Exception Handling

- An *exception* is an abnormal condition that arises in a code sequence at run time.
- Errors → Compile time, Run time
- Example
  - Division by zero
  - Array index not in range
  - Operation on NULL values
  - File not found

Why an exception occurs?

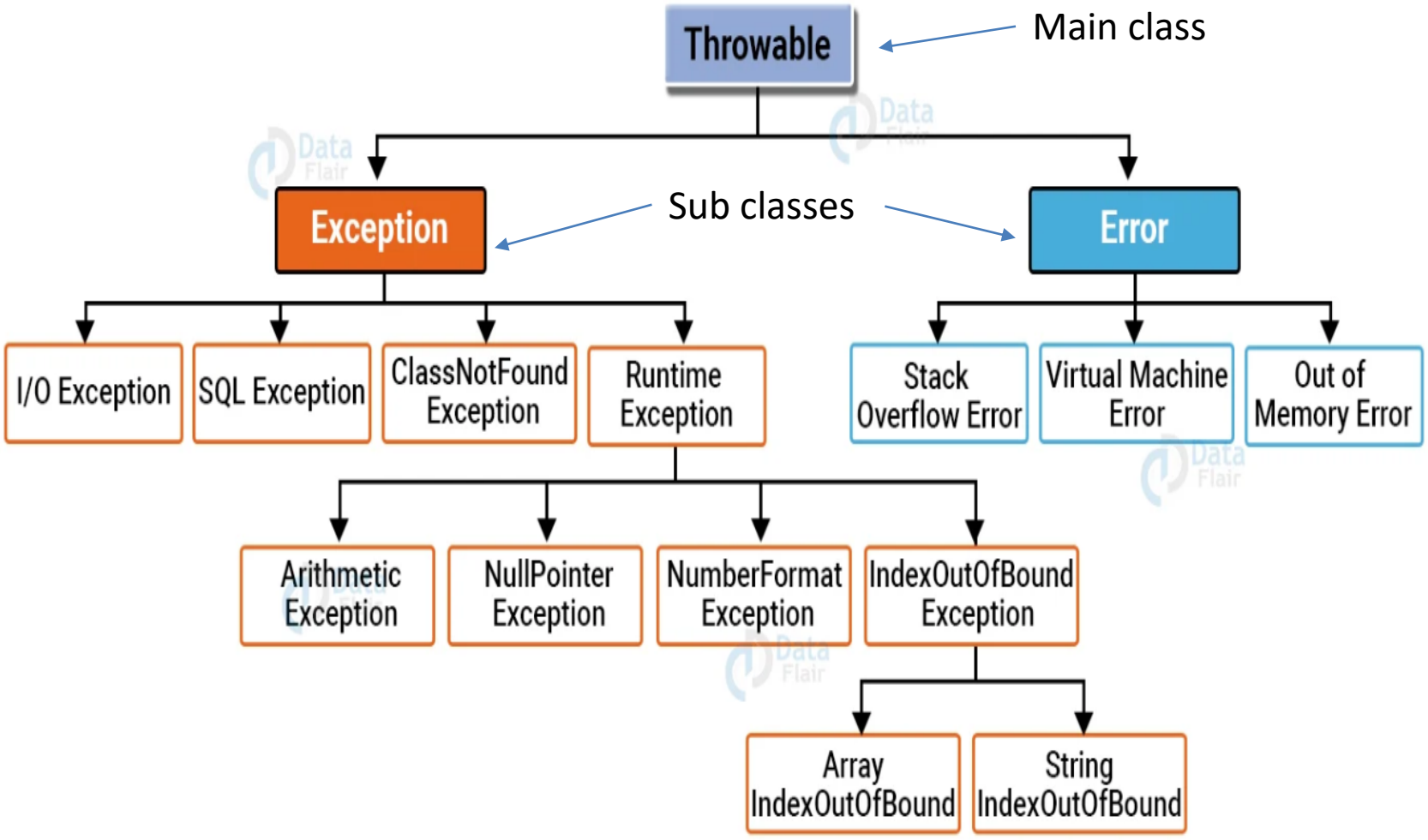
It can occur for various reasons say-

- A user has entered an invalid data
- File not found
- A network connection has been lost in the middle of communications
- The JVM has run out of a memory
- Data provided is not in expected format(eg. int instead of String).
- DB cannot be connected.
- An object is null.

Need for Exception Handling

- To make a program to handle illegal inputs and other unexpected situations
- Difficult to predict all run time errors at the beginning of the program.
- Allows for the program to anticipate and recover from errors, thus making the program more robust and resistant to unexpected conditions.

Exception Types



Types of Java Exceptions

two types of exceptions:

- Checked Exception
- Unchecked Exception
- Error - unchecked exception.

Difference between Checked and Unchecked Exceptions

1) **Checked Exception classes**

The classes which **directly inherit Throwable class** except RuntimeException and Error are known as checked exceptions e.g. IOException, SQLException etc. **Checked exceptions are checked at compile-time.**

2) **Unchecked Exception classes**

The classes which inherit RuntimeException are known as unchecked exceptions e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException etc. **Unchecked exceptions are not checked at compile-time, but they are checked at runtime**.

3) **Error class**

Error is **irrecoverable** e.g. OutOfMemoryError, VirtualMachineError, AssertionError etc.

Error vs Exception

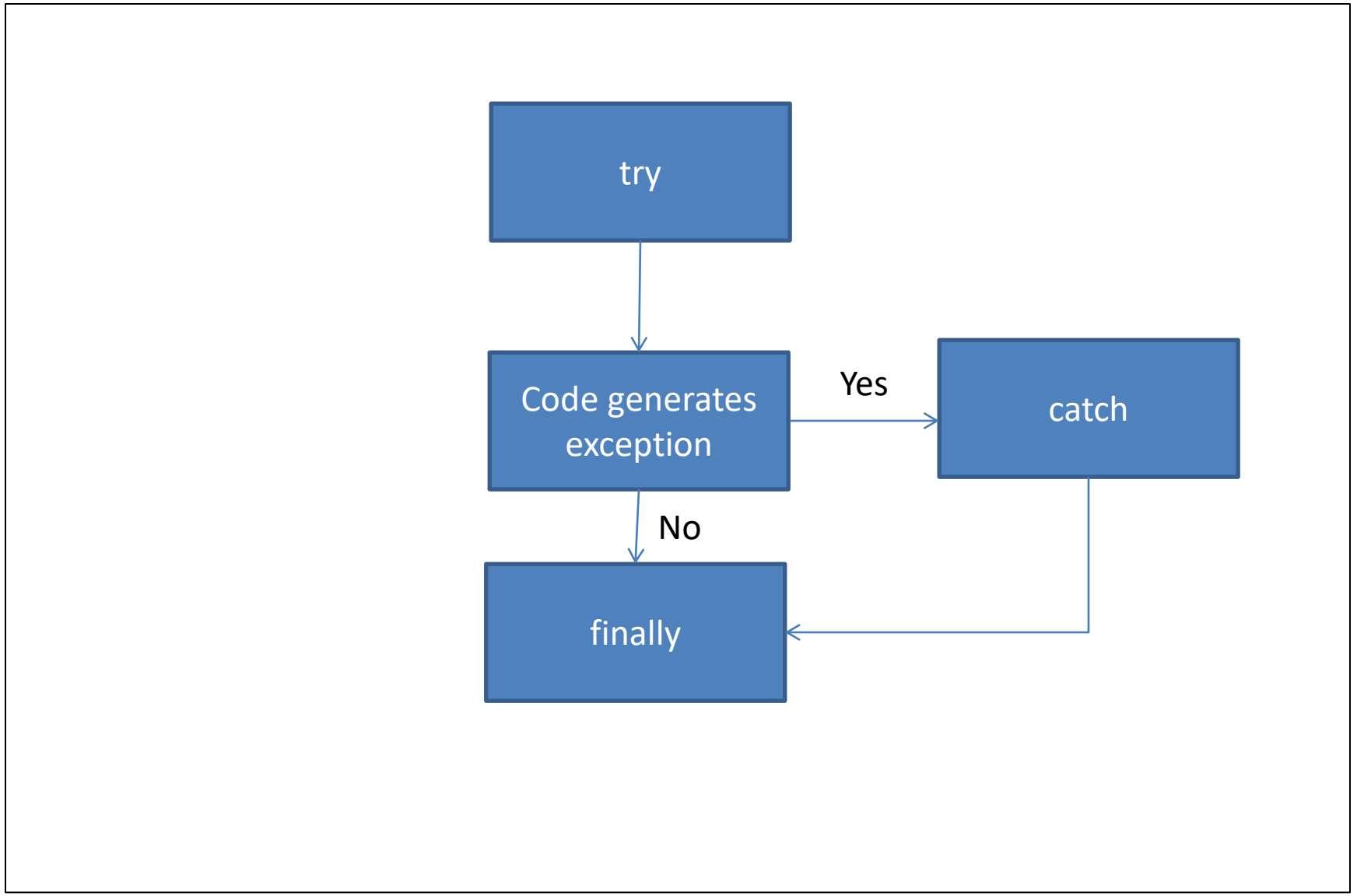
Errors	Exceptions
Errors in java are of type java.lang.Error.	Exceptions in java are of type java.lang.Exception.
All errors in java are unchecked type.	Exceptions include both checked as well as unchecked type.
Errors happen at run time. They will not be known to compiler.	Checked exceptions are known to compiler where as unchecked exceptions are not known to compiler because they occur at run time.
It is impossible to recover from errors.	You can recover from exceptions by handling them through try-catch blocks.
Errors are mostly caused by the environment in which application is running.	Exceptions are mainly caused by the application itself.
Examples : java.lang.StackOverflowError, java.lang.OutOfMemoryError	Examples : Checked Exceptions : SQLException, IOException Unchecked Exceptions : ArrayIndexOutOfBoundException, ClassCastException, NullPointerException

Java Exception Keywords

Keyword	Description
try	The "try" keyword is used **to specify a block where we should place exception code.** The try block must be **followed by either catch or finally**. It means, we can't use try block alone.
catch	The "catch" block is used **to handle the exception**. It must be preceded by try block which means we can't use catch block alone. It can be followed by finally block later.
finally	The "finally" block is used **to execute the important code of the program.** It is executed whether an exception is handled or not.
throw	The "throw" keyword is used **to throw an exception**.
throws	The "throws" keyword **is used to declare exceptions**. It doesn't throw an exception. It specifies that there may occur an exception in the method. It is always used with method signature.

General Form

```
try
{ // block of code to monitor for errors
}
catch (ExceptionType1 exOb)
{ // exception handler for ExceptionType1
}
finally
{ // block of code to be executed after try block ends
 }
```


try
Code generates exception
Yes
catch
No
finally

EXAMPLE

```
public class Main
{
        public static void main(String[] args)
        {
        try
        {
        int d = 0;
        int a = 42 / d;
        }
        catch(ArithmeticException e)
        {
        System.out.println("Division by zero.");
        }
}}
```

Java Exception Handling Example

```
public class JavaExceptionExample{
  public static void main(String args[]){
   try{
      //code that may raise exception
      int data=100/0;
   }catch(ArithmeticException e)
     {System.out.println(e);}
   //rest code of the program
   System.out.println("rest of the code...");
  }
}
```

Output:

```
Exception in thread main java.lang.ArithmeticException:/ by zero
rest of the code...
```

2) A scenario where NullPointerException occurs

- If we have a null value in any variable, performing any operation on the variable throws a NullPointerException.

- String s=null;
- System.out.println(s.length());//NullPointerException

3) A scenario where NumberFormatException occurs

- The wrong formatting of any value may occur NumberFormatException. Suppose I have a string variable that has characters, converting this variable into digit will occur NumberFormatException.

- String s="abc";
- int i=Integer.parseInt(s);//NumberFormatException

4) A scenario where ArrayIndexOutOfBoundsException occurs

- If you are inserting any value in the wrong index, it would result in ArrayIndexOutOfBoundsException as shown below:

- int a[]=new int[5];
- a[10]=50; //ArrayIndexOutOfBoundsException

Uncaught Exception

```
import java.util.*;
class Main {
public static void main(String args[]) {
try {
        int a []= {5,8,4,1,0};
        int x = 2/a[5];}
catch (ArithmeticException e) {
        System.out.println("Division by zero); }}
        }
```

```
import java.util.*;
class Main {
public static void main(String args[]) {
try {
        int a []= {5,8,4,1,0};
        int x = 2/a[4];    }
catch (ArrayIndexOutOfBoundsException e) {
        System.out.println("array index is not within
        bound.");     }
}}
```

MULTIPLE CATCH STATEMENT

```
import java.util.*;
class Main {
public static void main(String args[]) {
try {
int a []= {5,8,4,1,0};
//int x = 2/a[5];
int y= 2/a[4];
}
catch (ArrayIndexOutOfBoundsException e)  {
        System.out.println("array index is not within bound.");
}
catch(ArithmeticException e1)  {
        System.out.println("Division by Zero");
}
}}
```

OUTPUT??

```
class Main {
public static void main(String args[]) {
try {
int a = 0; int b = 42 / a; }
catch(Exception e) {
System.out.println("Generic Exception catch."); }
catch(ArithmeticException e) {
System.out.println("This is never reached."); }
}
}
```

OUTPUT

Compile time Error

Nested try Statements

- A try statement can be inside the block of another try.

```
class Main {
public static void main(String
args[]) {
try {
int a = args.length;
int b = 42 / a;
System.out.println("a = " + a);
try {
if(a==1) a = a/(a-a);
if(a==2) {
int c[] = { 1 };
c[42] = 99;
}
}
catch(ArrayIndexOutOfBoundsExc
eption e) {
System.out.println("Array index
out-of-bounds: " + e);
}
}
catch(ArithmeticException e) {
System.out.println("Divide by 0: "
+ e);
}
}
}
```

Whenever a try block does not have a catch block for a particular exception, then the catch blocks of parent try block are inspected for that exception, and if a match is found then that catch block is executed.

throw

- It is possible for your program to throw an exception explicitly, using the **throw** statement.
- General form
  - throw *ThrowableInstance*;
- Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.
- Two ways you can obtain a **Throwable** object:
  using a parameter in a **catch** clause or
  creating one with the **new** operator

EXAMPLE

```
class Main {
public static void main(String args[]) {
try {
throw new NullPointerException("demo");
}
catch(NullPointerException e) {
System.out.println("Caught inside demoproc."+ e);
throw e; // rethrow the exception
}
catch(Exception e) {
System.out.println("Recaught: " + e);
}
}
}
```

OUTPUT

```
Caught inside demoproc.java.lang.NullPointerException: demo
Exception in thread "main" java.lang.NullPointerException: demo
        at Main.main(Main.java:4)
```

throws

- A **throws** clause lists the types of exceptions that a method might throw.
- This is the general form of a method declaration that includes a **throws** clause:
- *type method-name*(*parameter-list*) throws *exception-list*
- {
- // body of method
- }

EXAMPLE

```
class ThrowsDemo {
static void throwOne() throws IllegalAccessException {
System.out.println("Inside throwOne.");
throw new IllegalAccessException("demo");
}
public static void main(String args[]) {
try {
throwOne();
} catch (IllegalAccessException e) {
System.out.println("Caught " + e);
}
}
}
```

finally

- **finally** creates a block of code that will be executed after a **try /catch** block has completed and before the code following the **try/catch** block.
- If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception.
- The **finally** clause is optional
- Each **try** statement requires at least one **catch** or a **finally** clause.

EXAMPLE

```
class Main {
static void procA() {
try {
System.out.println("inside procA");
throw new RuntimeException("demo");
}
finally {
System.out.println("procA's finally");
}
}
public static void main(String args[]) {
try {
procA();
}
catch (Exception e) {
System.out.println("Exception caught");
}
}
}
```

Java's Built-in Exceptions

- *Unchecked exceptions* → The compiler does not check to see if a method handles or throws these exceptions
- *Checked exceptions* → *E*xceptions that are checked at compile time.

Exception	Meaning
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Table 10-1 Java's Unchecked **RuntimeException** Subclasses Defined in **java.lang**

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the **Cloneable** interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.
ReflectiveOperationException	Superclass of reflection-related exceptions.

Table 10-2 Java's Checked Exceptions Defined in **java.lang**

Creating Own Exception

```
class MyException extends Exception {
private int detail;
MyException(int a) {
detail = a;
}
public String toString() {
return "MyException[" + detail + "]";
}
}
class Main {
static void compute(int a) throws
MyException {
System.out.println("Called compute(" + a +
")");
if(a > 10)
throw new MyException(a);
System.out.println("Normal exit");
}
public static void main(String args[]) {
try {
compute(1);
compute(20);
}
catch (MyException e) {
System.out.println("Caught " + e);
}
}
}
```

NegativeValueException

```
import java.util.Scanner;
class NegativeValueException extends Exception {
    public NegativeValueException(String message) {
        super(message);
    }
}
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.print("Enter a positive value: ");
          int value = scanner.nextInt();
            if (value < 0) {
                throw new NegativeValueException("Negative values not
                allowed!");
            }
            System.out.println("Entered value: " + value); }
        catch (NegativeValueException e) {
                System.out.println("Exception: " + e.getMessage());
            }
        finally {
                scanner.close();
            }
        }
}
```

Greater Value Exception

```
import java.util.Scanner;
class GreaterValueException extends Exception {
    public GreaterValueException(String message) {
        super(message);
    }
}
public class Main {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        try {
            System.out.print("Enter a value: ");
            int value = scanner.nextInt();
            if (value > 10) {
                throw new GreaterValueException("Value
greater than 10 are not allowed!");
            }
            else{
            System.out.println("Result: " + value*10); }
        }
 catch (GreaterValueException e) {
            System.out.println("Exception: " +
e.getMessage());
        }

   finally {
            scanner.close();
        }
    }
}
```