Exception Handling in Java

- **Exception**
  - Exception is an abnormal condition that arises at run time
  - Event that disrupts the normal flow of the program. It is an object which is thrown at runtime.
- **Exception Handling**
  - Exception Handling is a mechanism to handle runtime errors.
  - Normal flow of the application can be maintained. It is an object which is thrown at runtime.
  - Exception handling done with the **exception object.**

Types of Errors

There are three categories of errors:

- *Syntax errors* - arise because the rules of the language have not been followed. They are detected by the compiler.
- *Runtime errors* - occur while the program is running if the environment detects an operation that is impossible to carry out.
- *Logic errors* - occur when a program doesn't perform the way it was intended to.
- **Types of Exception:**
  - There are mainly two types of exceptions:
    - Checked
    - Unchecked – Eg. error

- The sun microsystem says there are three types of exceptions:
  - Checked Exception - are checked at compile-time.
  - Unchecked Exception - are not checked at compile-time rather they are checked at runtime.
  - Error
- Checked Exception - Classes that extend Throwable class except RuntimeException and Error are known as checked exceptions. Checked Exceptions means that compiler forces the programmer to check and deal with the exceptions. e.g.IOException, SQLException etc.
- Unchecked Exception - Classes that extends RuntimeException, Error and their subclasses are known as unchecked exceptions e.g. ArithmeticException,NullPointerException, ArrayIndexOutOf Bounds

Exception etc.

- Error is irrecoverable should not try to catch. e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Exception Classes

ClassNotFoundException

IOException

ArithmeticException

Object
Throwable

Exception Error

AWTException

RuntimeException Several more classes

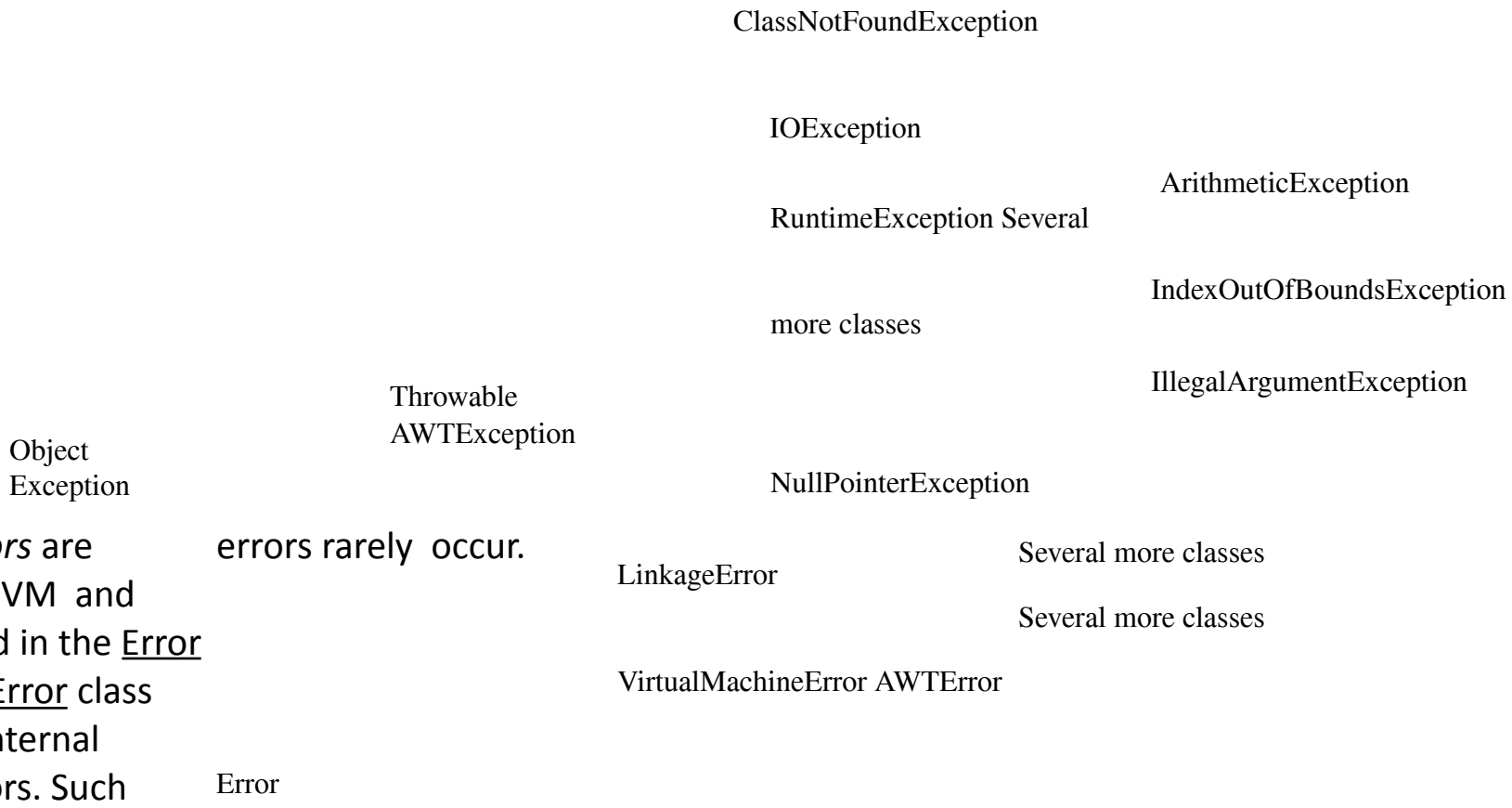
LinkageError

VirtualMachineError AWTError

Several more classes
NullPointerException

IndexOutOfBoundsException IllegalArgumentExceptionSeveral more classes

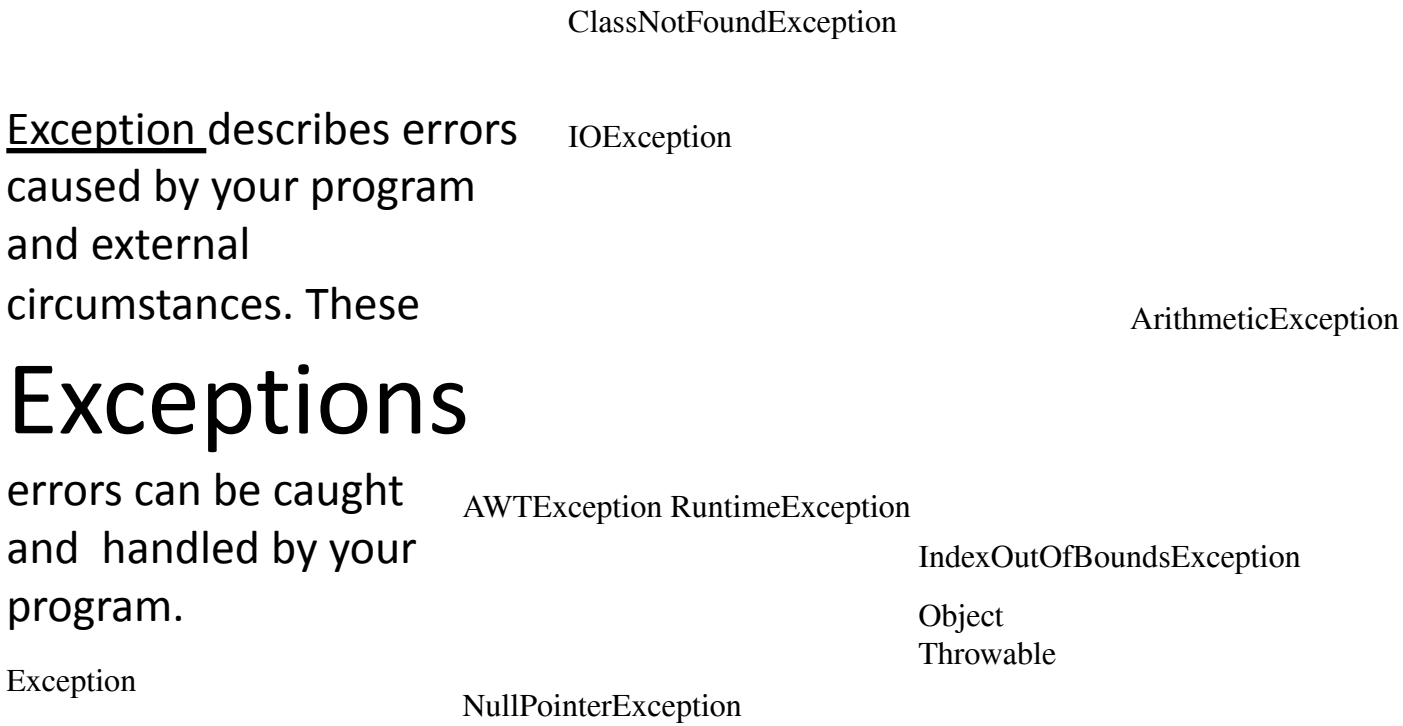
System Errors



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

Exceptions

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



Several more classes

IllegalArgumentException Several

VirtualMachineError AWTError

more classes

LinkageError

Several more classes

Error

Runtime Exceptions

ClassNotFoundException

IOException

ArithmeticException

Object
Throwable
Exception Error

AWTException

RuntimeException Several more classes

LinkageError

VirtualMachineError AWTError

Several more classes
NullPointerException

IndexOutOfBoundsException

IllegalArgumentException

Several more classes

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

Exception Handling Terms

- try – used to enclose a segment of code that may produce a exception
- throw – to generate an exception or to describe an instance of an exception

- catch – placed directly after the try block to handle one or more exception types
- finally – optional statement used after a try-catch block to run a segment of code regardless if a exception is generated

Exceptions –Syntax

```
try
{
  // Code which might throw an exception
}
catch(Exceptionclass object1)
{
  // code to handle an exception
}
catch(Exceptionclass object2)
{
```

```
  // code to handle an exception
}
finally
{
  // ALWAYS executed whether an exception was thrown or not
}
class Simple
{
public static void main(String args[])
{
int data=50/0;
System.out.println("rest of the code...");
}
}
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by zero Rest of the code is not executed (rest of the code..)statement is not printed.

❑ JVM first checks whether the exception is handled or not. ❑ If exception is not handled, JVM provides a default exception handler that performs the following tasks:

- ❑ Prints out exception description.
- ❑ Prints the stack trace (Hierarchy of methods where the exception occurred).
- ❑ Causes the program to terminate.

❑ if exception is handled by the application programmer, normal flow of the application is maintained i.e. rest of the code is executed.

class Simple1

```
{
public static void main(String args[])
{
try
{
int data=50/0;
}
catch(ArithmeticException e)
{
System.out.println(e);
}
System.out.println("rest of the code...");
}}
```

Output:

Exception in thread main java.lang.ArithmeticException:/ by

zero rest of the code…

Multiple catch block:

- If more than one exception can occur, then we use multiple catch blocks
- When an exception is thrown, each **catch** statement is inspected in order, and the first one whose type matches that of the exception is executed
- After one **catch** statement executes, the others are bypassed

Multiple Catch Exceptions –Syntax

```
try
{
  // Code which might throw an exception
}
catch(Exceptionclass object1)
{
  // code to handle an exception
```

```
}
catch(Exceptionclass object2)
{
  // code to handle an exception
}
```

Nested try Statements

- A **try** statement can be inside the block of another try
- Each time a **try** statement is entered, the context of that exception is pushed on the stack
- If an inner **try** statement does not have a catch, then the next **try** statement's catch handlers are inspected for a match
- If a method call within a **try** block has **try** block within it, then then

it is still nested **try**

```
try
{
statement 1;
statement 2;
try
{
statement 1;
statement 2;
}
```

Nested Try Block

```
catch(Exception
e) {
```

```
        }
      }
      catch(Exception
      e) {
      }
class Excep6
{
public static void main(String
args[]) {
try
{
try
{
S.o.p("going to divide");  int b =39/0;
}
catch(ArithmeticException e) {
System.out.println(e); }
try
{
int a[]=new int[5];
a[5]=4;
}
catch(ArrayIndexOutOfBoundsExcept
ion e) {
System.out.println(e);
}
```

```
System.out.println("other statement");  }
catch(Exception e)
{
System.out.println("handeled");
}
System.out.println("normal flow..");
}
}
```

❑ **Finally block**

- ❑ is a block that is always executed.

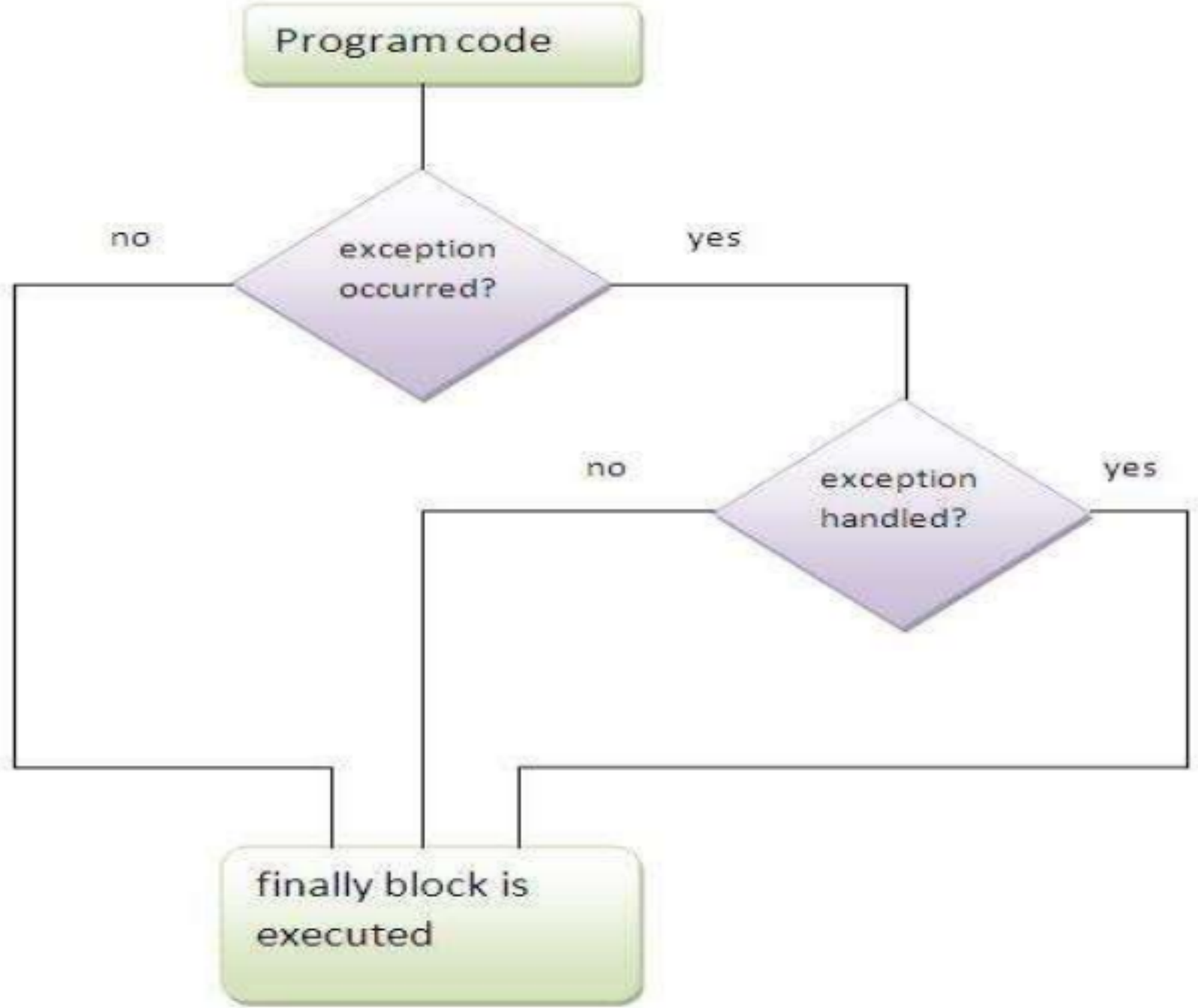
❑ To perform some important tasks such as closing connection, stream etc.

❑ Used to put "cleanup" code such as closing a file, closing connection etc.

❑ Finally creates a block of code that will be executed after a try/catch block has completed

❑ Finally block will be executed whether or not an exception is thrown. ❑ Each try clause requires at least one catch or finally clause. ❑ **Note:** Before terminating the program, JVM executes finally block(if any). ❑ **Note:** finally

must be followed by try or catch block.


Program code
exception occurred?
no
yes
exception handled?
no
yes
finally block is executed

```
class Simple
{
public static void main(String args[])
{
try
{
int data=25/0;
System.out.println(data);
}
catch(ArithmeticException e)
{
System.out.println(e);
}
finally
{
System.out.println("finally block is always
executed"); }
System.out.println("rest of the code...");
}}
```

- **throw keyword**
  - keyword is used to explictily throw an exception / custom exception.

    throw new ExceptionName("Error Message");
  - Throw either checked or uncheked exception.

    throw new *ThrowableInstance*
  - *ThrowableInstance* must be an object of type **Throwable /** subclass **Throwable**
  - There are two ways to obtain a **Throwable** objects:
    - Using a parameter into a catch clause
    - Creating one with the **new** operator

```
public class bank
{
public static void main(String args[])
{
```

```
int balance = 100, withdraw = 1000;
if(balance < withdraw)
{
//ArithmeticException e = new ArithmeticException("No money please");
//throw e;
//throw new ArithmeticException();
throw new ArithmeticException("No Money");
}
else
{
System.out.println("Draw & enjoy Sir, Best wishes of the day");
}
}
}
 import java.io.*;
 public class Example
 {
 public static void main(String args[]) throws
 IOException {
```

```
DataInputStream dis=new
DataInputStream(System.in); int x =
Integer.parseInt(dis.readLine());
if(x < 0)
{
throw new IllegalArgumentException();
throw new IllegalArgumentException
("You have entered no"+" "+ x +" "+ "which is less than
0"); }
else
{
System.out.println("The no is"+x);
}
}
}
```

throws

- If a method is capable of causing an exception that it does not

handle, it must specify this behavior so that callers of the method can guard themselves against that exception

type method-name parameter-list) **throws** ***exception-list***

{

// body of method

}

- It is not applicable for **Error** or **RuntimeException,** or any of their subclasses

unchecked Exception: under your control so correct your code. **error:** beyond your control e.g. you are unable to do anything E.g: VirtualMachineError or StackOverflowError.

throw keyword	**throws keyword**
throw is used to explicitly throw an exception.	throws is used to declare an exception.

checked exception can not be propagated without throws.

checked exception can be propagated with throws.

throw is followed by an instance.

throws is followed by class.

throw is used within the method.

throws is used with the method signature.

You cannot throw multiple exception

You can declare multiple exception e.g. IOException,SQLException.

public void method()throws

Create our Own Exception:

```
class NumberRangeException extends
Exception {
   String msg;

    NumberRangeException()
    {
```

```
        msg = new String("Enter a number between 20 and
      100"); }
   }
public class My_Exception
{
    public static void main (String args [ ])
    {
       try
       {
          int x = 10;

          if (x < 20 || x >100) throw new NumberRangeException(
       ); }
       catch (NumberRangeException e)
       {
          System.out.println (e);
       }
    }
}
```