JAVA PROGRAMMING

Chapter 4

File and Exception Handling

Chaskar R. R.

Definition

The Exception Handling in Java is one of the powerful mechanism to handle the runtime errors so that normal flow of the application can be maintained.

Types of Exceptions

1. Checked Exception
2. Unchecked Exception
3. Error

1) Checked Exception

- The classes which directly inherit Throwable class except RuntimeException and Error are known as checked exceptions.
- Checked exceptions are checked at compile-time.

e.g. IOException, SQLException

2) Unchecked Exception

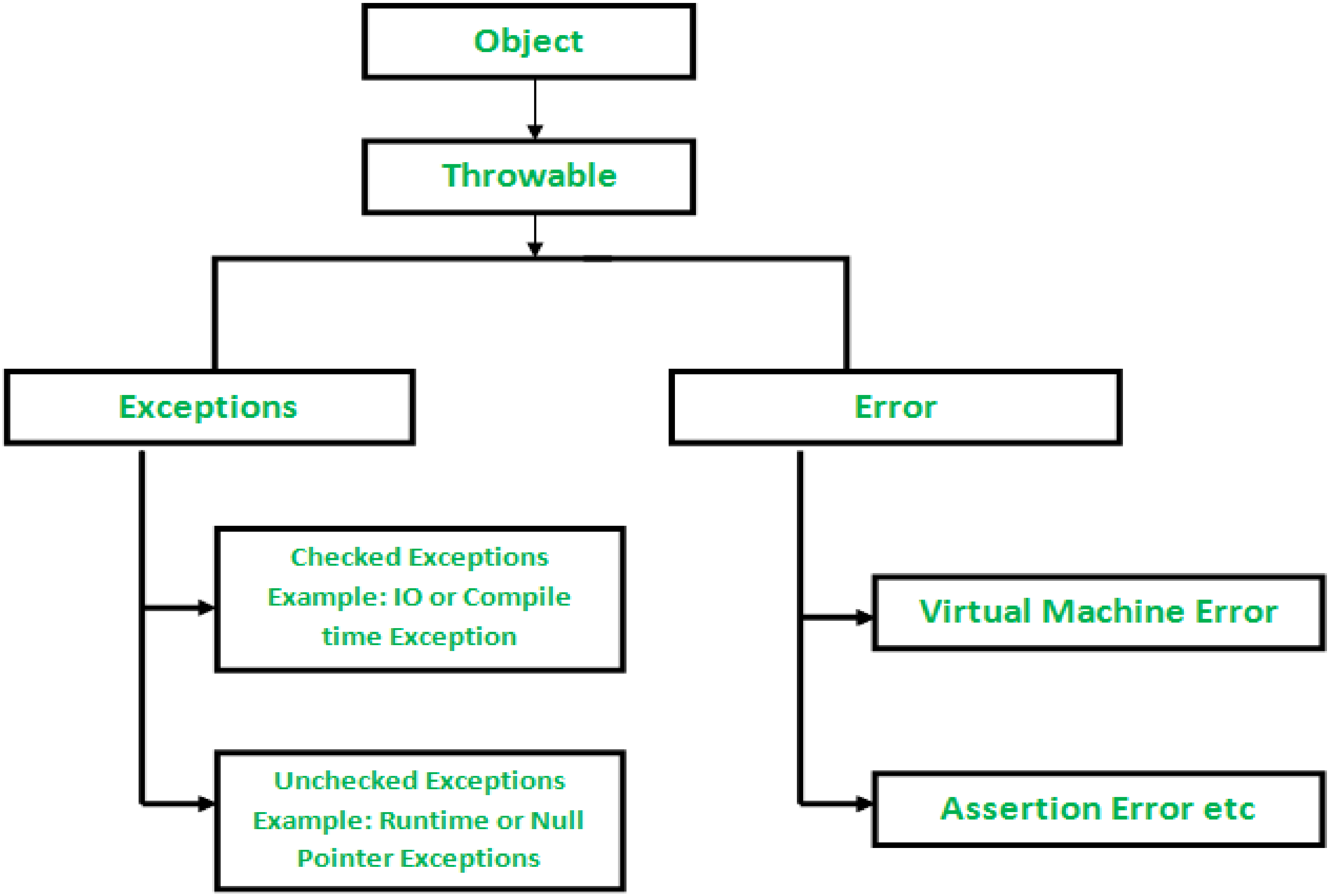
- The classes which inherit RuntimeException are known as unchecked exceptions
- Unchecked exceptions are not checked at compile-time, but they are checked at runtime.

e.g. ArithmeticException, NullPointerException, ArrayIndexOutOfBoundsException.

3) Error

Error is irrecoverable.

e.g. OutOfMemoryError, AssertionError etc.


Object
Throwable
Exceptions
Error
Checked Exceptions Example: IO or Compile time Exception
Unchecked Exceptions Example: Runtime or Null Pointer Exceptions
Virtual Machine Error
Assertion Error etc

Java Exception Keywords

- 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.

E.g.

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

```
O/P- java.lang.ArithmeticException: / by zero
        rest of the code...
```

Common Scenarios of Java Exceptions

```
1) int a=50/0;//ArithmeticException
2) string s=null;
   system.out.println(s.length()); //NullPointerException
3) string s="abc";
    int i=Integer.parseInt(s); //NumberFormatException
4) int a[]=new int[5];
   a[10]=50; //ArrayIndexOutOfBoundsException
```

try block

- Java **try** block is used to enclose the code that might throw an exception. It must be used within the method.
- Java try block must be followed by either catch or finally block.

Syntax

```
Try
{
    }
catch(Exception_class_Name ref)//code that may throw an exception
{
   }
```

Java catch block

- Java catch block is used to handle the Exception by declaring the type of exception within the parameter.
- he catch block must be used after the try block only.
- You can use multiple catch block with a single try block.

Java Multi-catch block

- A try block can be followed by one or more catch blocks.
- Each catch block must contain a different exception handler.

Program-

```
public class MultipleCatchBlock
 {
     public static void main(String[] args)
{
       try{
              int a[]=new int[5];
              a[5]=30/0;
             }
             catch(ArithmeticException e)
               {
 System.out.println("Arithmetic Exception oc
curs");
               }
catch(ArrayIndexOutOfBoundsException e)

               {
System.out.println("ArrayIndexOutOfBounds
Exception occurs");
                }
              catch(Exception e)
                {
System.out.println("Parent Exception occurs");
       }
      System.out.println("rest of the code");
    }
}
```

Nested try block

- The try block within a try block is known as nested try block in java

Syntax:

```
....
try
{
    statement 1;
    statement 2;
    try
    {
        statement 1;
        statement 2;
    }
    catch(Exception e)
    {
    }
}
catch(Exception e)
{
}
....
```

throws keyword

- The **Java throws keyword** is used to declare an exception.
- It gives an information to the programmer that there may occur an exception so it is better for the programmer to provide the exception handling code so that normal flow can be maintained.

SYNTAX

```
return_type method_name() throws exception_class_
name
{
//method code
}
```

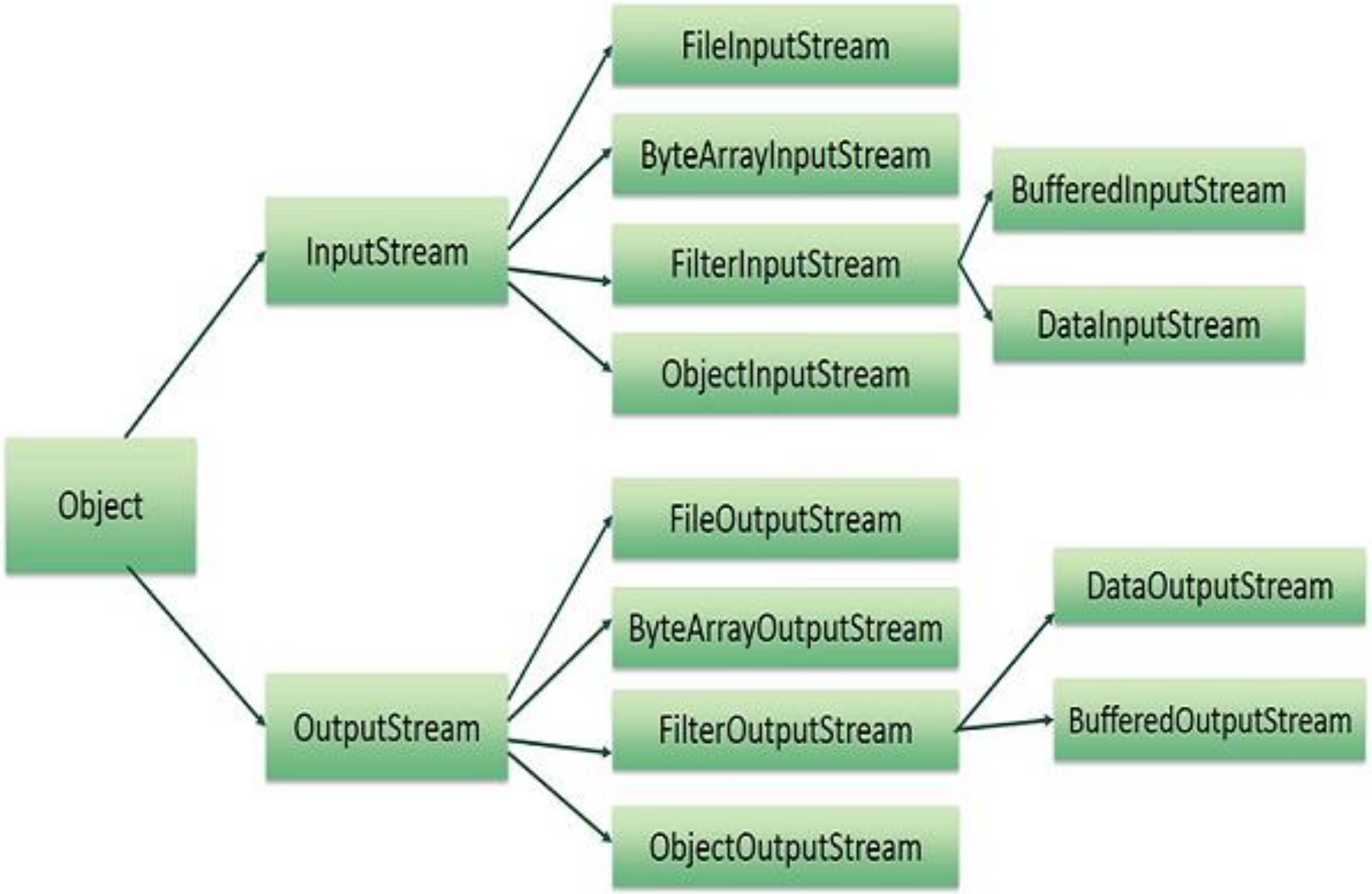
Java finally block

- **Java finally block** is a block that is used *to execute important code* such as closing connection, stream etc.
- Java finally block is always executed whether exception is handled or not.
- Java finally block follows try or catch block.

Program

```
class TestFinallyBlock
{
  public static void main(String args[])
{
  try
   {
     int data=25/5;
     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...");
  }
}
```

INPUT OUTPUT STREAM



Input Stream

- Java application uses an input stream to read data from a source
- It may be a file, an array, peripheral device or socket.

METHOD

- public int available():- returns an estimate of the number of bytes that can be read from the current input stream.
- public void close():- is used to close the current input stream.
- public abstract int read():- reads the next byte of data from the input stream. It returns -1 at the end of the file.

Program

```
package com.javatpoint;
import java.io.FileInputStream;
public class DataStreamExample
{
     public static void main(String args[])
{
          try
{
        FileInputStream fin=new FileInputStream("D:\\testout.txt");
            int i=0;
            while((i=fin.read())!=-1)
{
             System.out.print((char)i);
    }
            fin.close();
          }
catch(Exception e)
{
System.out.println(e);
}  }
        }
```

Output Stream

- Java application uses an output stream to write data to a destination
- It may be a file, an array, peripheral device or socket.

METHOD

- public void close() :- is used to close the current output stream.
- public void write(int):- is used to write a byte to the current output stream.
- public void flush():- flushes the current output stream.
- public void write(byte[]):- is used to write an array of byte to the current output stream.

Program:-

```
import java.io.FileOutputStream;
public class FileOutputStreamExample
 {
   public static void main(String args[])
{
           try
 {
             FileOutputStream fout=new FileOutputStream("D:\\testout.txt");
             fout.write(65);
             fout.close();
             System.out.println("success...");
  }
catch(Exception e)
{
System.out.println(e);
} }
}
```

Java FileReader Class

- Java FileReader class is used to read data from the file.

SYNTAX

public class FileReader **extends** InputStreamReader

CONSTRUCTOR

- FileReader(String file):- It gets filename in string. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.
- FileReader(File file):- It gets filename in file instance. It opens the given file in read mode. If file doesn't exist, it throws FileNotFoundException.

METHOD

- int read():- It is used to return a character in ASCII form. It returns -1 at the end of file.
- void close():- It is used to close the FileReader class.