

A vertical decorative strip on the left side of the slide, featuring a close-up of autumn leaves in shades of red, orange, and yellow.

JAVA Exceptions

A vertical decorative strip on the left side of the slide, featuring a close-up of autumn leaves in shades of red, orange, and yellow.

What is an exception?



Errors and Error Handling

- A running (executing) program may encounter some problems resulting to an error.
- An Error is any **unexpected** result obtained from a program during execution.
- These errors if not handled may manifest themselves as incorrect results or behavior, or make the program terminate abnormally .



Errors and Error Handling

- A program that is written with ability to manage errors when they occur is a stable and robust program.
- Therefore a program should be developed with abilities to manage errors where possible.
- The programmer, should keep to prevent errors from reaching the user.



Errors and Error Handling

- Some typical causes of errors:
- **Array errors** (i.e. accessing element -1)
- **Conversion errors** (i.e. convert 'q' to a number)
- **Calculation errors** (i.e. divide by 0)
- **Memory errors** (i.e. memory incorrectly allocated, memory leaks, "null pointer")
- **File system errors** (i.e. disk is full, disk has been removed)
- **Network errors** (i.e. network is down, URL does not exist)

Example: Runtime Errors

```
import java.util.Scanner;

public class ExceptionDemo {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter an integer: ");
        int number = scanner.nextInt();

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

If an exception occurs on this line, the rest of the lines in the method are skipped and the program is terminated.

Terminated.



Exceptions

- What are they?
 - An exception is a representation of an error condition or a situation that is not the expected result of a method or a part of a program.
 - Exceptions are built into the Java language and are available to all program code.
 - Exceptions isolate the code that deals with the error condition from regular program logic.
 - This give the programmer a way to manage errors in case they occur.



Java is strict

- **Unlike C++, Java is quite strict about catching exceptions**
 - **Especially If it is a checked exception**
-
- **Why is this a good idea?**
- **By enforcing exception specifications from top to bottom, Java guarantees exception correctness at compile time.**

A vertical decorative strip on the left side of the slide, featuring a close-up of autumn leaves in shades of red, orange, and yellow.

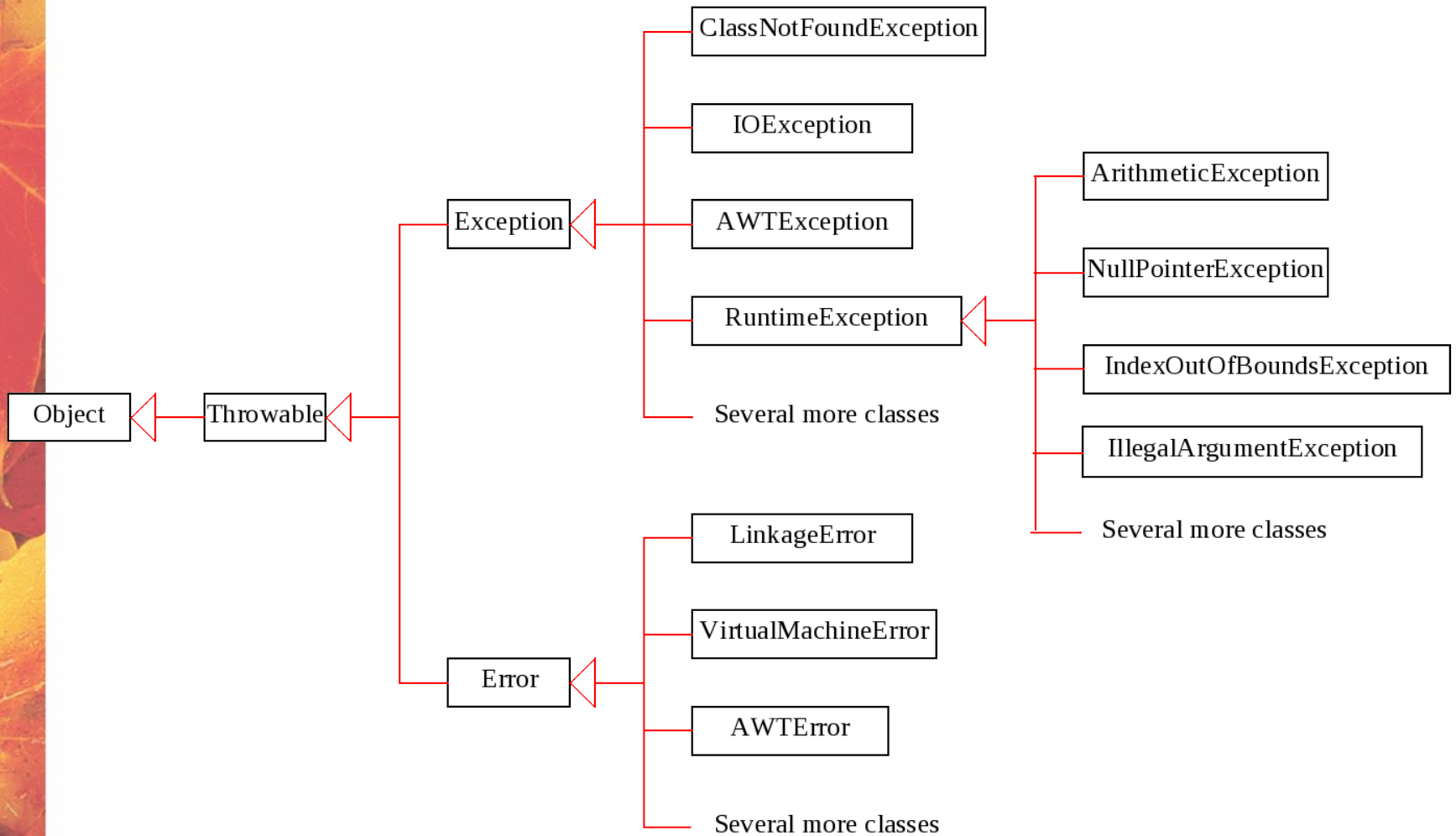
Approaches to handling an exception



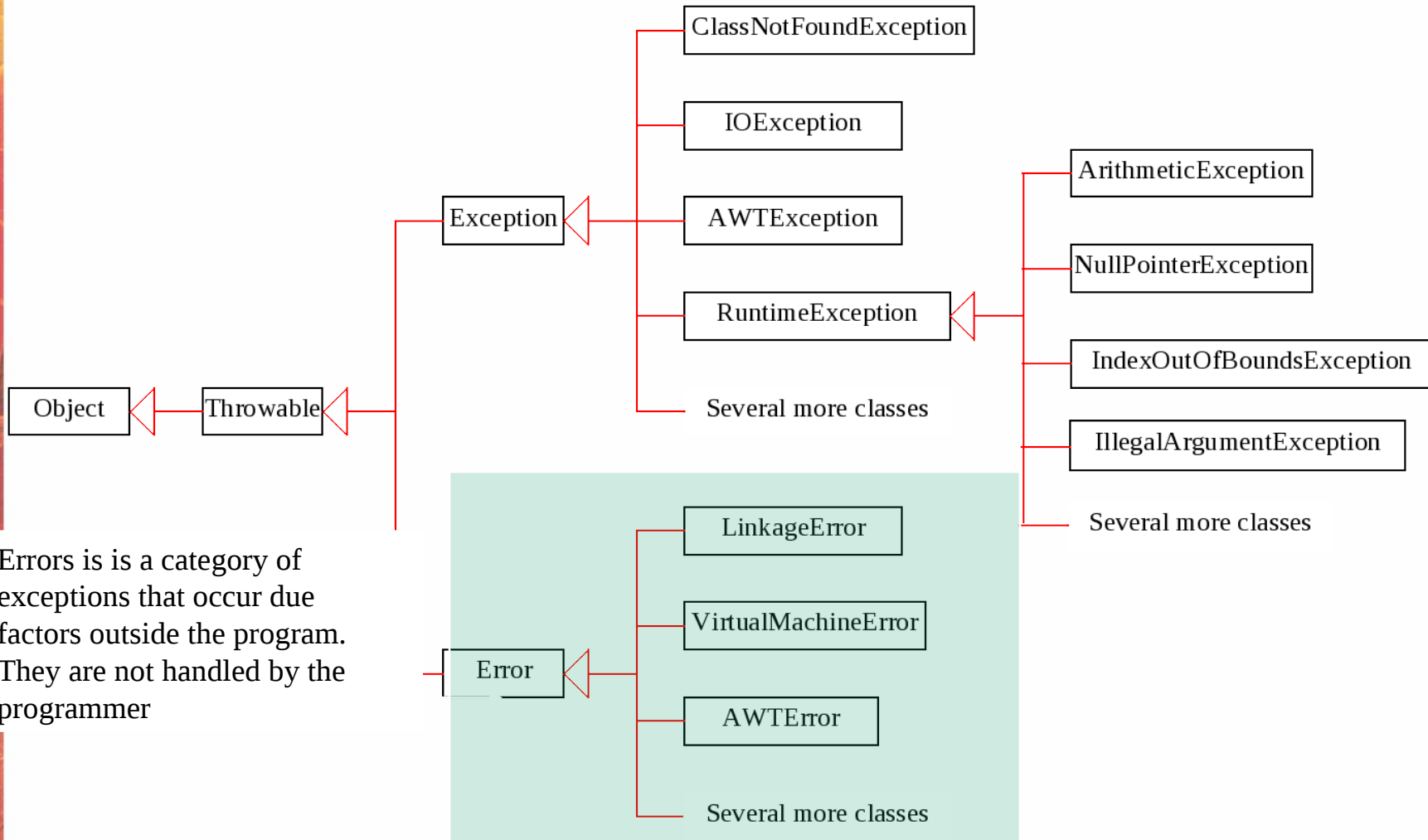
Exceptions

- Exception handling is an important aspect of object-oriented design
- We focus on:
 - the purpose(Why) of exceptions
 - the try-catch statement
 - propagating exceptions
 - The exception class hierarchy
 - Exception Classes
 - Arithmetic Classes
 - Error Classes

Exception Classes

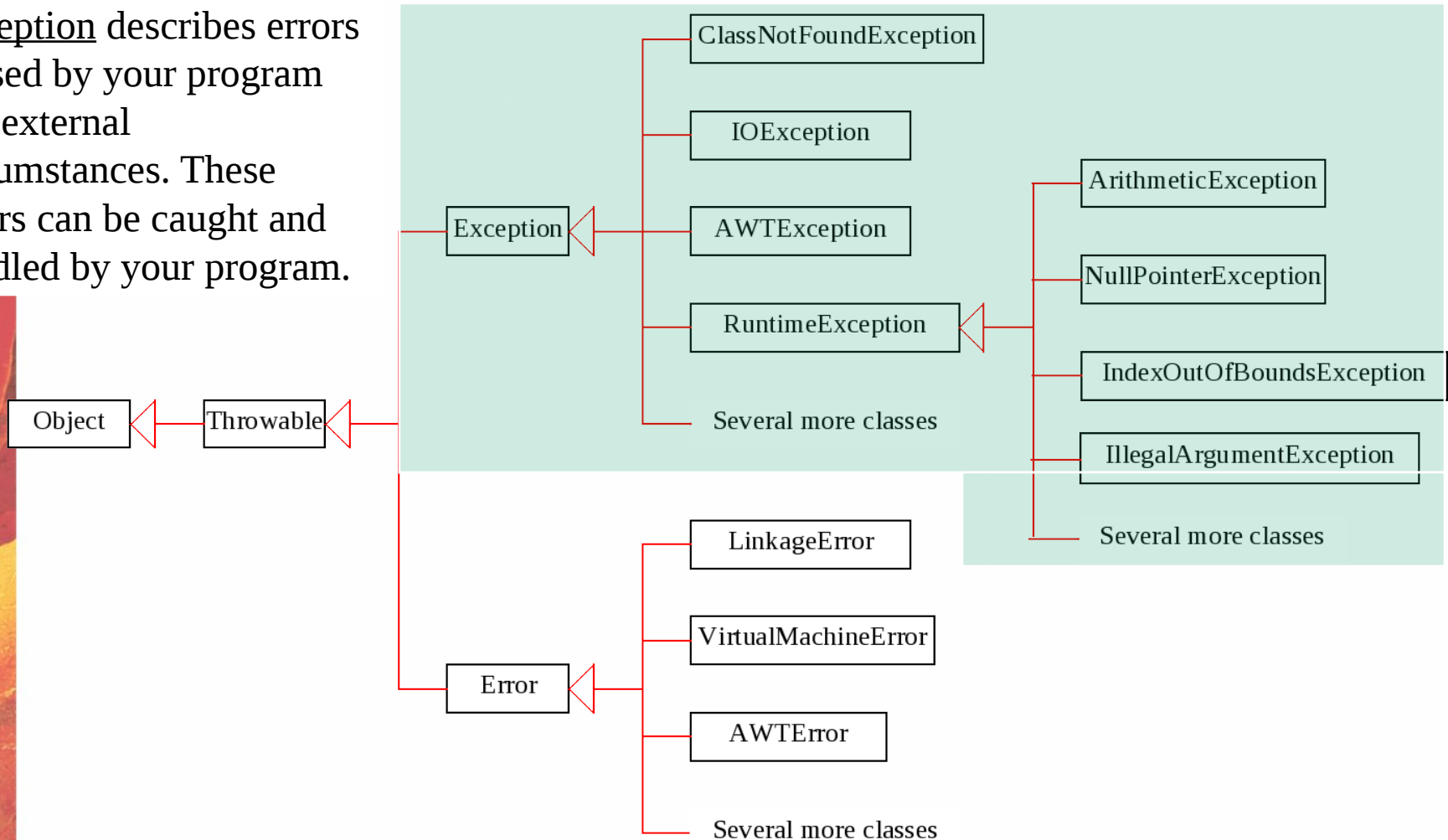


System Errors

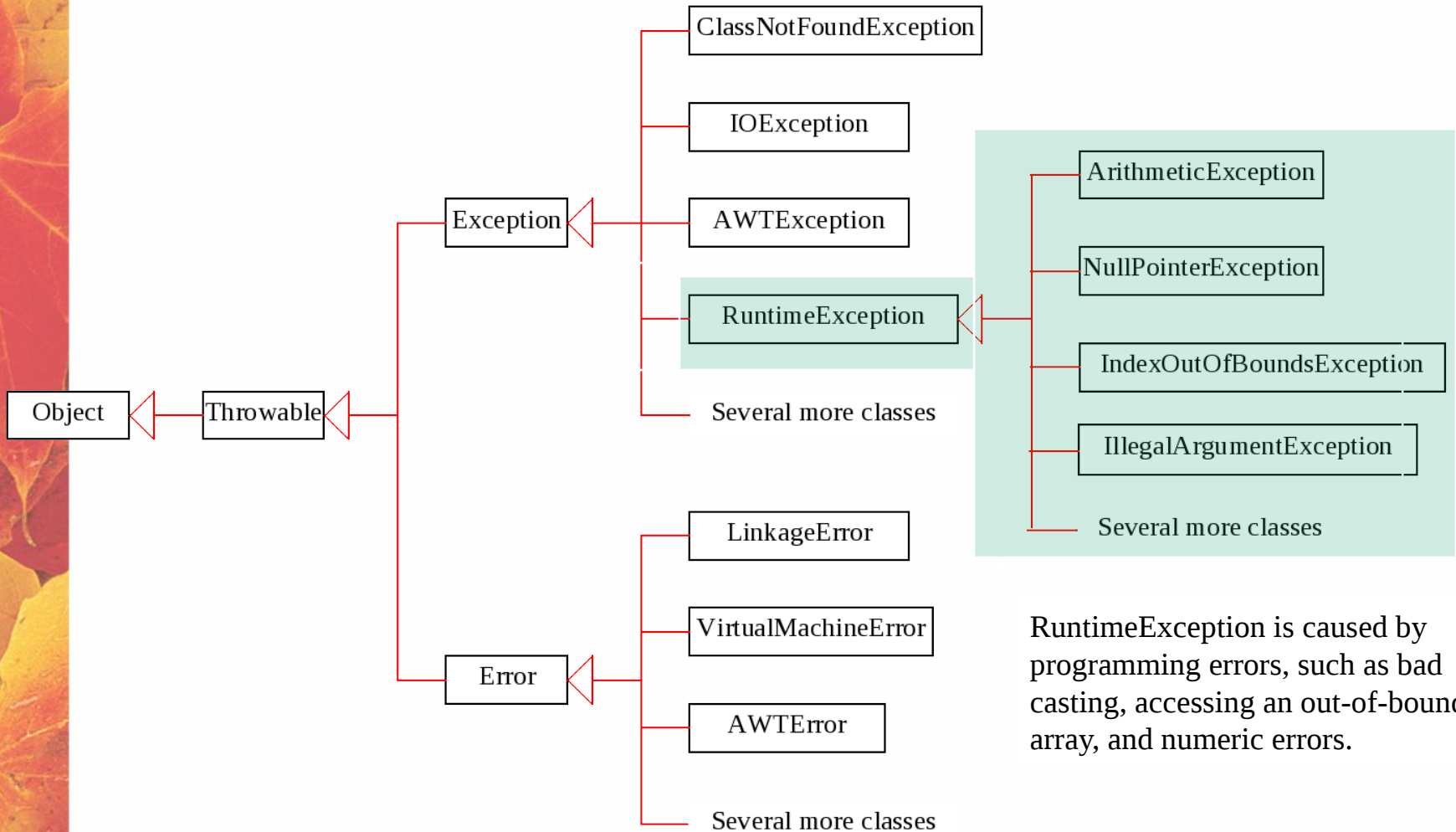


Exceptions

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



Runtime Exceptions



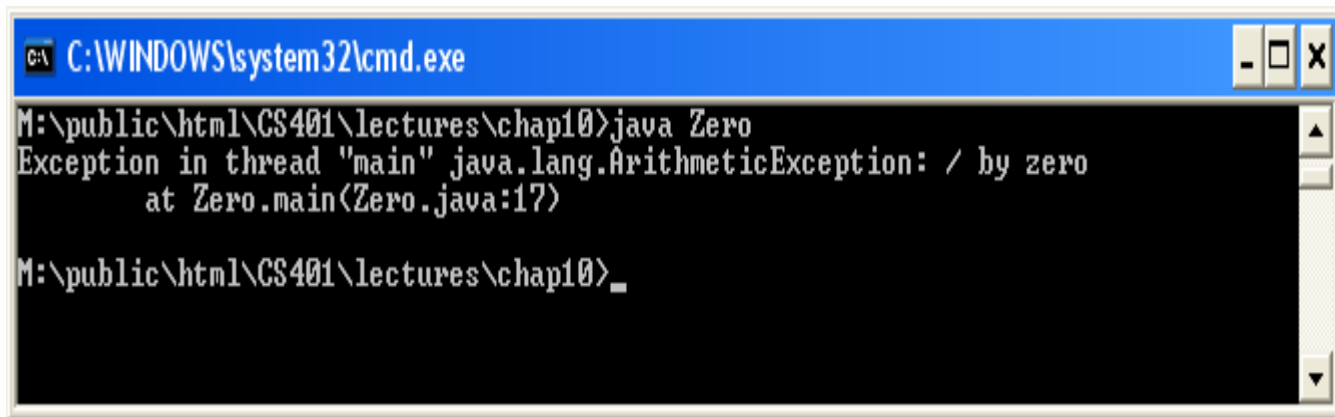


Exception Handling

- **Java has a predefined set of exceptions and errors that can occur during execution**
- **A program can deal with an exception in one of three ways:**
 - **ignore it**
 - **handle it where it occurs**
 - **handle it in another place in the program**
- **The manner in which an exception is processed is an important design consideration**

Exception Handling

- If an exception is ignored by the program, the program will terminate abnormally and produce an appropriate message
- The message includes a *call stack trace* that:
 - indicates the line on which the exception occurred
 - shows the method call trail that lead to the attempted execution of the offending line

A screenshot of a Windows command prompt window. The title bar is blue and contains the text 'C:\WINDOWS\system32\cmd.exe' and standard window control buttons. The command prompt shows the following text:

```
M:\public\html\CS401\lectures\chap10>java Zero  
Exception in thread "main" java.lang.ArithmeticException: / by zero  
    at Zero.main(Zero.java:17)  
  
M:\public\html\CS401\lectures\chap10>_
```




The try Statement

- To handle an exception in a program, the line that throws the exception is executed within a *try block*
- A *try block* is followed by one or more *catch clauses*
- Each *catch clause* has an associated exception type and is called an *exception handler*
- When an exception occurs, processing continues at the first catch clause that matches the exception type



Catching an Exceptions

- Try-catch syntax

```
try {  
... normal program code  
}  
catch(Exception e) {  
... exception handling code  
}
```

Catching an exception

```
try {  
    // statement that could throw an exception  
}  
catch (<exception type> e) {  
    // statements that handle the exception  
}  
catch (<exception type> e) { //e higher in hierarchy  
    // statements that handle the exception  
}  
finally {  
    // release resources  
}  
//other statements
```

- At most one catch block executes
- finally block always executes once, whether there's an error or not

Catch Runtime Errors

```
1      import java.util.*;
2
3      public class HandleExceptionDemo {
4          public static void main(String[] args) {
5              Scanner scanner = new Scanner(System.in);
6              boolean continueInput = true;
7
8              do {
9                  try {
10                     System.out.print("Enter an integer: ");
11                     int number = scanner.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                     scanner.nextLine(); // discard input
23                 }
24             } while (continueInput);
25         }
26     }
```

If an exception occurs on this line,
the rest of lines in the try block are
skipped and the control is
transferred to the catch block.



The finally Clause

- A try statement can have an optional clause following the catch clauses, designated by the reserved word **finally**
- The statements in the finally clause always are executed
- If no exception is generated, the statements in the finally clause are executed after the statements in the try block complete
- If an exception is generated, the statements in the finally clause are executed after the statements in the appropriate catch clause complete



Coding Exceptions

- Using the **throws** keyword
 - If a method contains code that might throw an exception, it should declare the “**throws**” keyword followed by the exception it is expected to throw. This way we avoid handling the exception yourself
 - Example
 - ```
public void myMethod throws IOException {
 ... normal code with some I/O
}
```



# Declaring Exceptions

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

Example

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

# 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");
}
```



# Exceptions are ubiquitous in Java

- Exception handling required for all read methods
- Also many other system methods
- If you use one of Java's built in class methods and it throws an exception, you must catch it (i.e., surround it in a try/catch block) or rethrow it, or you will get a compile time error:

```
char ch;
try { ch = (char) System.in.read(); }
 catch (IOException e)
 { System.err.println(e); return;
```

# Trace a Program Execution

Suppose no  
exceptions in the  
statements

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

Next statement;

# Trace a Program Execution

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

The final block is  
always executed

Next statement;

# Trace a Program Execution

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

Next statement;

Next statement in  
the method is  
executed

# Trace a Program Execution

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

Next statement;

Suppose an exception  
of type Exception1 is  
thrown in statement2

# Trace a Program Execution

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

Next statement;
```

The exception is handled.

# Trace a Program Execution

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

Next statement;

The final block is  
always executed.

# Trace a Program Execution

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

Next statement;

The next statement in the method is now executed.



# Trace a Program Execution

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

statement2 throws an exception of type Exception2.

Next statement;

# 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;
}
```

Execute the final  
block



Next statement;

# 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

# Outline

**Exception Handling**

**The try-catch Statement**



**Exception Classes**

**I/O Exceptions**



# The Exception Class Hierarchy

- Classes that define exceptions are related by inheritance, forming an exception class hierarchy
- All error and exception classes are **descendents** of the **Throwable** class
- A programmer can define an exception by extending the **Exception** class or one of its descendants
- The parent class used depends on how the new exception will be used



# Checked Exceptions

- An *exception* is either *checked* or *unchecked*
- A *checked exception* either must be caught by a method, or must be listed in the *throws clause* of any method that may throw or propagate it
- A *throws clause* is appended to the method header
- The compiler will issue an error if a checked exception is not caught or asserted in a throws clause



# Unchecked Exceptions

- An **unchecked exception** does not require explicit handling, though it could be processed that way
- The only unchecked exceptions in Java are objects of type **RuntimeException** or any of its descendants
- Errors are similar to **RuntimeException** and its descendants in that:
  - Errors should not be caught
  - Errors do not require a throws clause



# The throw Statement

- Exceptions are thrown using the *throws statement*
- It follows the signature of a method
- Usually a *throws statement* is executed inside an if statement that evaluates a condition to see if the exception should be thrown
- Example
  - *Public in t( argument) throws Exception 1...2 {*
  - *}*

# Outline

**Exception Handling**

**The try-catch Statement**

**Exception Classes**



**I/O Exceptions**



# I/O Exceptions

- Let's examine issues related to exceptions and I/O
- A *stream* is a sequence of bytes that flow from a source to a destination
- In a program, we read information from an input stream and write information to an output stream
- A program can manage multiple streams simultaneously

# Standard I/O

- There are three standard I/O streams:
  - *standard output* – defined by `System.out` (**To Monitor**)
  - *standard input* – defined by `System.in` (**from Kbd**)
  - *standard error* – defined by `System.err` (**To Monitor**)
- We use `System.out` when we execute `println` statements
- `System.out` and `System.err` typically represent a particular window on the monitor screen
- `System.in` typically represents keyboard input, which we've used many times with `Scanner` objects



# The IOException Class

- Operations performed by some I/O classes may throw an `IOException`
  - A file might not exist
  - Even if the file exists, a program may not be able to find it
  - The file might not contain the kind of data we expect
- An `IOException` is a checked exception



# Summary

- **We have focused on:**
  - the purpose of exceptions
  - exception messages
  - the try-catch statement
  - propagating exceptions
  - the exception class hierarchy