

Exceptions: An OO Way for Handling Errors

Introduction

- Rarely does a program runs successfully at its very first attempt.
- It is common to make mistakes while developing as well as typing a program.
- Such mistakes are categorised as:
 - syntax errors - compilation errors.
 - semantic errors– leads to programs producing unexpected outputs.
 - runtime errors – most often lead to abnormal termination of programs or even cause the system to crash.

Types of Errors:

1. Compile-Time Errors

- ❖ Due to typing error..
`;'expected
"undefined variable"
- ❖ Compiler cannot create **.class** files

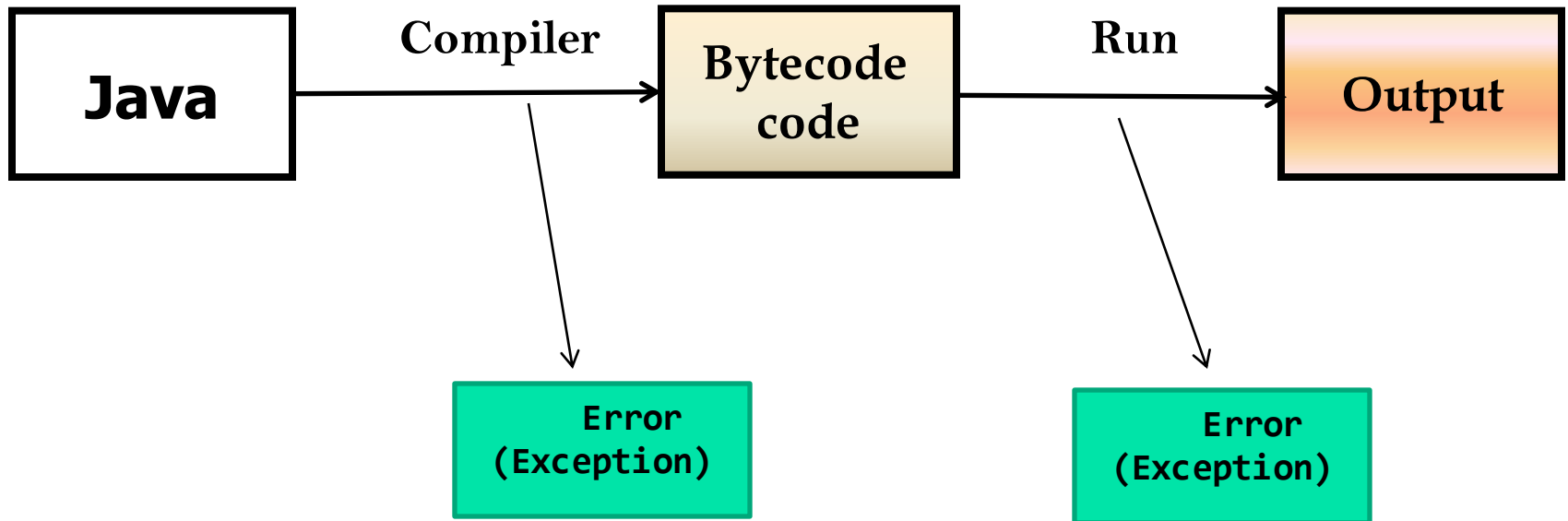
2. Run-Time Errors are thrown as exception

- ❖ may produce wrong result
- ❖ may terminate due to errors such as stack overflow
- ❖ Compiler successfully create **.class** files



- ❖ Dividing an integer by zero
- ❖ Accessing an element that is out of bounds of an array....etc.,

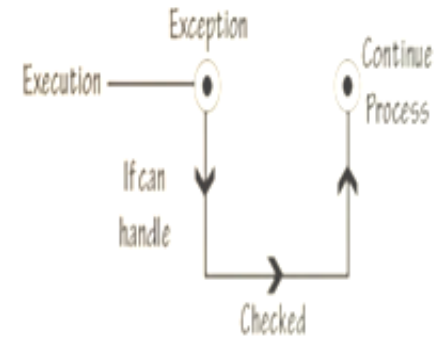
Programming Execution



Exception

- *Exception is a condition that is caused by a run-time error in the program*
- When java interpreter encounters an error , it creates an exception object and throws it
- If we want to continue with the execution of remaining code, then we should try to catch the exception object thrown by the error condition and then display an appropriate message for taking corrective actions. This task is known as ***exception handling***

Exception Types



➤ Checked Exception(handled)

- SQLException, IOException, ClassNotFoundException
- Java app connected with outer resource
- Checked by compiler

➤ Unchecked Exception(handle/not)

- ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException
- Java app not connected with outer resource
- Not checked by compiler

➤ Errors

- Exceptions that are hard enough to recover
- Errors can be called unchecked exception
- Program may not recover –users work save
- We cannot handle(JVM Error,Ram no memory)

Common Runtime Errors

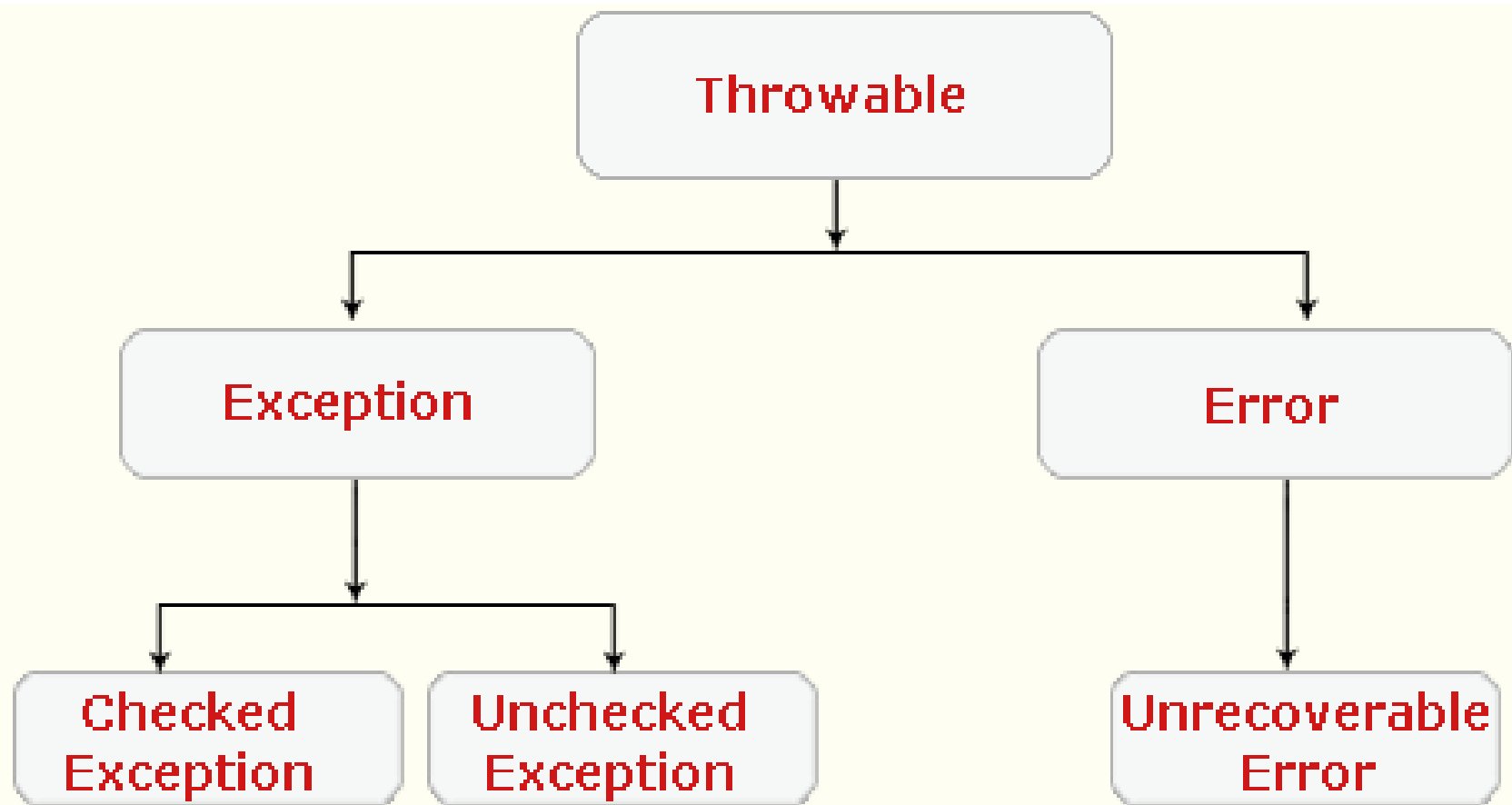
- Dividing a number by zero.
- Accessing an element that is out of bounds of an array.
- Trying to store incompatible data elements.
- Using negative value as array size.
- Trying to convert from string data to a specific data value (e.g., converting string "abc" to integer value).
- File errors:
 - opening a file in "read mode" that does not exist or no read permission
 - Opening a file in "write/update mode" which has "read only" permission.
- Corrupting memory: - common with pointers

- **Checked exceptions:** A checked exception is an exception that is typically a user error or a problem that cannot be foreseen by the programmer but that should be handled by the programmer
- For example, if a file is to be opened, but the file cannot be found, an exception occurs. These exceptions cannot simply be ignored at the time of compilation.
- **Checked at compiletime**
- **Eg IOException, SQLException**

- **Unchecked/ Runtime exceptions:** A runtime exception is an exception that occurs that probably because of a bug and that cannot be handled by the program.
- As opposed to checked exceptions, runtime exceptions are ignored at the time of compilation.
- **Checked at Runtime**
- **ArithmeticException,
ArrayIndexOutOfBoundsException**

Exception Hierarchy:

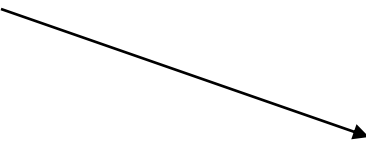
- All exception classes are subtypes of the `java.lang.Exception` class.
- The exception class is a subclass of the `Throwable` class. Other than the exception class there is another subclass called `Error` which is derived from the `Throwable` class.



Without Error Handling – Example 1

```
class NoErrorHandling{  
    public static void main(String[] args){  
        int a,b;  
        a = 7;  
        b = 0;  
        System.out.println("Result is " + a/b);  
        System.out.println("Program reached this line");  
    }  
}
```

Program does not reach here

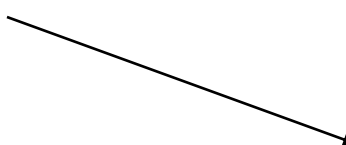


No compilation errors. While running it reports an error and stops without executing further statements:
java.lang.ArithmeticException: / by zero at Error2.main(Error2.java:10)

Traditional way of Error Handling - Example 2

```
class WithErrorHandling{
    public static void main(String[] args){
        int a,b;
        a = 7;  b = 0;
        if (b != 0){
            System.out.println("Result is " + a/b);
        }
        else{
            System.out.println(" B is zero);
        }
        System.out.println("Program is complete");
    }
}
```

Program reaches here



With Exception Handling - Example 3

```
class WithExceptionHandling{
```

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

```
        int a,b; float r;
```

```
        a = 7;  b = 0;
```

```
        try{
```

```
            r = a/b;
```

```
            System.out.println("Result is " + r);
```

```
        }
```

Program Reaches here

```
        catch(ArithmeticException e){
```

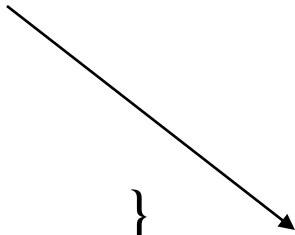
```
            System.out.println(" B is zero);
```

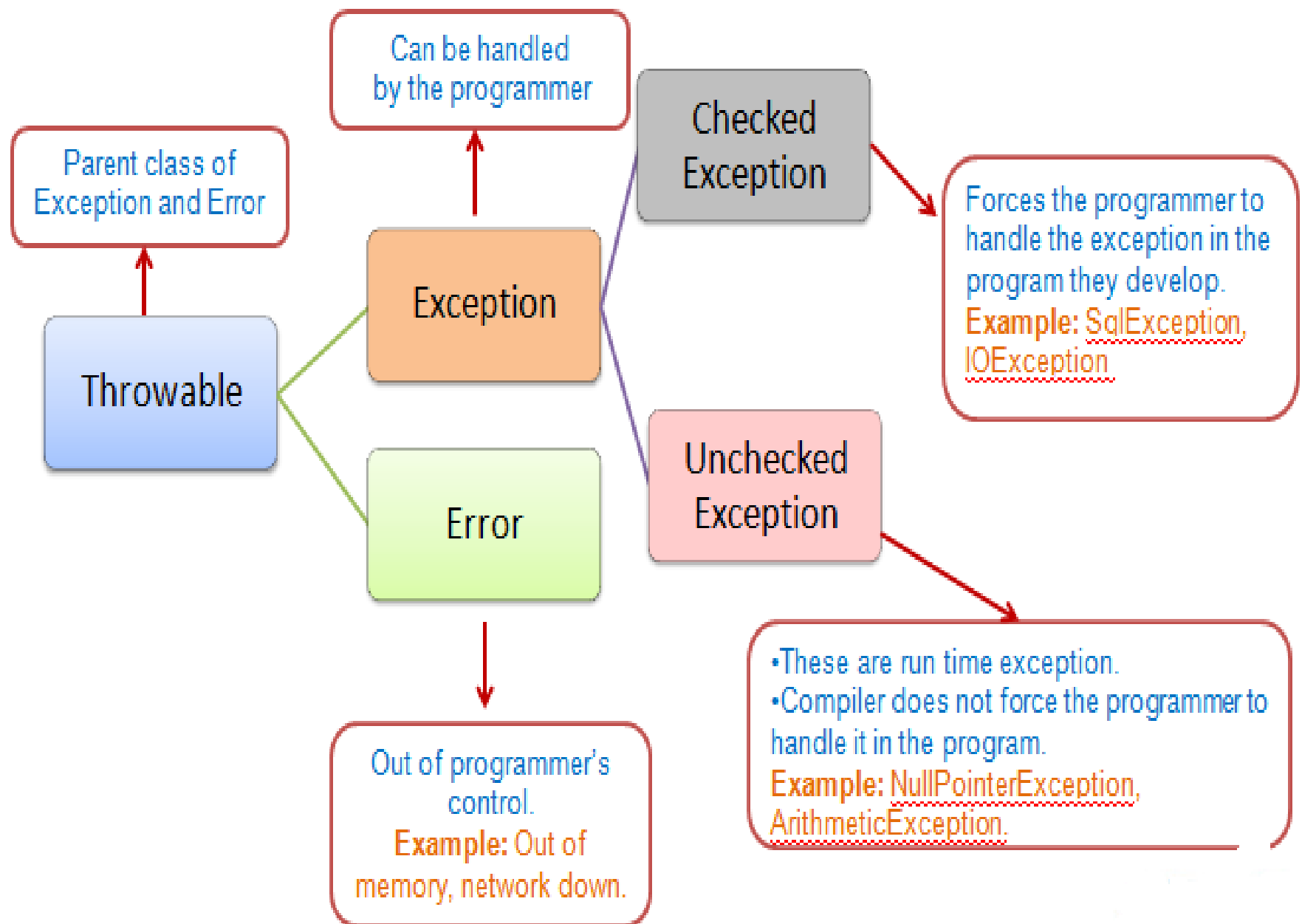
```
        }
```

```
        System.out.println("Program reached this line");
```

```
    }
```

```
}
```





Checked Exception	Unchecked Exception
At compile time, the java compiler automatically checks that a program contains handlers for checked exceptions.	The compiler doesn't force them to be declared in the throws clause.
Checked exceptions must be explicitly caught or propagated using try-catch-finally blocks	Unchecked exceptions do not have this requirement. They don't have to be caught or declared thrown.
Checked exceptions in Java extend the <i>java.lang.Exception</i> class	Unchecked exceptions extend the <i>java.lang.RuntimeException</i> .
Exception handling is mandated by JVM for these exceptions	It is not advisable to catch these exceptions since it might make the code unstable.
Example: <u>IOException</u>	Example: <u>NullPointerException</u>

Checked Exception	Description
ClassNotFoundException	This Exception occurs when Java run-time system fail to find the specified class mentioned in the program.
<u>InstantiationException</u>	This Exception occurs when you create an object of an abstract class and interface.
<u>IllegalAccessException</u>	This Exception thrown when one attempts to access a method or member that visibility qualifiers do not allow
<u>NoSuchMethodException</u>	This Exception occurs when the method you call does not exist in class.

Unchecked Exception	Description
ArithmeticException	These Exception occurs, when you divide a number by zero causes an Arithmetic Exception
ClassCastException	These Exception occurs, when you try to assign a reference variable of a class to an incompatible reference variable of another class.
NullPointerException	These Exception occurs, when you try to invoke a method on a object without instantiating it.
ArrayIndexOutOfBoundsException	These Exception occurs, when you assign an array which is not compatible with the data type of that array.

Exceptions and their Handling

- When the JVM encounters an error such as divide by zero, it creates an exception object and throws it – as a notification that an error has occurred.
- If the exception object is not caught and handled properly, the interpreter will display an error and terminate the program.
- If we want the program to continue with execution of the remaining code, then we should try to catch the exception object thrown by the error condition and then take appropriate corrective actions. This task is known as *exception handling*.

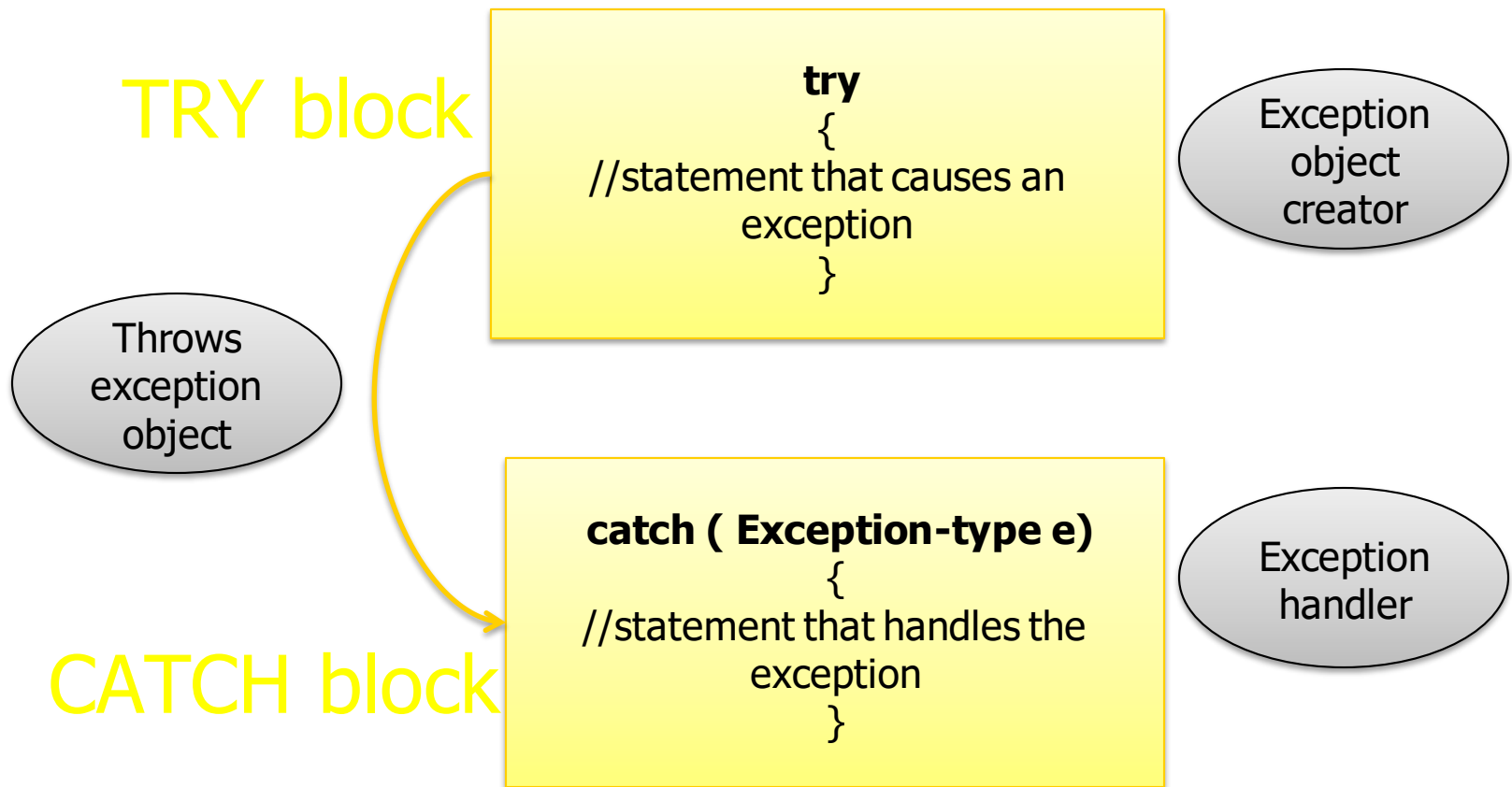
Error handling code performs the following tasks:

1. Find the problem (Hit the exception)
2. Inform that an error has occurred (Throw the exception)
3. Receive the error information (Catch the exception)
4. Take corrective actions (Handle the exception)

Exceptions in Java

- A method can signal an error condition by throwing an exception – *throws*
- The calling method can transfer control to a exception handler by catching an exception - *try, catch*
- Clean up can be done by - *finally*

Exception handling Mechanism



Syntax of Exception Handling Code

```
...  
...  
try {  
    // statements  
}  
catch( Exception-Type e)  
{  
    // statements to process exception  
}  
..  
..
```


Example 1

```
class Error
{
public static void main (String
args[])
{
    int a =10; int b=5;
    int c=5;
    int x,y;
    try
    {
        x=a/(b-c);
    }
    catch(ArithmeticException e)
```

```
{
    System.out.println("Division
by zero");
}
y=a/(b+c);
System.out.println("y ="
+y);
}
}
```

Output:

Division by zero
Y=1

EX-2 catching invalid command line arguments

```
class Input
{
    public static void main(String
        args[i])
    {
        int valid =0;
        int number,count=0;
        for(int i=0;i<args.length;i++) }
        {
            try
            {
                number =
                Integer.parseInt(args[i]);
            }
            catch
            (NumberFormatException e)
            {
                invalid = invalid+!;
                System.out.println("Invalid
                Number:" +args[i]);
                continue;
            }
            count=count+1;
        }
        System.out.println("Vslid
        Numbers="+count)
        System.out.println("Invalid
        Numbers = "+invalid);
    }
}
```

java Input **15 24.3 java 97**

Output:

Invalid Number:24.3

Invalid Number:java

Valid : 2

Invalid :2

Multiple Catch Statements

- If a try block is likely to raise more than one type of exceptions, then multiple catch blocks can be defined as follows:

```
...  
...  
try {  
    // statements  
}  
catch( Exception-Type1 e)  
{  
    // statements to process exception 1  
}  
..  
..  
catch( Exception-TypeN e)  
{  
    // statements to process exception N  
}  
...
```

Using multiple catch block

```
class Error
{
public static void main(String
    args[])
{
int a[]={6,12};
int b=2;
try
{
int x=a[2] /b-a[1];
}
catch(ArithmeticException e)
{
system.out.println("Division by
    zero");
}
```

```
catch(ArrayIndexOutOfBoundsException e)
{
System.out.println("array index
    error");
}
catch (ArrayStoreException e)
{
System.out.println ("wrong
    data type");
}
int y=a[1]/a[0];
System.out.println("y =" +y);
}
```

finally block

- Java supports definition of another block called finally that be used to handle any exception that is not caught by any of the previous statements. It may be added immediately after the try block or after the last catch block:

```
...
try {
    // statements
}
catch( Exception-Type1 e)
{
    // statements to process exception 1
}
..
..
finally {
    ....
}
```


- When a finally is defined, it is executed regardless of whether or not an exception is thrown. Therefore, it is also used to perform certain house keeping operations such as closing files and releasing system resources.

finally statement

- “finally” can be used to handle an exception that is not caught by any of the previous catch statements
- “finally” can be used to handle any exception generated within a try block
- finally block is guaranteed to execute, regardless of whether or not an exception is thrown
- For each try block there can be zero or more catch blocks, but only one finally block.
- Use to close a file and releasing system resources

Example

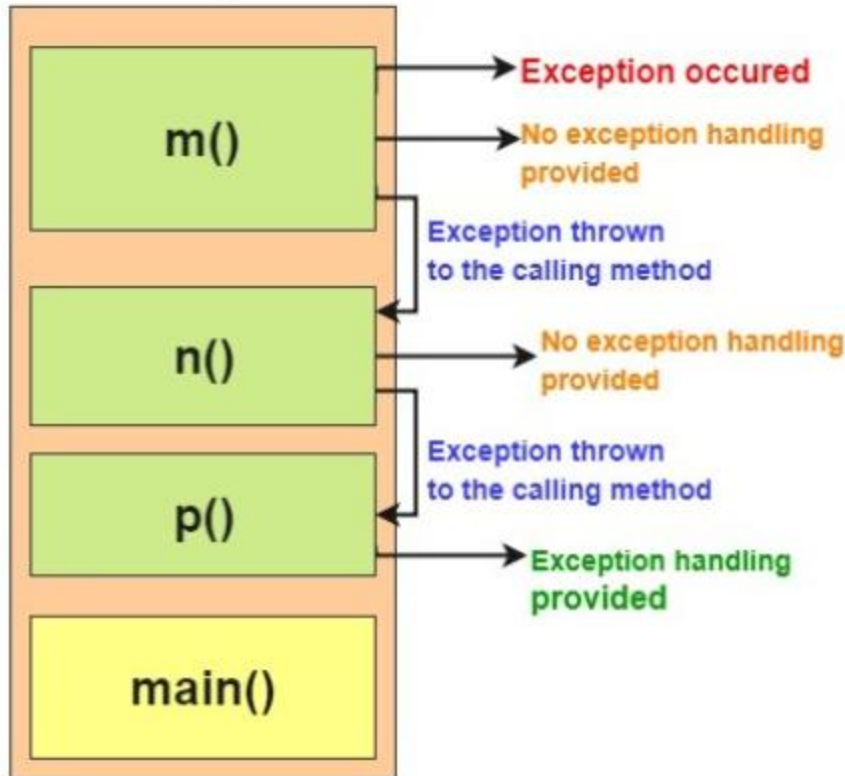
```
class TestFinallyBlock1
{
    public static void main(String[] args)
    {
        try
        {
            int data=25/0;
            System.out.println(data);
        }
        catch(NullPointerException e)
        {
            System.out.println(e);
        }
    }
}
```



```
finally
{
    System.out.println("finally block is always executed");
}
System.out.println("rest of the code...");
}
```

Exception propagation

Memory Stack



An exception is first thrown from the top of the stack and if it is not caught, it drops down the call stack to the previous method, If not caught there, the exception again drops down to the previous method, and so on until they are caught or until they reach the very bottom of the call stack. This is called exception propagation.


```
class TestExceptionPropagation1{  
void m(){ int data=50/0; }  
void n(){ m(); }  
void p(){  
try{  
n();  
}catch(Exception e)  
{System.out.println("exception handled");}  
}  
public static void main(String args[])  
{ TestExceptionPropagation1 obj=new TestExceptionPropagation1();  
  obj.p(); System.out.println("normal flow...");  
} }
```

Catching and Propagating Exceptions

- Exceptions raised in try block can be caught and then they can be thrown again/propagated after performing some operations. This can be done by using the keyword “throw” as follows:
- So far, you have only **been catching exceptions that are thrown by the Java run-time system.**
- However, it is possible for your program to throw an exception explicitly, using the **throw statement.**
- **throw** keyword is used inside a function. It is used when it is required to throw an Exception logically.
- **Throw within a block and in a method**
 - throw exception-object;
 - OR
 - throw new Throwable_Subclass;

Throw

- The flow of execution stops immediately after the **throw statement**; **any subsequent** statements are not executed.
- The nearest enclosing **try block** is inspected to see if it has a **catch statement** that matches the type of the **exception**.
- **If it does find a match, control** is transferred to that statement.
- If not, then the next enclosing **try statement** is **inspected**, and so on.
- If no matching **catch** is found, then the **default exception handler** halts the program and prints the stack trace.

Throw

- 1)Already existing exceptions
- 2)New userdefined exceptions

Throw-Example 1

```
static int sum(int num1, int num2){  
    if (num1 == 0) throw new  
        ArithmeticException("First parameter is not valid");  
    else  
        System.out.println("Both parameters are correct!!");  
    return num1+num2;  
}
```

Throw-Example 1

```
try{  
    int res=sum(0,12);  
    }catch(ArithmeticException e)  
{  
    System.out.println("Exception"+e.toString());  
}
```

Throws

- **throws** – When we are throwing any exception in a method and not handling it, then we need to use **throws keyword in method signature** to let caller program know the exceptions that might be thrown by the method.
- It is used when the function has some statements that can lead to some exceptions.
- The caller method might handle these exceptions or propagate it to its caller method using throws keyword.
- We can provide multiple exceptions in the throws clause and it can be used with main() method also.

Throws-Example

```
class Sample1
```

```
{
```

```
    void test(int x) throws ArithmeticException
```

```
    {
```

```
        if(x<20)
```

```
        {
```

```
            ArithmeticException ob=new ArithmeticException();
```

```
            throw ob;
```

```
        }
```

```
        else
```

```
        {
```

```
            System.out.println(" Validation Success!");
```

```
        }  
    }  
}
```


Throws-Example

```
Sample1 s1=new Sample1();
```

```
try
```

```
{
```

```
    s1.test(10);
```

```
} catch(ArithmeticException e)
```

```
{System.out.println("Exception"+e.toString());}
```

Defining Your Own Exceptions-User defined exception

- To define your own exception you must do the following:
 - Create an exception class to hold the exception data.
 - Your exception class must subclass "Exception" or another exception class
 - Minimally, your exception class should provide a constructor which takes the exception description as its argument.
- To throw your own exceptions:
 - If your exception is checked, any method which is going to throw the exception must define it using the throws keyword
 - When an exceptional condition occurs, create a new instance of the exception and throw it.

User-Defined Exceptions-Ex-1-class

```
class MyOwnException extends Exception {  
    public MyOwnException(String msg){  
        super(msg);  
    }  
}
```

Throwing our exceptions

- Throw statement
 throw new throwable_subclass;

- Throws statement

```
return_type method_name() throws except  
ion_class_name  
{  
    //method code  
}
```

User-Defined Exceptions-Ex-1- Method

```
static void employeeAge(int age)
throws MyOwnException{
    if(age < 0)
    throw new MyOwnException("Age not less than zero")
    else
    System.out.println("Input is valid!!");
}
```

User-Defined Exceptions-Ex-1-

Method call

```
try {  
    employeeAge(-2);  
}  
catch (MyOwnException e) {  
    e.printStackTrace();  
}
```

User-Defined Exceptions

■ Problem Statement :

- Consider the example of the Circle class
- Circle class had the following constructor

```
public Circle(double centreX, double centreY,  
              double radius){  
    x = centreX; y = centreY; r = radius;  
}
```

- How would we ensure that the radius is not zero or negative?

Defining your own exceptions

```
import java.lang.Exception;  
class InvalidRadiusException extends Exception {  
  
    private double r;  
  
    public InvalidRadiusException(double radius){  
        r = radius;  
    }  
    public void printError(){  
        System.out.println("Radius [" + r + "] is not valid");  
    }  
}
```


Throwing the exception

```
class Circle {
    double x, y, r;

    public Circle (double centreX, double centreY, double
radius ) throws InvalidRadiusException {
        if (r <= 0 ) {
            throw new InvalidRadiusException(radius);
        }
        else {
            x = centreX ; y = centreY; r = radius;
        }
    }
}
```

Catching the exception

```
class CircleTest {  
    public static void main(String[] args){  
        try{  
            Circle c1 = new Circle(10, 10, -1);  
            System.out.println("Circle created");  
        }  
        catch(InvalidRadiusException e)  
        {  
            e.printStackTrace();  
        }  
    }  
}
```

<u>No.</u>	<u>Throw</u>	<u>Throws</u>
1)	Java throw keyword is used to explicitly throw an exception.	Java throws keyword is used to declare an exception.
2)	Checked exception cannot be propagated using throw only.	Checked exception can be propagated with throws.
3)	Throw is followed by an instance.	Throws is followed by class.
4)	Throw is used within the method.	Throws is used with the method signature.
5)	You cannot throw multiple exceptions.	You can declare multiple exceptions e.g. public void method()throws IOException,SQLException.

```

class FactException
    extends Exception
{
public FactException
    (String s)
    {
        super(s);
    }
}

public class
    UseFactorial
{
public long dofact (int
    a) throws
    FactException
    {
        long f=1;
        if(a>0)
        {
            for(int
                i=1;i<=a;i++)
            {
                f=f*i;
            }
        }
    }
}

```

```

    }
    return f;
}
else if(a==0)
    return
    1;
    else
    {
        throw new
        FactException
        ("Factorial is avaible
        only for positive
        number");
    }
}

public static void
main(String arg[])
{
    try
    {
        int
        a=Integer.parseInt
        (arg[0]);
    }
}

```

```

UseFactorial uf=new
UseFactorial();
System.out.println
(uf.dofact(a));
}
catch(FactException e)
{
    System.out.println
    (e.getMessage());
}
}
}

```