public static void main(String[] args) {
3      try {
4        new CircleWithCustomException(5);
5        new CircleWithCustomException(-5);
6        new CircleWithCustomException(0);
7      }
8      catch (InvalidRadiusException ex) {
9        System.out.println(ex);
10     }
11
12     System.out.println("Number of objects created: " +
13       CircleWithCustomException.getNumberOfObjects());
14   }
15 }
16
```

```
InvalidRadiusException: Invalid radius -5.0
Number of objects created: 1
```