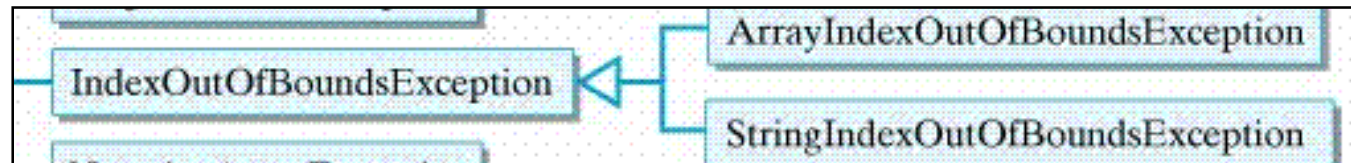
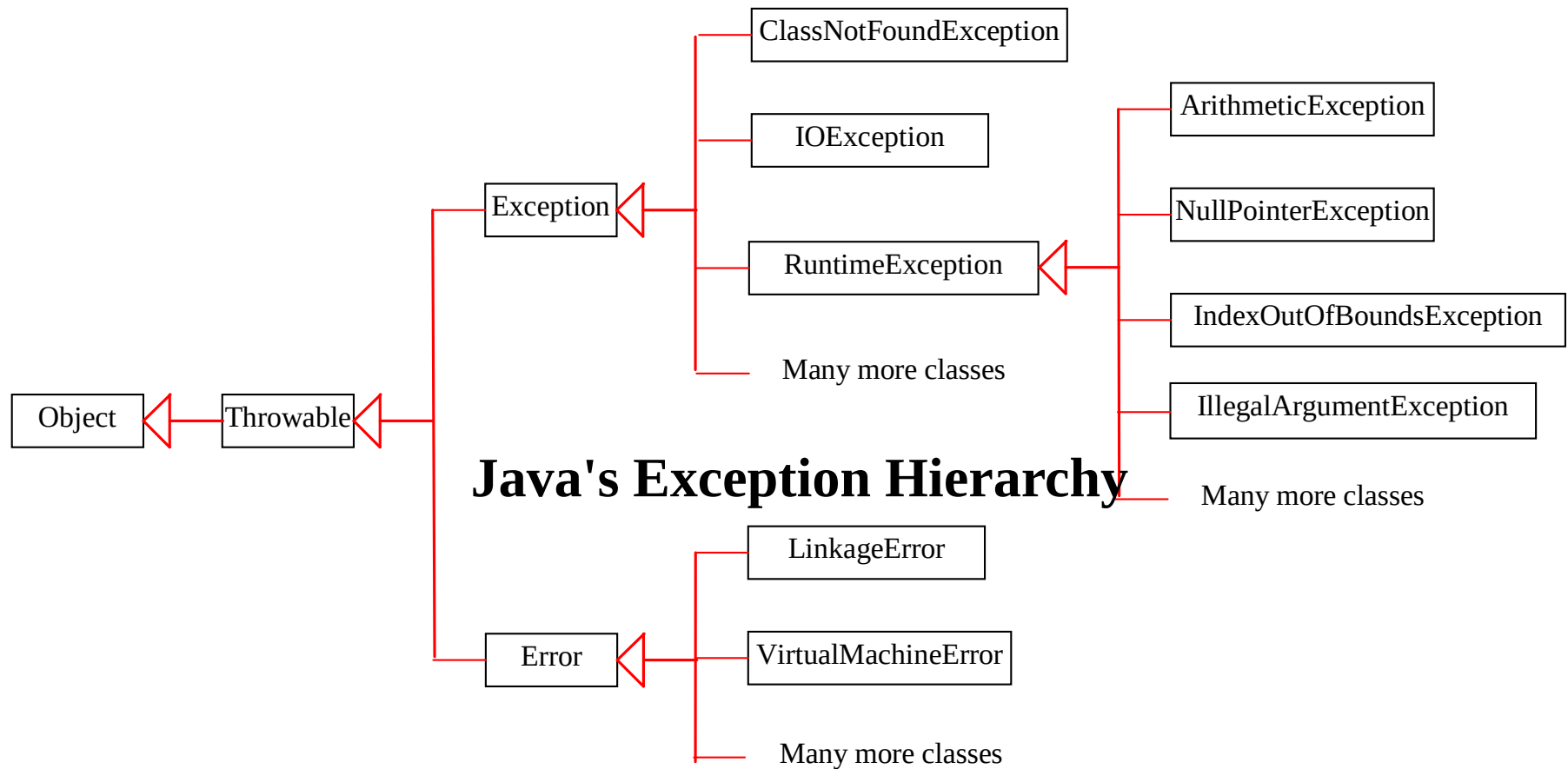


Exception Handling and Text I/O

Exception Types (Continued)

`InputMismatchException`, `IllegalArgumentException`, `Exception`
More on Exception Handling

Exception Types



Each of the classes above identifies a particular type of exception, and each is a subclass of the Exception class. Obviously a subclass defines a more specific exception than its superclass.

InputMismatchException Example 1

```
import java.util.InputMismatchException;
import java.util.Scanner;

public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        int n, m, sum;
        System.out.println("Enter two integers:");
        try {
            n = scan.nextInt();
            m = scan.nextInt();
            sum = n + m;
            System.out.println("sum is: " + sum);
        }
        catch(InputMismatchException ime) {
            System.out.println("You should enter integers.");
        }

        System.out.println("The end.");
    }
}
```

Handling InputMismatchException

The program in the previous slide (slide 15) will output the sum of the two input values and then the message “The end.” if the user enters two integers.

However, if at least one of the two input values is not an integer, then the program will output the below two messages:

```
You should enter integers.
```

```
The end.
```

InputMismatchException Example 2

```
import java.util.*;
public class Main {
    public static void main(String[] args) {
        int n, m, sum;
        Scanner scan = new Scanner(System.in);
        boolean repeat = true;
        System.out.print("Enter two integers: ");
        do {
            try {
                n = scan.nextInt();
                m = scan.nextInt();
                sum = n + m;
                System.out.println("sum is: " + sum);
                repeat = false;
            }
            catch (InputMismatchException ime) {
                scan.nextLine(); // To skip the invalid input
                System.out.print("Enter two integers: ");
            }
        } while (repeat);
        System.out.println("The end.");
    }
}
```

InputMismatchException

The program in the previous slide (slide 17) will output the sum of the two input values and then the message “The end.” if the user enters two integers.

However, if at least one of the two input values is not an integer, then the program will continuously read two input values until they are both integers.

IllegalArgumentException Example

```
public class Student{
    private int id;
    private String name;
    public Student(int i, String n) throws IllegalArgumentException {
        if (i < 0)
            throw new IllegalArgumentException("ID cannot be negative.");
        if (n.length() < 2)
            throw new IllegalArgumentException("Name should have at least 2 letters.");
        id = i;
        name = n;
    }
    public void print() {
        System.out.println(ID + " " + name);
    }
}

import java.util.Scanner;
public class Main {
    public static void main(String[] args) {
        Scanner scan = new Scanner(System.in);
        String nm;
        int id;
```

IllegalArgumentException Example Continued

```
System.out.println("Enter the name and the id: ");
String name = scan.next();
int id = scan.nextInt();
try
{
    Student s = new Student(id,name);
    s.print();
}
catch(IllegalArgumentException iae)
{
    System.out.println(iae.getMessage());
}
}
```

Handling IllegalArgumentException

The program in the previous two slides (slides 22 and 23) will output the message `'ID cannot be negative.'` if we pass to the constructor a negative value for the argument `id`.

The program will output the message `'Name should have at least 2 letters.'` if we pass to the constructor an argument `name` with `length < 2`.

Otherwise, the program will output the `id` and `name` of the student object.

Checked Exceptions vs. Unchecked Exceptions

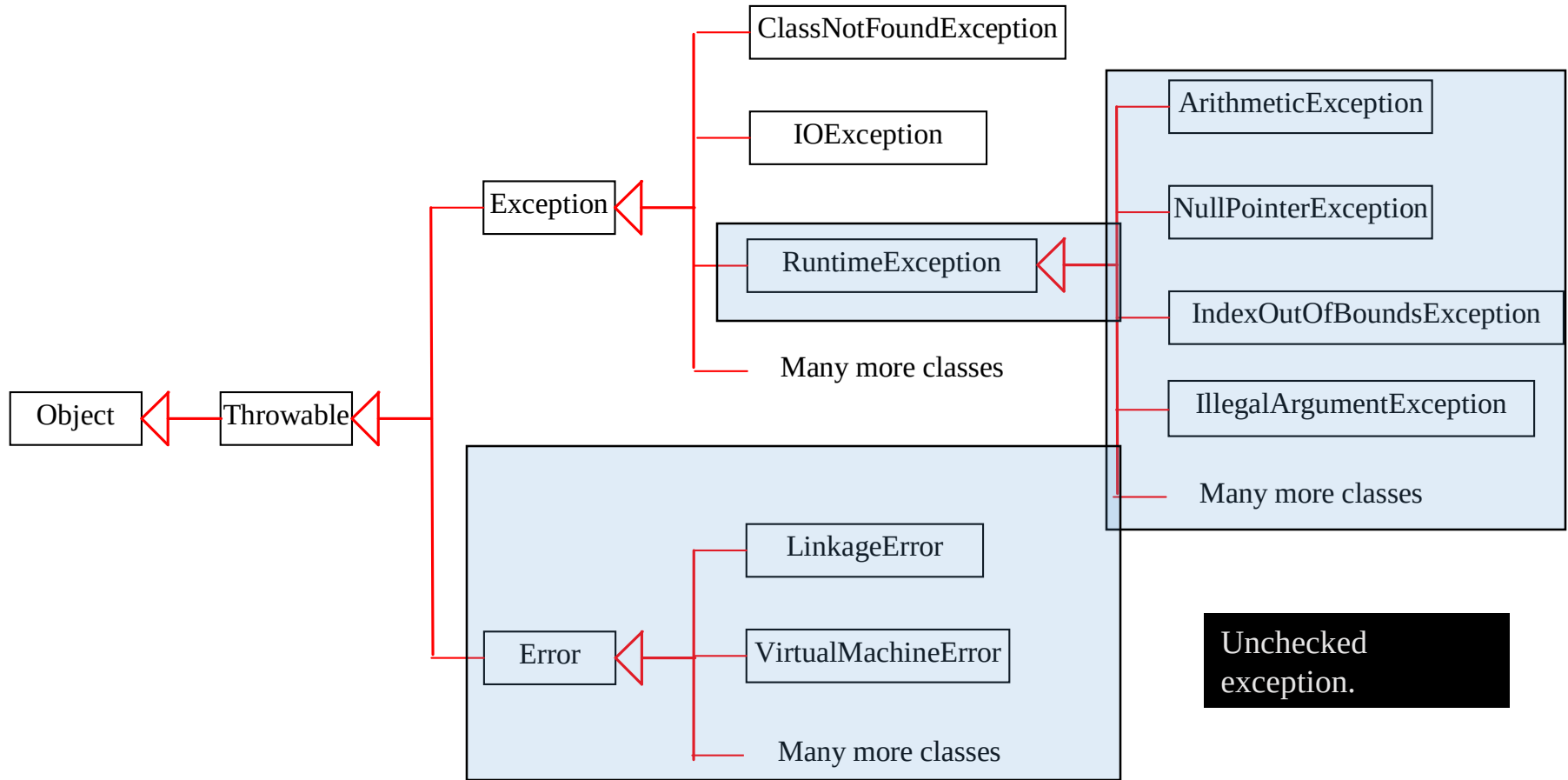
RuntimeException, Error and their subclasses are known as *unchecked exceptions*.

For example, the following exceptions are all unchecked exceptions:

ArithmeticException,
NullPointerException,
IndexOutOfBoundsException, InputMismatchException,
IllegalArgumentException.

All other exceptions are known as *checked exceptions*, meaning that the compiler forces the programmer to check and deal with those exceptions.

Unchecked 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 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.

The Exception Class

The Exception class has a set of sub-classes for handling different types of exceptions such as **IOException**, **RuntimeException**, etc.

We can replace all the subclasses of Exception by the class Exception itself as in the below:

```
try {  
    ...  
}  
catch (Exception e) {  
    ...  
}
```

The **Exception** class provides the following constructors:

1. `public Exception( )`

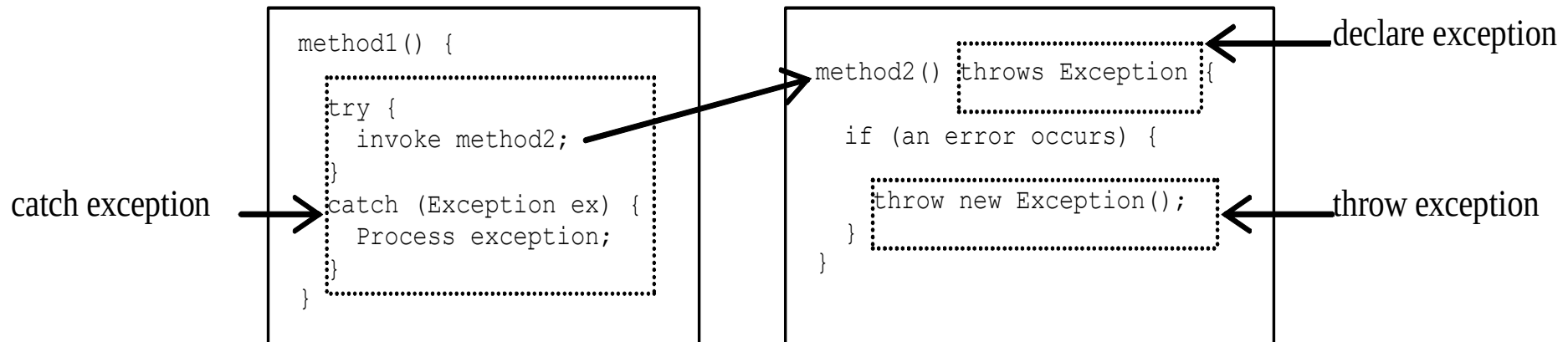
This one constructs an exception with a null detail message.

2. `public Exception(String message)`

This one constructs an exception with the given detailed message.

More on Exception Handling

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;
```

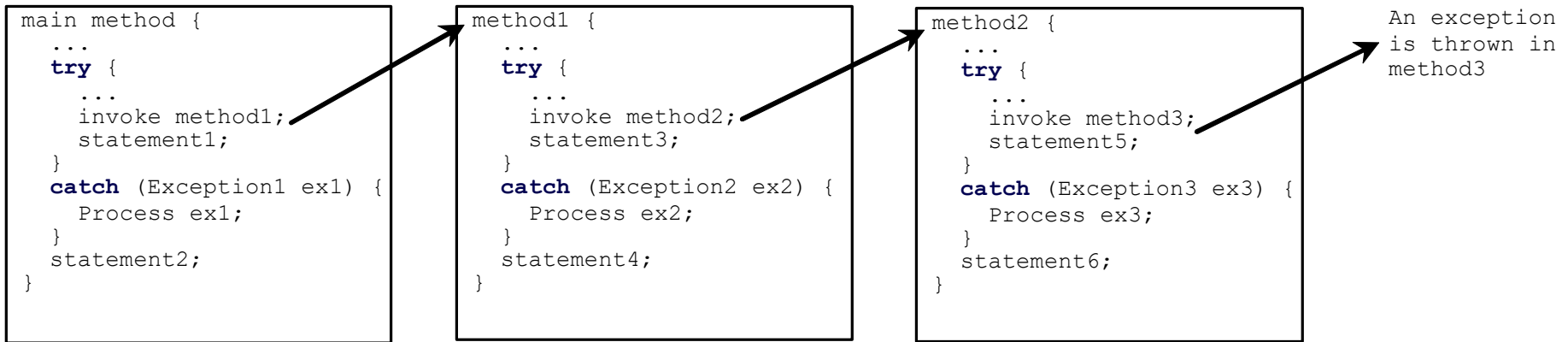
Throwing Exceptions Example

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

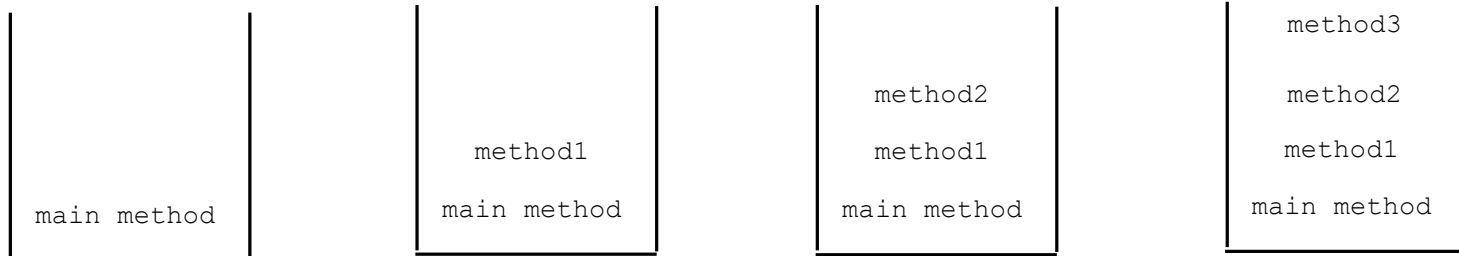
Catching Exceptions

```
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



Call Stack



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)

Getting Information from Exceptions

An exception object contains valuable information about the exception. You may use the following instance methods in the **java.lang.Throwable** class to get information regarding the exception, as shown in the figure below.

java.lang.Throwable

```
+getMessage(): String  
+toString(): String  
+printStackTrace(): void
```

Returns the message that describes this exception object.

Returns the concatenation of three strings: (1) the full name of the exception class; (2) ":" (a colon and a space); (3) the `getMessage()` method.

Prints the `Throwable` object and its call stack trace information on the console.

Getting Information from Exceptions

java.lang.Throwable

+getMessage(): String
+toString(): String
+printStackTrace(): void

Returns the message that describes this exception object.

Returns the concatenation of three strings: (1) the full name of the exception class; (2) ":" (a colon and a space); (3) the `getMessage()` method.

Prints the `Throwable` object and its call stack trace information on the console.

```
try {  
    Student s = new Student(id,name);  
    s.print();  
}  
catch (IllegalArgumentException iae) {  
    System.out.println(iae.getMessage());  
    // or  
    System.out.println(iae.toString());  
    //  
    iae.printStackTrace();  
}
```