



Exception Handling

CSGE601021 Dasar-Dasar Pemrograman 2
Fakultas Ilmu Komputer Universitas Indonesia

Try this program: write division

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

```
public class Quotient {
```

```
    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();
```

```
        System.out.println(number1 + " / " + number2 + " is " + (number1 / number2));
```

```
    }
```

```
}
```

What will happen if
we enter 2 and 0?

Runtime Error

Command Prompt

```
C:\Users\Laksmitha\Desktop\Demo>javac Quotient.java

C:\Users\Laksmitha\Desktop\Demo>java Quotient
Enter two integers: 2
0
Exception in thread "main" java.lang.ArithmeticException: / by zero
    at Quotient.main(Quotient.java:12)

C:\Users\Laksmitha\Desktop\Demo>
```

How can we handle this?

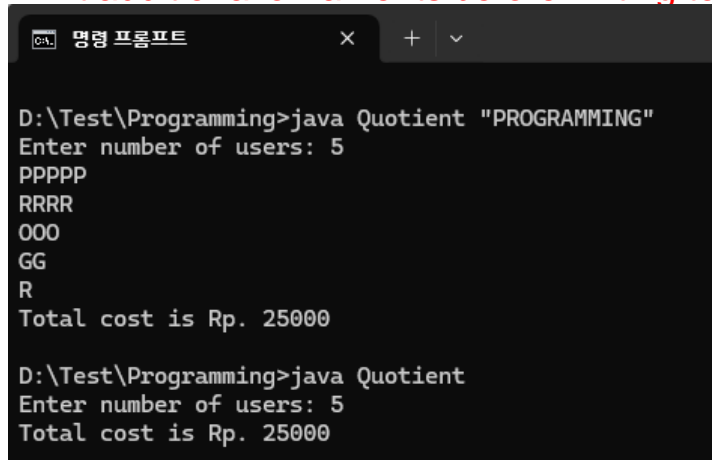
Try this program: write total cost

```
import java.util.Scanner;

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

        System.out.print("Enter number of users: ");
        int number = input.nextInt();
        if (args.length != 0) {
            String name = args[0];
            for (int i=0; i<number; i++) {
                for (int j=number-1; j>=i; j--) {
                    System.out.print(name.charAt(i));
                }
                System.out.println("");
            }
        }
        System.out.println("Total cost is Rp. " + (number * 5000));
    }
}
```

Print additional ornaments before writing total cost



```
D:\Test\Programming>java Quotient "PROGRAMMING"
Enter number of users: 5
PPPPP
RRRR
OOO
GG
R
Total cost is Rp. 25000

D:\Test\Programming>java Quotient
Enter number of users: 5
Total cost is Rp. 25000
```

What could be the problem?

Another Runtime Error

```
명령 프롬프트
D:\Test\Programming>java Quotient "DDP2"
Enter number of users: 5
DDDDD
DDDD
PPP
22
Exception in thread "main" java.lang.StringIndexOutOfBoundsException: Index 4 out of bounds for length 4
    at java.base/jdk.internal.util.Preconditions$1.apply(Preconditions.java:55)
    at java.base/jdk.internal.util.Preconditions$1.apply(Preconditions.java:52)
    at java.base/jdk.internal.util.Preconditions$4.apply(Preconditions.java:213)
    at java.base/jdk.internal.util.Preconditions$4.apply(Preconditions.java:210)
    at java.base/jdk.internal.util.Preconditions.outOfBounds(Preconditions.java:98)
    at java.base/jdk.internal.util.Preconditions.outOfBoundsCheckIndex(Preconditions.java:106)
    at java.base/jdk.internal.util.Preconditions.checkIndex(Preconditions.java:302)
    at java.base/java.lang.String.checkIndex(String.java:4832)
    at java.base/java.lang.StringLatin1.charAt(StringLatin1.java:46)
    at java.base/java.lang.String.charAt(String.java:1555)
    at Quotient.main(Quotient.java:14)
D:\Test\Programming>
```

How can we handle this?

With If-Clause

```
import java.util.Scanner;

public class QuotientWithIf {
    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();

        if (number2 != 0)
            System.out.println(number1 + " / " + number2 + " is " +
                               (number1 / number2));
        else
            System.out.println("Divisor cannot be zero ");
    }
}
```

With Method

```
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);
    }
}
```

Exception Handling

<http://www.cs.armstrong.edu/liang/intro10e/html/QuotientWithException.html?>

```
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 ");  
}
```


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.
- By throwing an exception, the program can still continue

Handling InputMismatchException

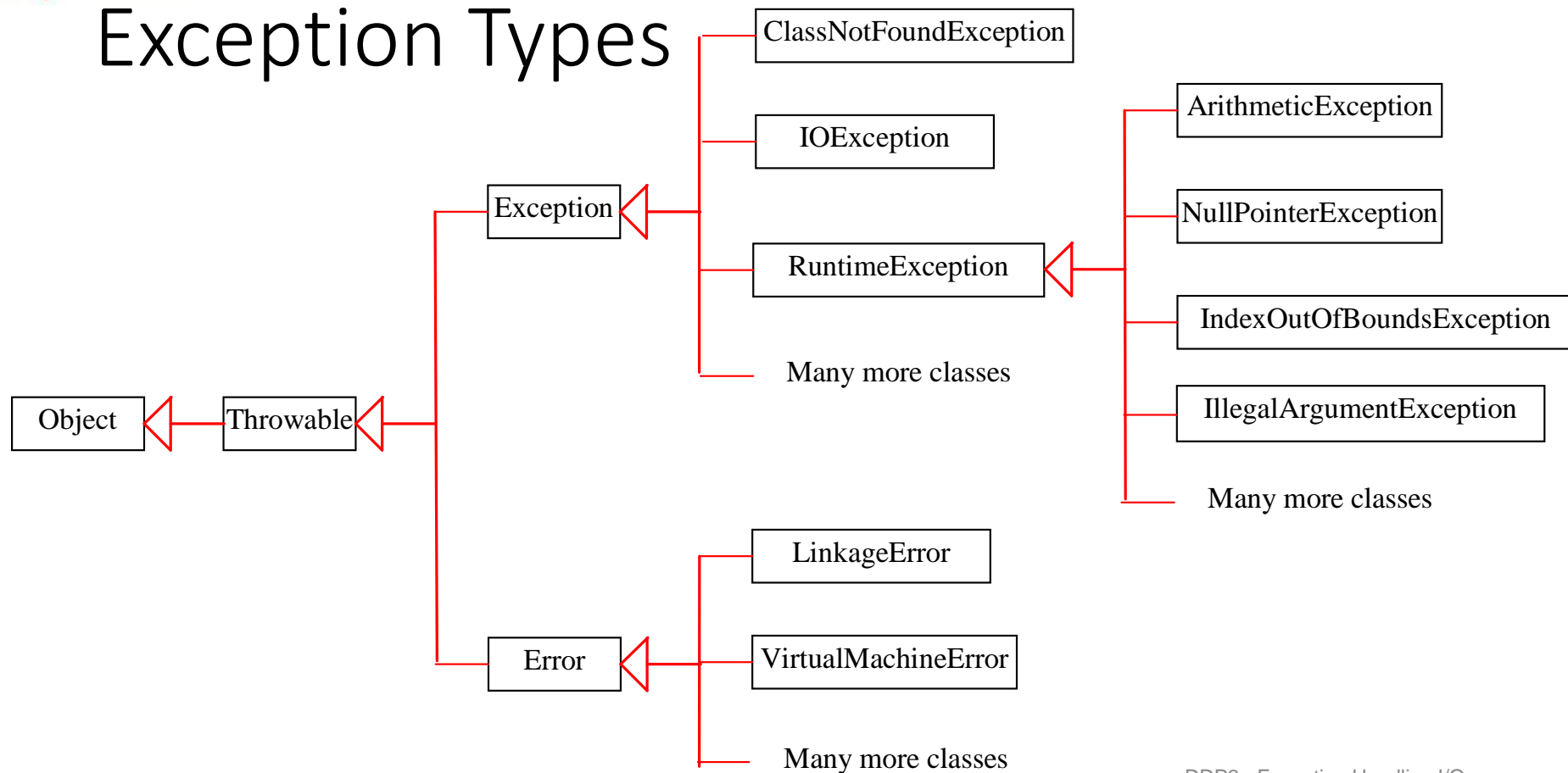
```
Scanner input = new Scanner(System.in);
boolean continueInput = true;
do {
    try {
        System.out.print("Enter an integer: ");
        int number = input.nextInt();

        // Display the result
        System.out.println(
            "The number entered is " + number);

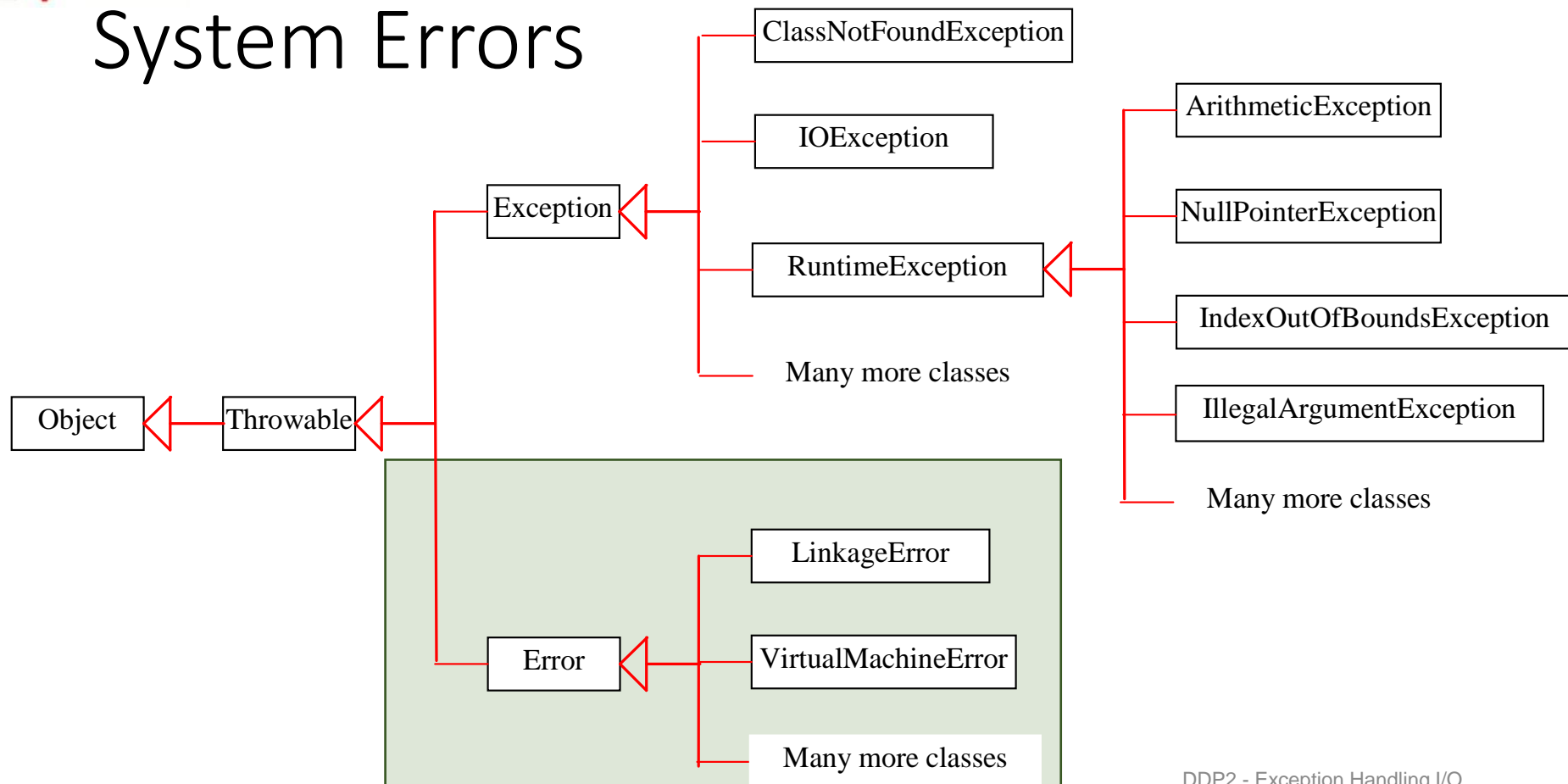
        continueInput = false;
    }
    catch (InputMismatchException ex) {
        System.out.println("Try again. (" +
            "Incorrect input: an integer is required)");
        input.nextLine(); // discard input
    }
} while (continueInput);
```

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

Exception Types



System Errors



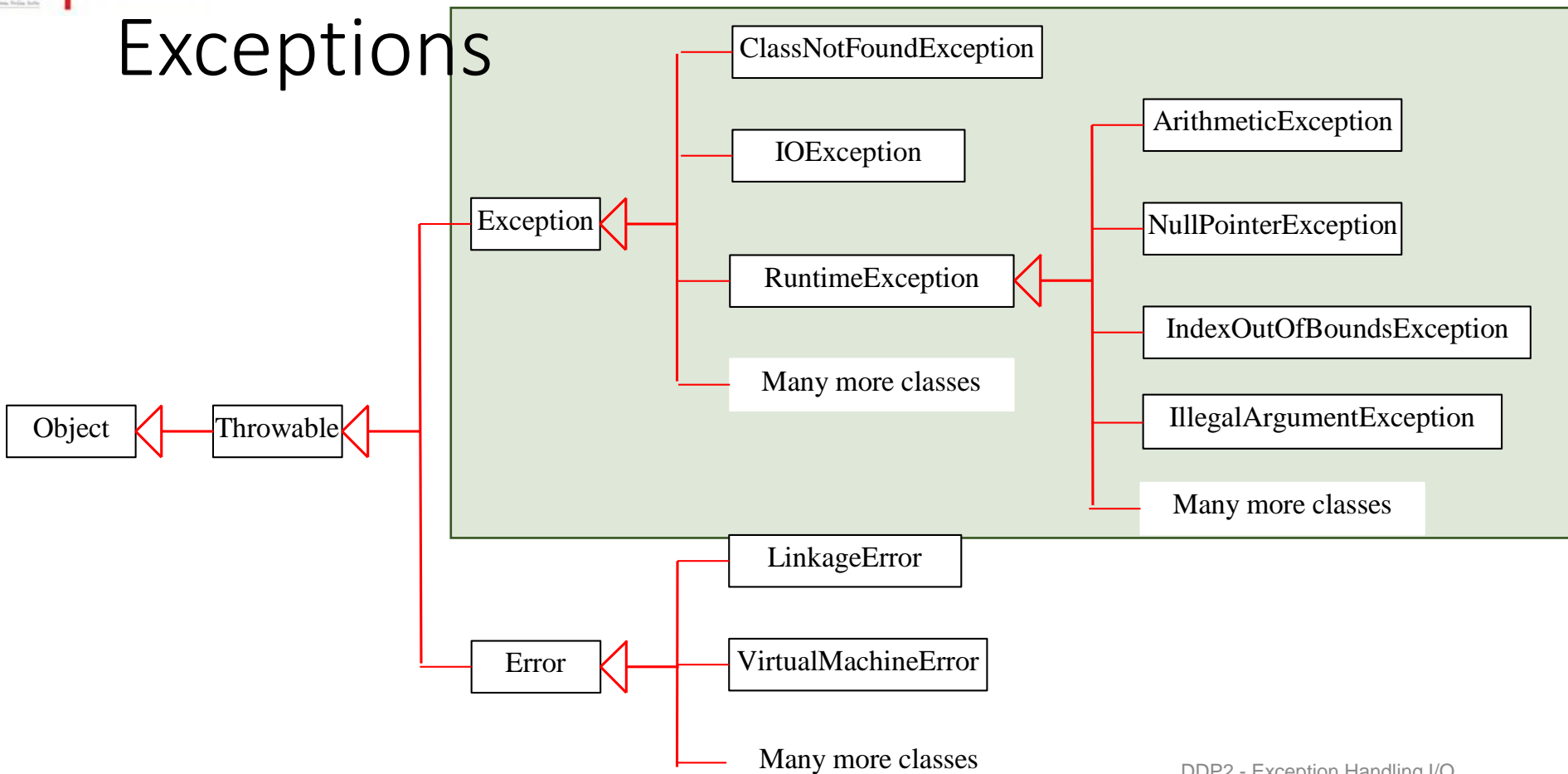
System Errors

- System errors are thrown by JVM and represented in the Error class.
- The Error class describes internal system errors.

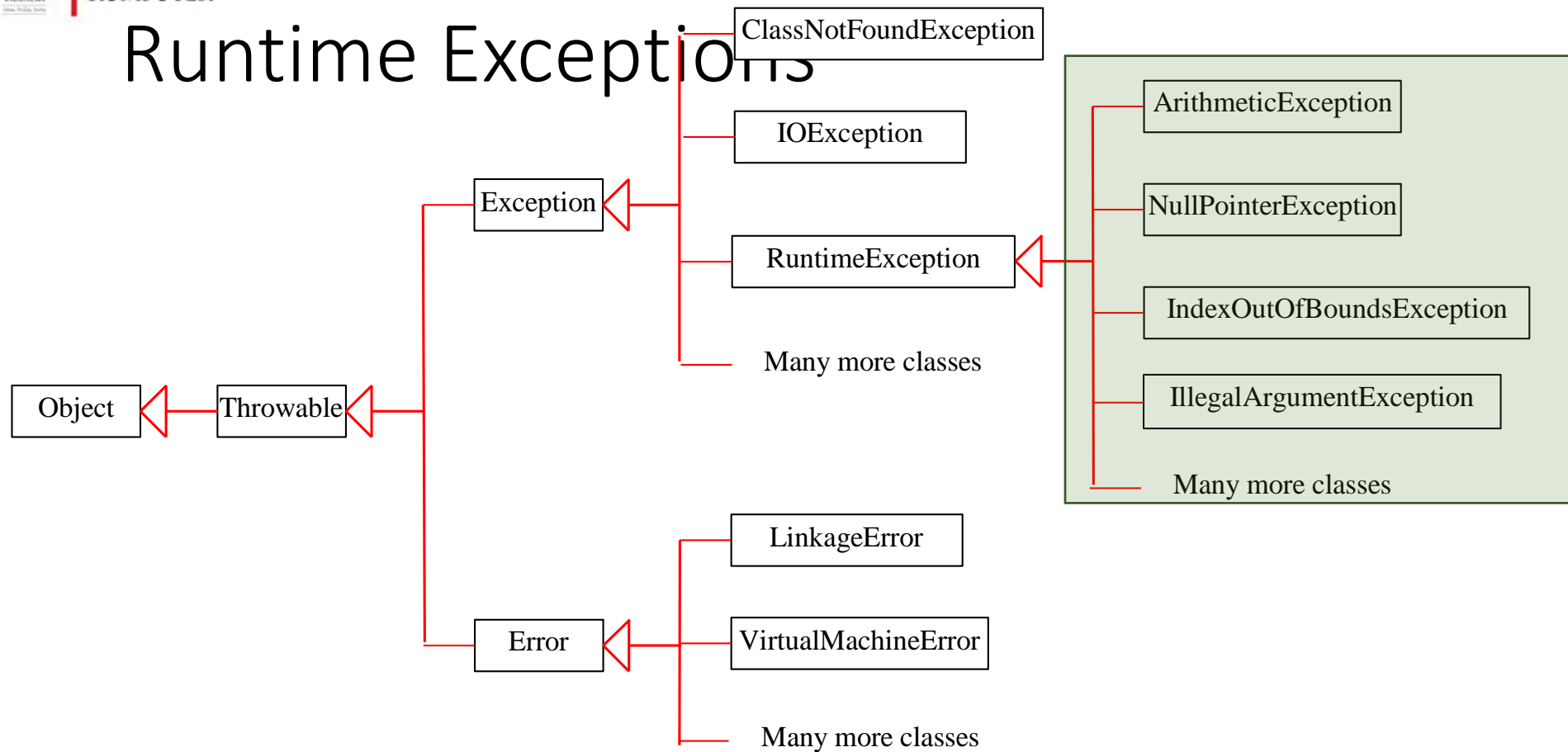
Such errors rarely occur.

If one does, there is little you can do beyond notifying the user and trying to terminate the program gracefully

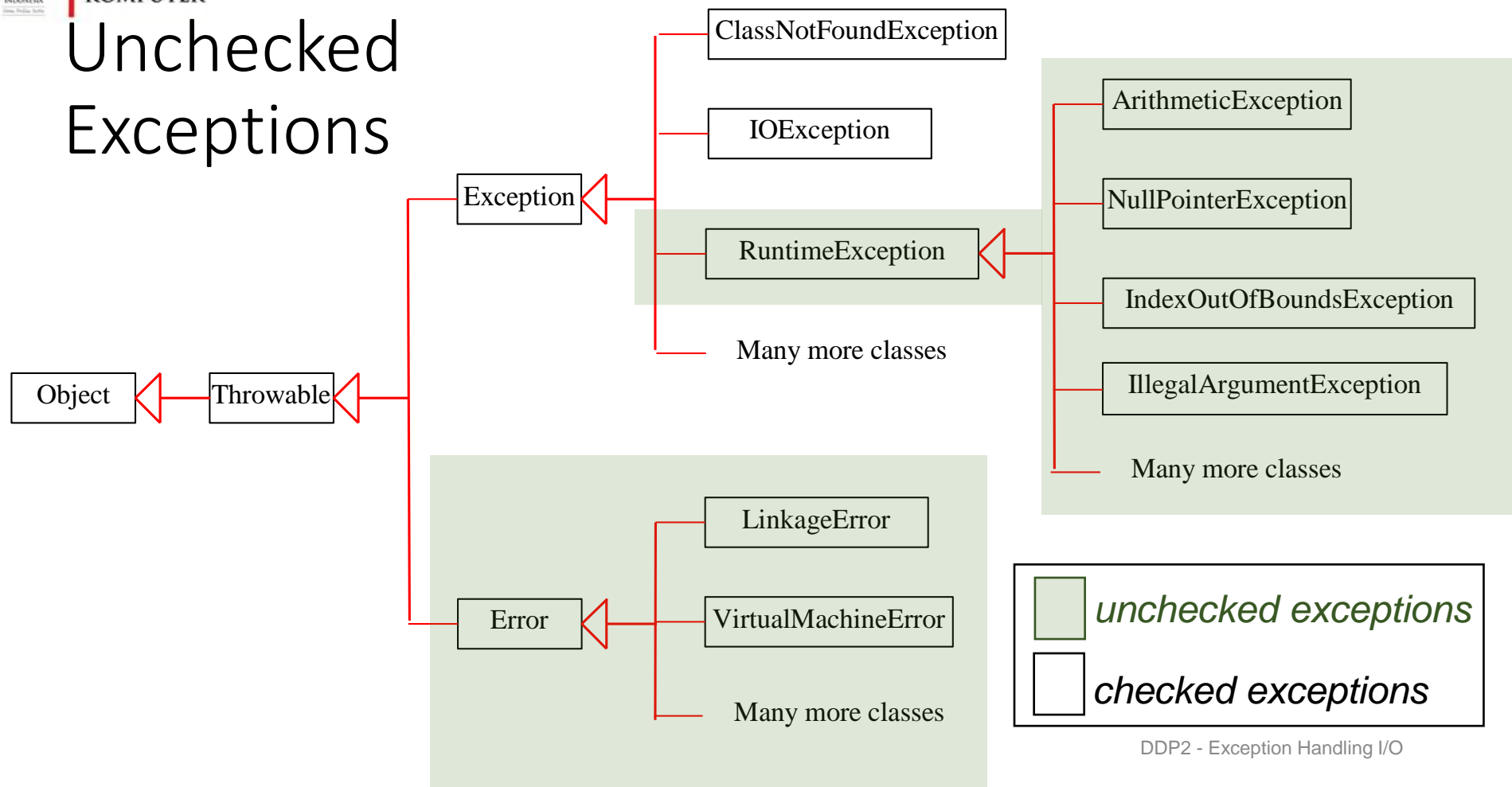
Exceptions



Runtime Exceptions



Checked vs. Unchecked Exceptions



Checked vs. Unchecked Exceptions

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

Unchecked Exceptions

- Programming logic errors that are not recoverable.
- These are the logic errors **that should be corrected** in the program.
- Unchecked exceptions can occur **anywhere** in the program.
- To avoid overuse of try-catch blocks, Java **does not mandate you** to write code to catch unchecked exceptions.

Unchecked Exceptions - Examples

- A `NullPointerException` is thrown if you access an object through a reference variable before an object is assigned to it.
- `IndexOutOfBoundsException` is thrown if you access an element in an array outside the bounds of the array.

Declaring, Throwing, and Catching Exceptions

```
method(){  
    try{  
        method2();  
    }  
    catch(Exception ex){  
        //process exception  
    }  
}
```

Catch Exception

DeclareException

```
method2() throws Exception{  
    if (an error occurs){  
        throw new Exception();  
    }  
}
```

Throw Exception

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.

```
throw new TheException();
```

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

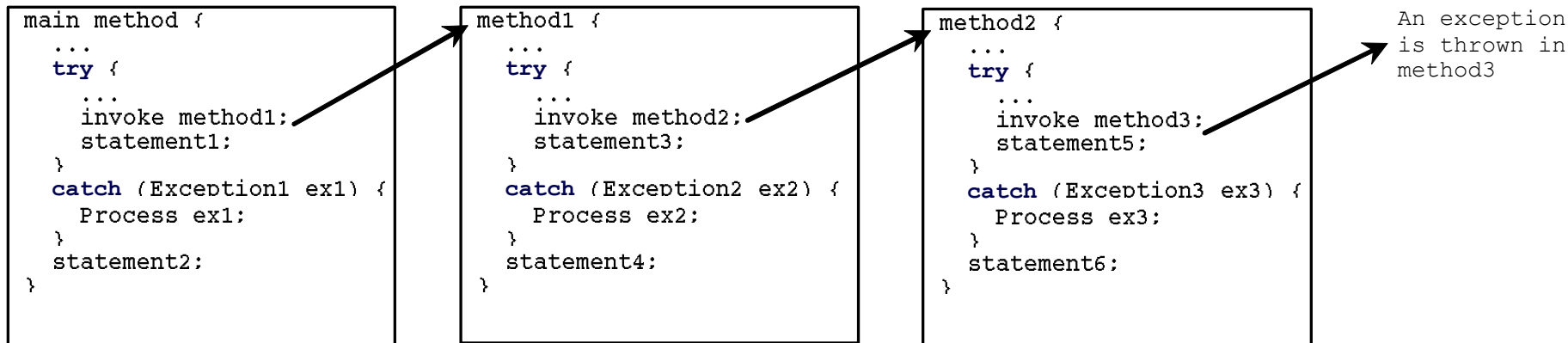
Throwing Exceptions Example

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

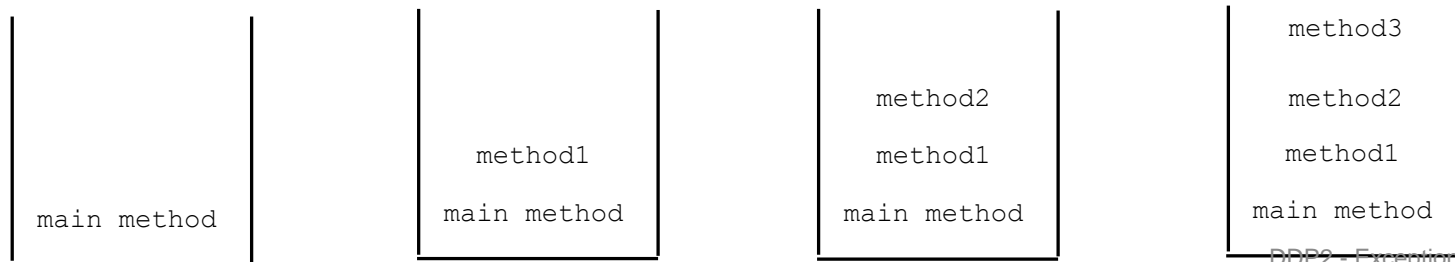
Catching Exceptions

```
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




Catching or Declaring Checked Exceptions

- Check this method


```
void p2() throws IOException{  
    if(a file does not exist){  
        throw new IOException("Files does not exist");  
    }  
}
```

Catching or Declaring Checked Exceptions

- Java **forces you to deal with** checked exceptions.
- If a method declares a checked exception, **you must invoke it in a try-catch block or declare to throw the exception in the calling method.**



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



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

Rethrowing Exceptions

```
try {  
    statements;  
}  
catch(Exception ex) {  
    perform operations before exits;  
    throw ex;  
}
```

The **finally** Clause

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

Caution!

- Exception handling separates error-handling code from normal programming tasks, making programs easier to read and to modify.
- **But** -- 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

- To deal with unexpected error conditions.
- Do **not** use it to deal with simple, expected situations.

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


When to Use Exceptions (2)

- The previous example only needs this:

```
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

```
public class InvalidRadiusException extends Exception {  
    private double radius;  
  
    /** Construct an exception */  
    public InvalidRadiusException(double radius) {  
        super("Invalid radius " + radius);  
        this.radius = radius;  
    }  
  
    /** Return the radius */  
    public double getRadius() {  
        return radius;  
    }  
}
```

Custom Exception Usage Example

```
public class CircleWithRadiusException {
    /** The radius of the circle */
    private double radius;

    /** The number of the objects created */
    private static int numberOfObjects = 0;

    /** Construct a circle with radius 1 */
    public CircleWithRadiusException() {
        this(1.0);
    }

    /** Construct a circle with a specified radius */
    public CircleWithRadiusException(double newRadius) {
        try {
            setRadius(newRadius);
            numberOfObjects++;
        }
        catch (InvalidRadiusException ex) {
            ex.printStackTrace();
        }
    }
}
```

```
/** Return radius */
public double getRadius() {
    return radius;
}

/** Set a new radius */
public void setRadius(double newRadius)
    throws InvalidRadiusException {
    if (newRadius >= 0)
        radius = newRadius;
    else
        throw new InvalidRadiusException(newRadius);
}

/** Return numberOfObjects */
public static int getNumberOfObjects() {
    return numberOfObjects;
}

/** Return the area of this circle */
public double findArea() {
    return radius * radius * 3.14159;
}
}
```

Assertions

- A Java statement that enables you to assert an **assumption** about your program.
- An assertion contains a **Boolean expression** that should be true during program execution.
- Assertions can be used to assure program correctness and avoid logic errors.

Declaring Assertions

- An assertion is declared using the new Java keyword assert in JDK 1.4 as follows:

```
assert assertion;  
or  
assert assertion : detailMessage;
```

- `assertion` is a Boolean expression
- `detailMessage` is a primitive-type or an Object value.

Executing Assertions

- When an assertion statement is executed, Java evaluates the assertion.
- If false, an `AssertionError` will be thrown.
- The `AssertionError` class has a no-arg constructor and seven overloaded single-argument constructors of type `int`, `long`, `float`, `double`, `boolean`, `char`, and `Object`.
- Since `AssertionError` is a subclass of `Error`, when an assertion becomes false, the program displays a message on the console and exits.

Executing Assertions

```
public class AssertionDemo {  
    public static void main(String[] args) {  
        int i; int sum = 0;  
        for (i = 0; i < 10; i++) {  
            sum += i;  
        }  
        assert i == 10;  
        assert sum > 10 && sum < 5 * 10 : "sum is " + sum;  
    }  
}
```


Compiling Programs with Assertions

- Since `assert` is a new Java keyword introduced in JDK 1.4, you have to compile the program using a JDK 1.4 compiler. Furthermore, you need to include the switch `-source 1.4` in the compiler command as follows:

```
javac -source 1.4 AssertionDemo.java
```

- You do not need “`-source 1.4`” in older JDK version
- By default, the assertions are disabled at runtime. To enable it, use the switch `-enableassertions`, or `-ea` for short, as follows:

```
java -ea AssertionDemo
```

- Assertions can be selectively enabled or disabled at class level or package level. The disable switch is `-disableassertions` or `-da` for short

Exception Handling vs Assertions

- **Assertion should not be used to replace exception handling.**
- Exception handling deals with unusual circumstances during program execution.
- Assertions are to assure the correctness of the program.
- Exception handling addresses robustness and assertion addresses correctness.
- Like exception handling, assertions are for internal consistency and validity checks.
- Assertions are checked at runtime and can be turned on or off at startup time.

Note

- **Do not use assertions for argument checking in public methods.**
- Valid arguments that may be passed to a public method are considered to be part of the method's contract.
- The contract must always be obeyed whether assertions are enabled or disabled.
- Check the following:

```
public void setRadius(double newRadius) {  
    assert newRadius >= 0;  
    radius = newRadius;  
}
```

This code in Circle class should be rewritten using exception handling.

Exception Handling vs Assertions (2)

- *Use assertions to reaffirm assumptions.* This gives you more confidence to assure correctness of the program.
- A common use of assertions is to replace assumptions with assertions in the code.
- Another good use of assertions is in a switch statement without a default case.

```
switch (month) {  
    case 1: ... ; break;  
    case 2: ... ; break;  
    ...  
    case 12: ... ; break;  
    default: assert false : "Invalid month: " + month  
}
```

.:END:.