

Java Programming

Unit 3: Contents

Inheritance and Exception Handling:

Inheritance, super Keyword, final Keyword, Method Overriding and Abstract Class. Interfaces, Creating Packages, Using Packages, Importance of Class path. Exception Handling, Importance of try, catch, throw, throws and finally Block.

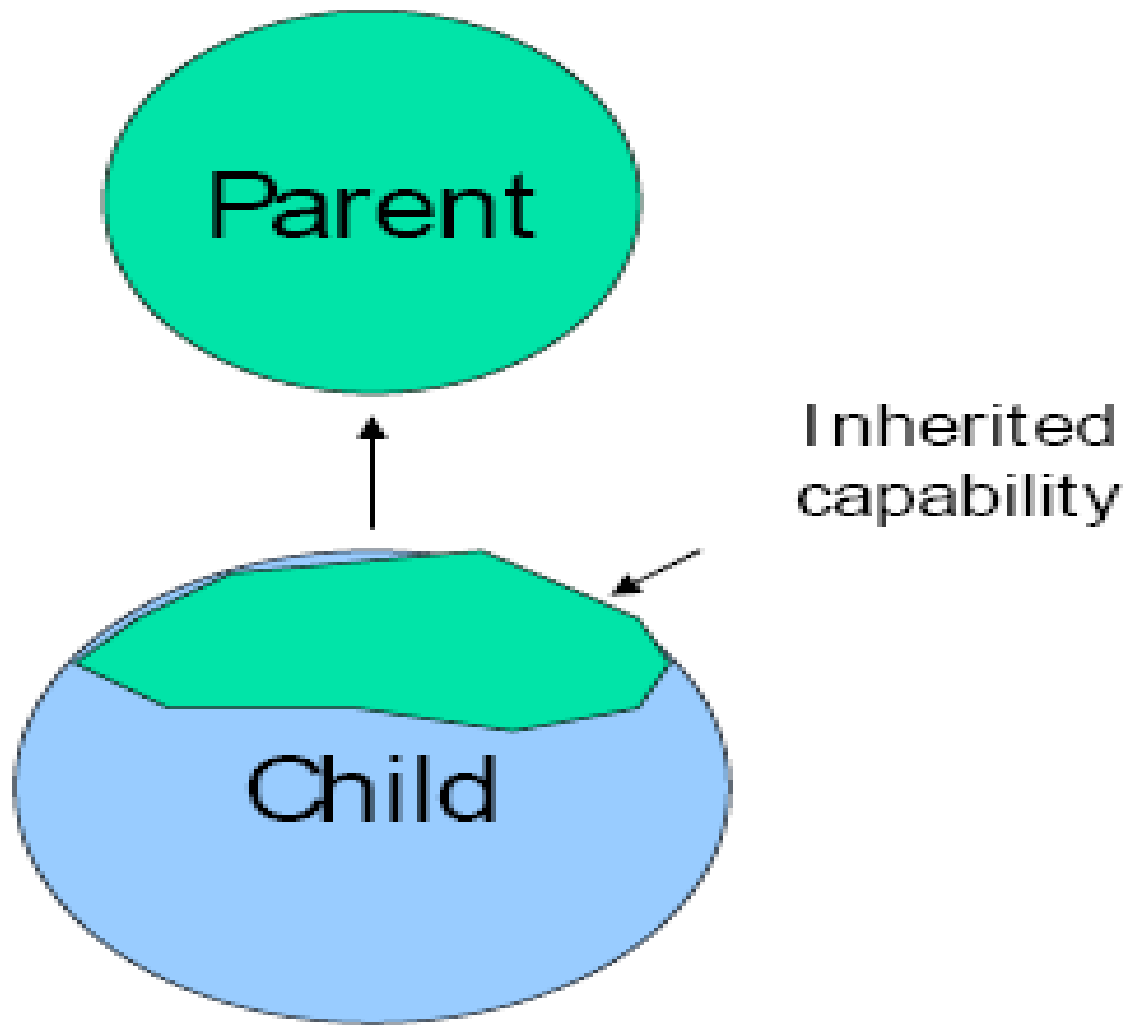
Inheritance

The process of deriving all the information from one class into another class known as **inheritance**. **Inheritance** is a compile-time mechanism in java that allows you to extend a class (called the **base class** or **super class**) with another class (called the **derived class** or **subclass**).

Which providing many advantages.

1. Reusability
2. Reducing design time and software cost
3. Reducing file size and disk space
4. Easy to maintainable

Inheritance Capability



Parent Class:

The class whose properties and functionalities are **used(inherited)** by another class is known as parent class, super class or Base class.

Child Class:

The class that **extends** the features of parent class is known as child class, sub class or derived class.

extends

It is a keyword. It gives a instruction to compiler that class deriving the information from another class.

EXAMPLE:

```
class Abc
{
    // Abc is Parent Class

}

class Xyz extends Abc
{
    // Xyz Is Child Class

}
```

Types of inheritances

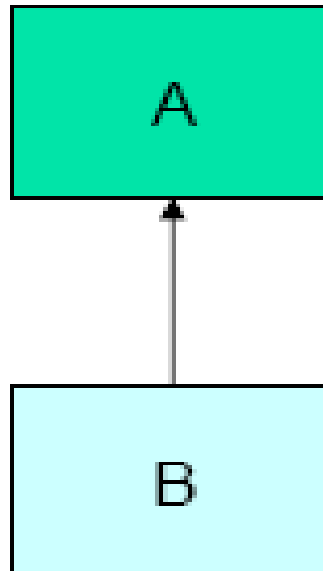
In java we have multiple types of inheritances.
They are as follows:

- 1) Single Inheritance
- 2) Multi-Level Inheritance
- 3) Hierarchical level Inheritance
- 4) Multiple Inheritance

1. Single Level Inheritance

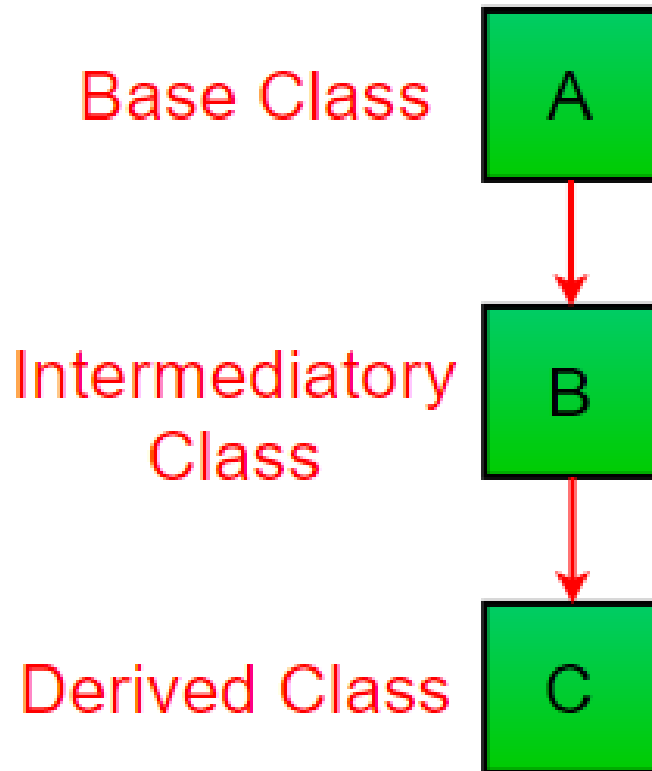
Single level

- One super class has one sub class.
- Here A is super class and B is sub-class



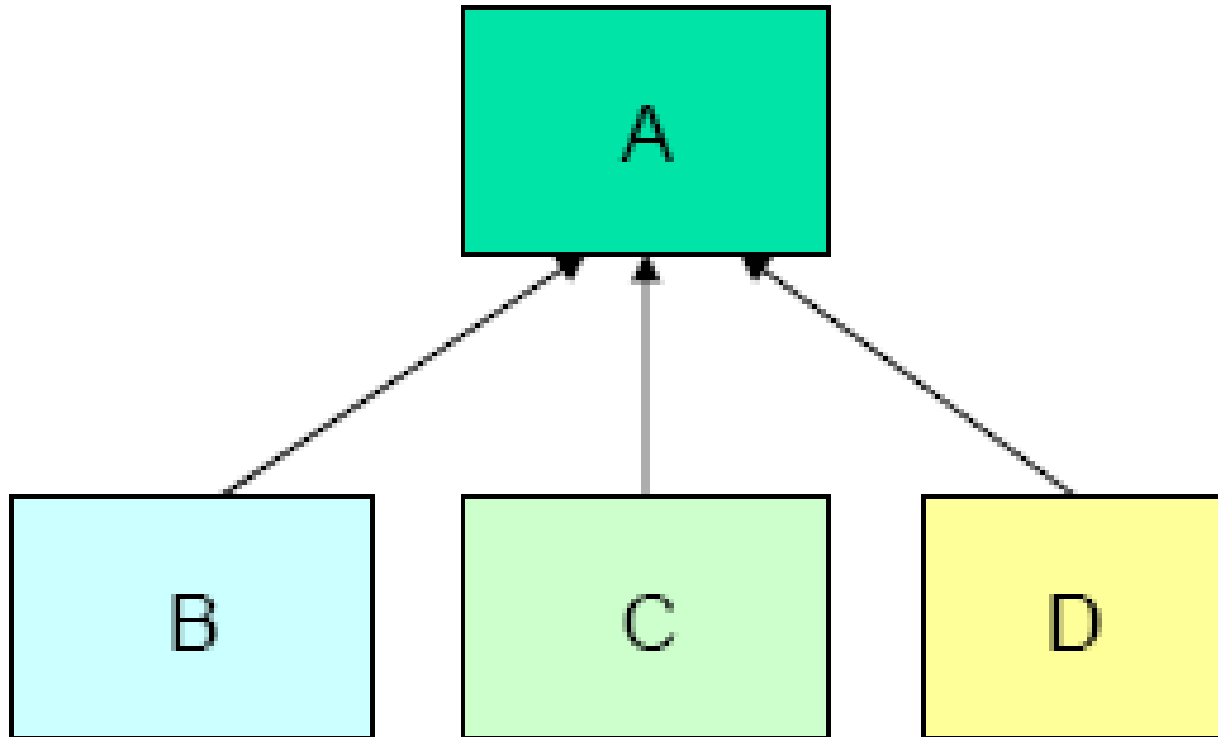
2. Multi Level Inheritance

One super class has one sub class. That sub class has one more sub class like.



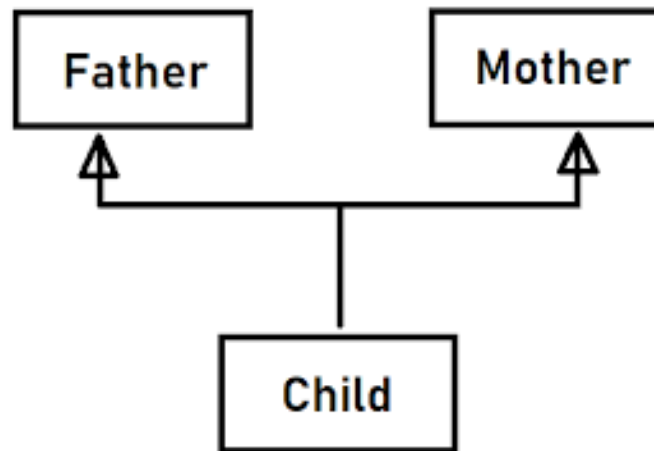
3. Hierarchical level

One super class has many sub classes individually.



4. Multiple Inheritance

Multiple Inheritance is a feature of an object-oriented concept, where a class can inherit properties of more than one parent class. The problem occurs when there exist methods with the same signature in both the super classes and subclass. On calling the method, the compiler cannot determine which class method to be called and even on calling which class method gets the priority. This is possible by using **Interfaces**



super keyword in java

The **super** keyword in Java is a reference variable which is used to refer immediate parent class object. Whenever you create the instance of subclass, an instance of parent class is created implicitly which is referred by super reference variable. The **super keyword** can also be used to **invoke the parent class constructor**.

```
1 /*Write program to explain accessing
2 parent class instance overridden
3 (variable with same name in parent
4 and child class) variable from
5 child class.*/
6
7 class Parent
8 {
9 //parent class instance variable
10 int var=54;
11 }
12 public class Child extends Parent
13 {
14 //child class instance variable
15 int var=74;
16 public static void main(String[] args)
17 {
18 Child obj = new Child();
19 obj.display();
20 }
21 void display()
22 {
23 System.out.println(var); //refers child class var
24 System.out.println(super.var); // refers parent class var
25 }
26 }
```

```
/*3. Write a program to invoke parent  
class no argument constructor from  
child class using super */
```

```
class Parent2  
{  
    //parent class constructor  
    Parent2()  
    {  
        System.out.println("no argument constructor of parent class");  
    }  
}  
public class Child2 extends Parent2  
{  
    //child class constructor  
    Child2()  
    {  
        super(); // calls parent class no argument constructor  
        System.out.println("no argument constructor of child class");  
    }  
    public static void main(String[] args)  
    {  
        Child2 obj = new Child2(); // calls child class constructor  
    }  
}
```

```
/*2. Write program to explain
accessing parent class overridden
method (method with same name
in parent and child class)
from child class.*/

class Parent1
{
    //parent class print method
    void print()
    {
        System.out.println("print method of parent class");
    }
}

public class Child1 extends Parent1
{
    public static void main(String[] args)
    {
        Child1 obj = new Child1();
        obj.display();
    }
    void display()
    {
        print(); // executes child class print method
        super.print(); // executed parent class print method
    }
    //child class print method
    void print()
    {
        System.out.println("print method of child class");
    }
}
```

Final Keyword

The final keyword in java is used to restrict the access to parent class variables and methods, and even to prevent the class from being inherit. A Final can be applied for variables, methods, and class.

Java final variable

If you make any variable as final, you cannot change the value of final variable(It will be constant).

Write a program using final variable, after initializing it with a value. try to update the variable again. mention the compile time error that occurs while modifying final variable.

```
class Parent
{
    final int i =65;
    void display()
    {
        i = 60;
        System.out.println(i);
        System.out.println("method display of class Parent");
    }
    public static void main(String args[])
    {
        Parent ob = new Parent();
        ob.display();
    } }
```

Write a program to extend a class that is final, mention the error while you obtain while implementing that
Note: We cannot extend a final class.

```
final class Parent
{
    final void display()
    {
        System.out.println("This method cannot be overridden in
        child class");
    }
}

public class Child extends Parent
{
    public static void main(String args[])
    {
        Child ch = new Child();
    }
}
```

Method Overriding

If **subclass** (child class) has the same method as declared in the parent class, then the function is said to be overridden in child class. When we create an object and try to invoke the method with same name, method from child class will be invoked. It is also called as **dynamic polymorphism**.

write a program to explain how method overriding takes place between parent and child class

```
class Parent
{
    void display()
    {
        System.out.println("method display of class Parent");
    }
}

public class Child extends Parent
{
    void display()
    {
        System.out.println("method display of class Child");
    }
    public static void main(String args[])
    {
        Child ch = new Child();
        ch.display(); // calls child display method
    }
}
```

OUTPUT:

method display of class Child

Note: When we want to access parent class overridden functions we can do, using super.

Abstract classes & Methods

Abstract method: A method which is declared as abstract and does not have implementation is known as an abstract method.

```
abstract void printStatus();
```

```
// no functionality will be defined when  
declared abstract.
```

Abstract class:

A class which is declared abstract, is known as an **abstract class**. It can have both abstract and non-abstract methods.

1. An abstract class must be declared with an **abstract** keyword.
2. It can have abstract and non-abstract methods.
3. object cannot be created for abstract classes.
4. It can have constructors and static methods.
5. It can have final methods which will force the subclass not to change the body of the method.

Write a program explaining how to use abstract and non-abstract methods of an abstract class.

```
abstract class Parent
{
    void method1()
    {
        System.out.println("this is a non abstract method class
                             Parent");
    }
    // abstract methods will not have any functionality
    abstract int method2(int a, int b);
}
class Child extends Parent
{
    public static void main(String args[])
    {
        Child ob = new Child();
        ob.method1();
        int sum = ob.method2(8,9);
        System.out.println("sum of two values is "+sum);
    }
    int method2(int i, int j)
    {
        return i+j;
    }
}
```

Expected Output

this is a non abstract method class Parent
sum of two values is 17

Note : If we extend an abstract class, we must implement all the abstract methods of the parent abstract class. otherwise we should declare implementing class also as abstract.

Interfaces

In English, an interface is a device or system that unrelated entities use to interact. According to this definition, a remote control is an interface between you and a television set, the English language is an interface between two people.

A Java interface defines a set of methods but does not implement them. A class that implements the interface agrees to implement all of the methods defined in the interface, thereby agreeing to certain behavior.

Interface

An interface is a class declared by keyword interface, in interface all the methods will be abstract.

1. An interface does not contain any constructors.
2. All of the methods in an interface are abstract.
3. An interface cannot contain instance fields. The only fields that can appear in an interface must be declared both static and final.
4. An interface is not extended by a class, it is implemented by a class.
5. An interface can extend multiple interfaces.

Interface declaration

```
interface interfaceName  
{  
    method declaration // abstract, by default  
}
```

Note:

1. An interface is implicitly abstract. You do not need to use the **abstract keyword while declaring an interface.**
2. Each method in an interface is also implicitly abstract, so the abstract keyword is not needed.
3. Methods in an interface are implicitly public

Define an interface which specifies there should be functions called sum and multiply that accepts two integer numbers and returns sum and multiplication values respectively. also write a class to implement that interface.

```
interface InterfaceOne
{
int sum(int a, int b);
int multiply(int i, int j);
}
class Impl implements InterfaceOne
{
public int sum(int a1, int a2)
{
int sum;
sum = a1+a2;
return sum;
}
public int multiply(int b1, int b2)
{
int mul;
mul = b1 * b2;
return mul;
}
public static void main(String args[])
{
Impl obj = new Impl();
int r1 = obj.sum(2,4);
System.out.println("sum of values is "+r1);
int r2 = obj.multiply(3, 4);
System.out.println("multiplication of values is "+r2);
}
}
```

Packages

Packages in Java are used to group related classes together. name conflicts can also be resolved using packages and helps in maintaining readability of code. Java uses file system directory for packages.

Types of packages:

- 1. Built in packages**
- 2. user defined packages**

Built in packages

classes which are a part of Java **API**, are called **built in packages**.

Example:

1. **java.lang,**
2. **java.io,**
3. **java.util, etc.,**

User-defined packages –

These are the packages defined by the user.

creating a package:

First, we create a folder say, myPackage (name should be same as the name of the package we want create).

Then create any class say, MyClass inside that directory. The first statement should be

package packagename

Example program for Package Creation

```
package firstpackage;  
class A  
{  
    public static void main(String args[])  
    {  
        display();  
    }  
    public void display()  
    {  
        System.out.println("first program on packages");  
    }  
}
```

Compile: `javac firstpackage/A.java`

Execute: `java firstpackage/A`

Importance of Classpath

CLASSPATH:

CLASSPATH is an environment variable in Java, and tells Java applications and the Java Virtual Machine (JVM) where to find the classes that we use in the program.

setting class path from command prompt

Example:

```
SET CLASSPATH=%CLASSPATH%;C:\Desktop\Vasavi
```


Example Program

```
class MyProgram
```

```
{  
    public void display()  
    {  
        System.out.println("from display method of class  
myprogram");  
    }  
}
```

Save this program in **D Drive** with file name **MyProgram.java**

Now create another program in **E drive** with filename
ClassPathExample.java

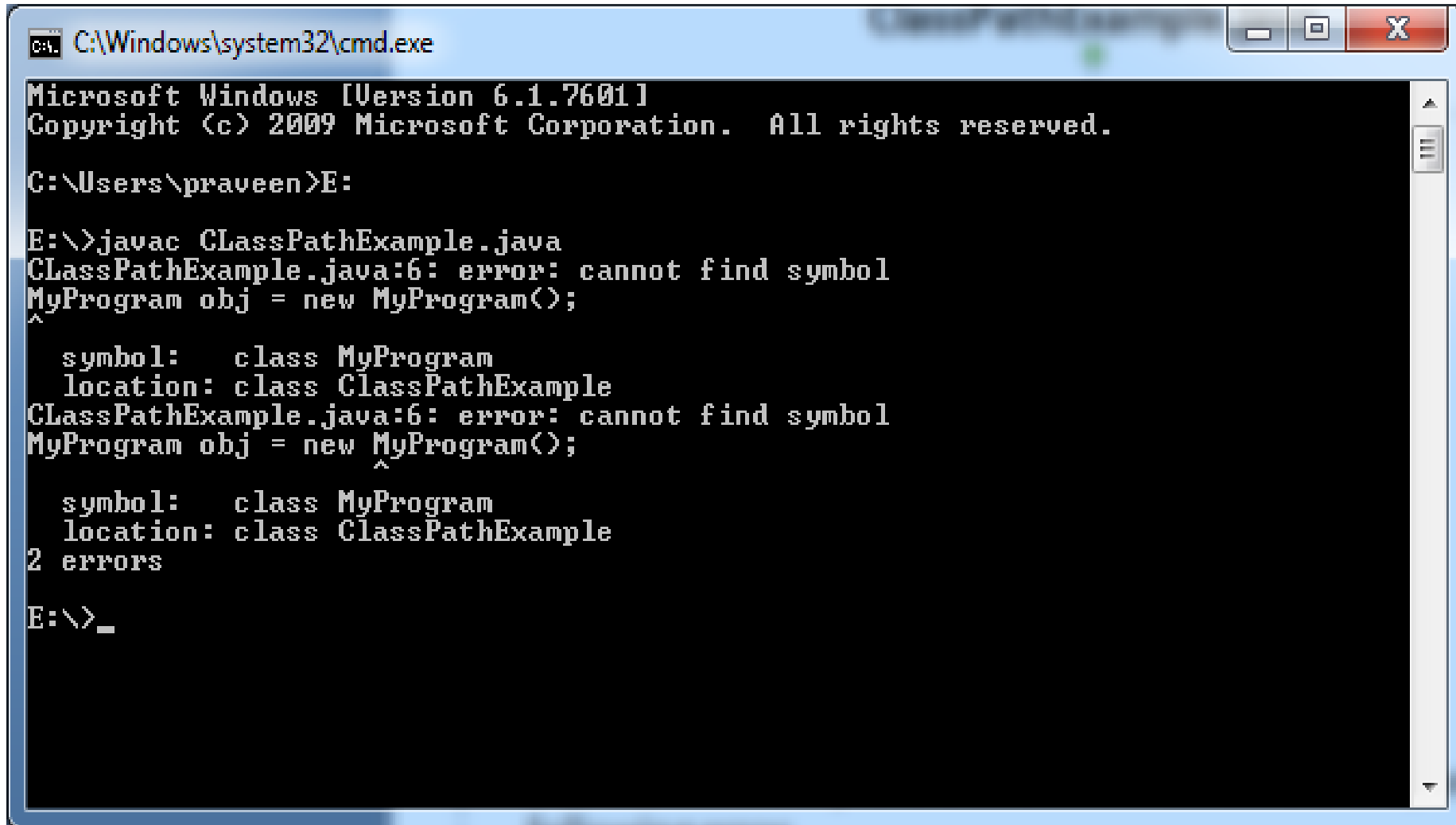
```
public class ClassPathExample
{
    public static void main(String[] args)
    {

        MyProgram obj = new MyProgram();
        obj.display();
    }
}
```

Note:

Now if we try to execute this file from E drive , we will get following error

We get following error because MyProgram.java is not in E-Drive



```
C:\Windows\system32\cmd.exe

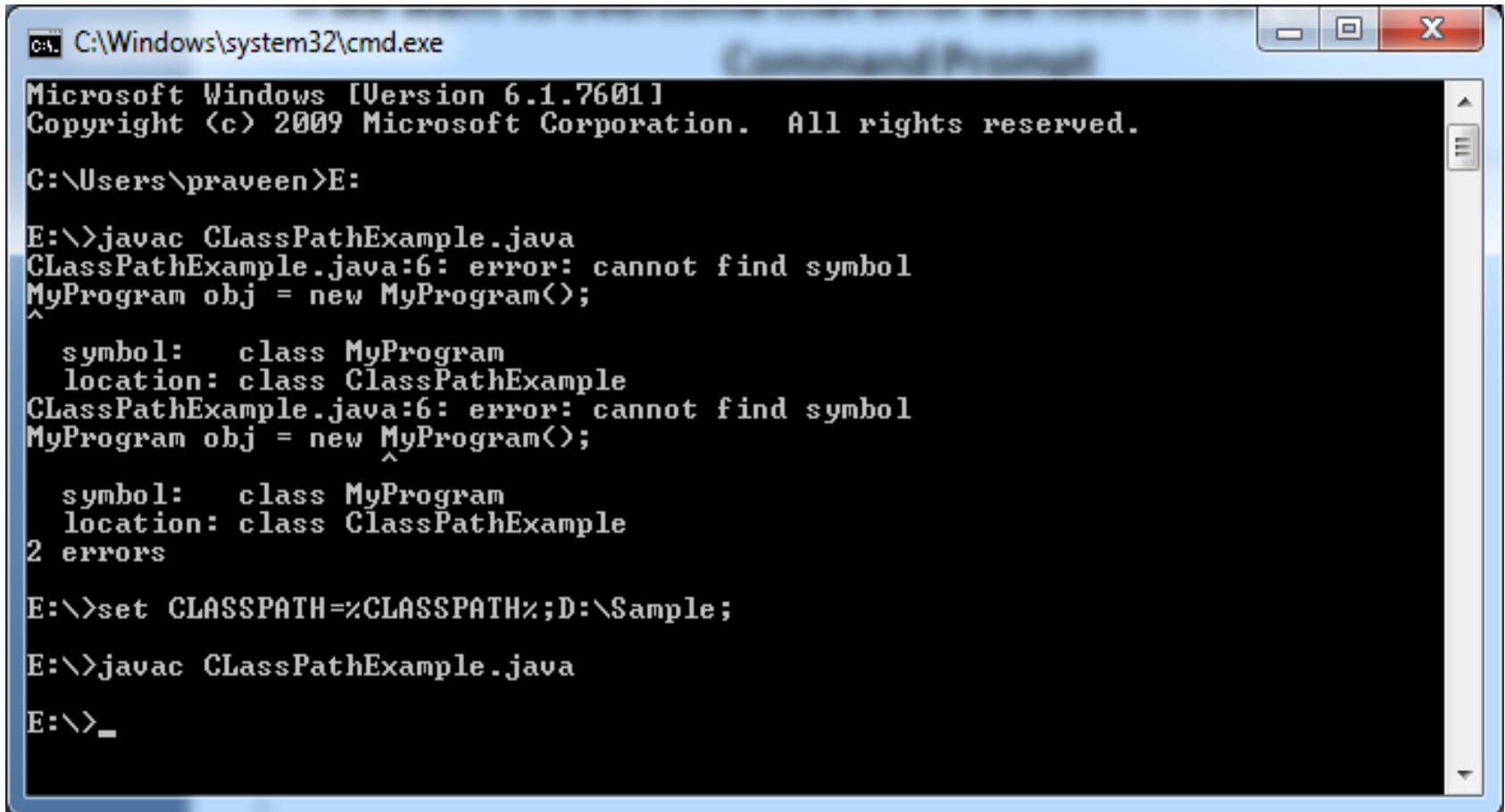
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\praveen>E:

E:\>javac ClassPathExample.java
ClassPathExample.java:6: error: cannot find symbol
MyProgram obj = new MyProgram();
^
    symbol:   class MyProgram
    location: class ClassPathExample
ClassPathExample.java:6: error: cannot find symbol
MyProgram obj = new MyProgram();
                        ^
    symbol:   class MyProgram
    location: class ClassPathExample
2 errors

E:\>_
```

If we want to overcome that error we need to set Classpath in Command Prompt



```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\praveen>E:

E:\>javac ClassPathExample.java
ClassPathExample.java:6: error: cannot find symbol
MyProgram obj = new MyProgram();
^
    symbol:   class MyProgram
    location: class ClassPathExample
ClassPathExample.java:6: error: cannot find symbol
MyProgram obj = new ^MyProgram();
                    ^
    symbol:   class MyProgram
    location: class ClassPathExample
2 errors

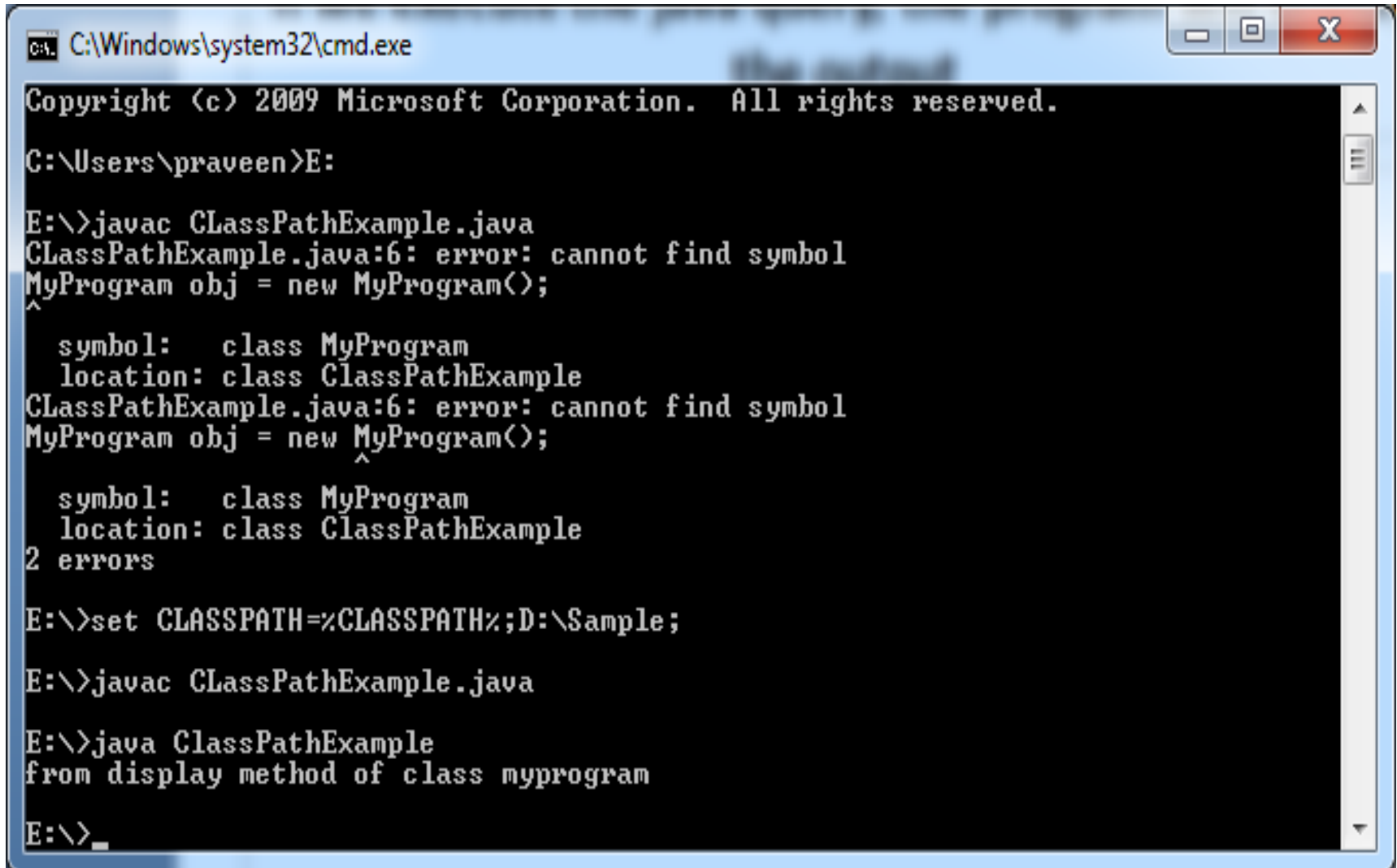
E:\>set CLASSPATH=%CLASSPATH%;D:\Sample;

E:\>javac ClassPathExample.java

E:\>_
```

NOTE: As I kept MyProgram.java file in D drive Sample folder. So that is the reason why we gave classpath till that folder

If we execute the java query, the program will display
the output



```
C:\Windows\system32\cmd.exe

Copyright (c) 2009 Microsoft Corporation. All rights reserved.

C:\Users\praveen>E:

E:\>javac ClassPathExample.java
ClassPathExample.java:6: error: cannot find symbol
MyProgram obj = new MyProgram();
^
    symbol:   class MyProgram
    location: class ClassPathExample
ClassPathExample.java:6: error: cannot find symbol
MyProgram obj = new MyProgram();
                        ^
    symbol:   class MyProgram
    location: class ClassPathExample
2 errors

E:\>set CLASSPATH=%CLASSPATH%;D:\Sample;

E:\>javac ClassPathExample.java

E:\>java ClassPathExample
from display method of class myprogram

E:\>_
```

Exception Handling

Exception in Java is an event that interrupts the execution of program-instructions and terminates the execution abnormally.

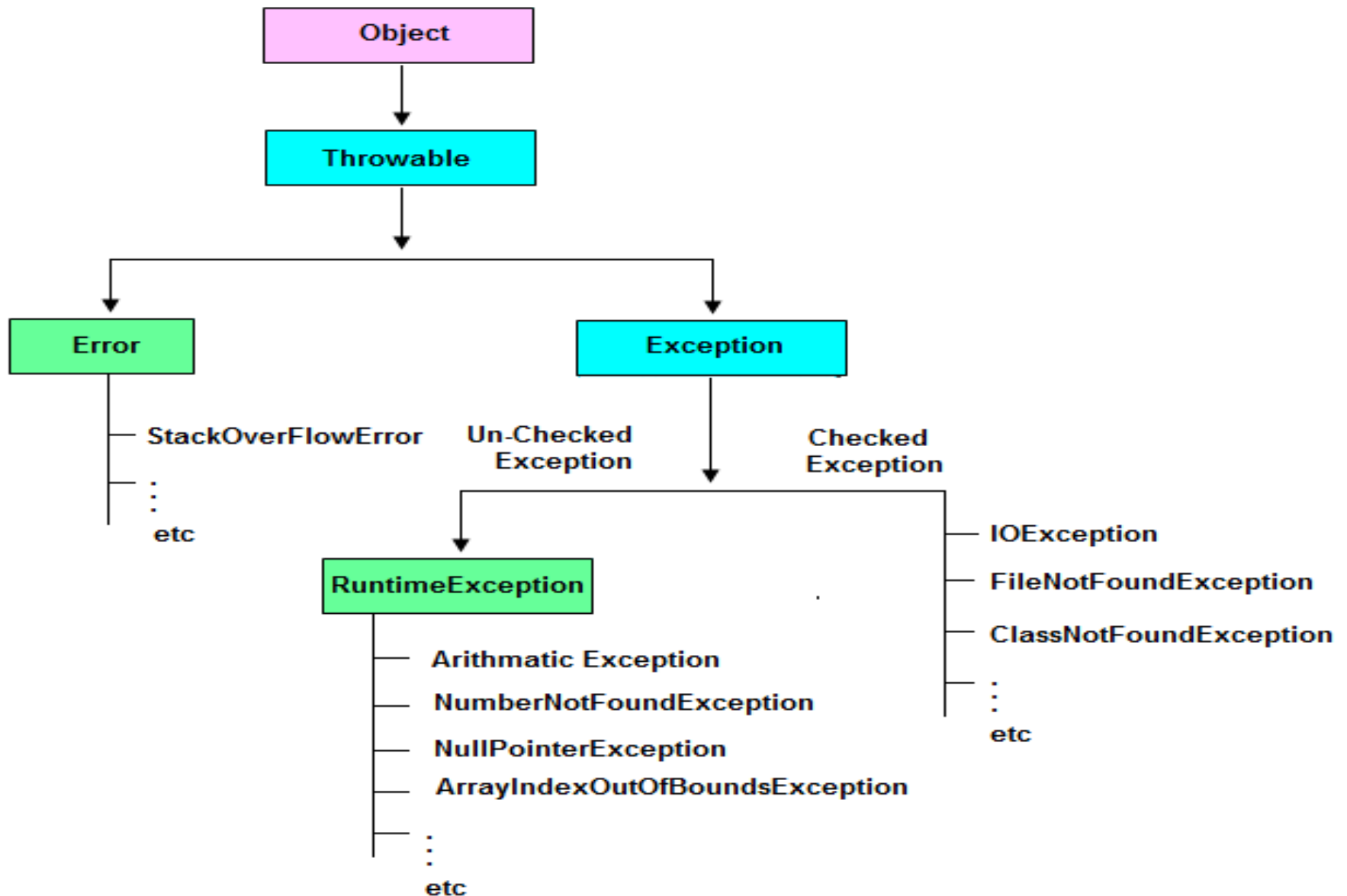
why we need exception handling

Exception handling ensures that the flow of the program doesn't break when an exception occurs. For example, if a program has large no of statements and an exception occurs in middle after executing certain statements then the statements after the exception will not be executed and the program will terminate abruptly. By handling we make sure that all the statements execute and the flow of program doesn't break.

Exceptions are mainly categorized into

1. Checked Exceptions
2. unchecked Exceptions

Exception Handling Flow Diagram



Checked exceptions/Compile time exceptions –

A checked exception is an compile time exception that will be checked by the compiler. These exceptions cannot simply be ignored, the programmer should take care of (handle) these exceptions.

Example - FileNotFoundException

Unchecked exceptions/Runtime Exceptions –

An unchecked exception is an exception that occurs at the time of execution. Runtime exceptions will not be checked by the compiler. It is the programmer responsibility to handle these exceptions.

Example – Arithmetic Exception, ArrayIndexOutOfBoundsException

Errors doesn't come under the category of Exceptions

Errors can be categorized to

1. Compile time errors
2. Any syntax or semantic error in a program

example –

flt a; -> instead of float a;
a = b+c , missing semicolon

2. Runtime errors

Errors / Runtime errors – Errors are the conditions from which application cannot get recovered by any handling techniques. It will cause termination of the program abnormally. Errors occur at runtime.

example:

StackOverflowError

UnsupportedClassVersionError

VirtualMachineError

Java provides exception handling with five keywords:

- 1. try,**
- 2. catch,**
- 3. throw,**
- 4. throws, and**
- 5. finally.**

try block

Program statements that can raise exceptions should be written within a try block. If an exception occurs within the try block, it is thrown to corresponding catch block.

catch block

We can catch the exception raised in try and handle the flow of control without letting it to terminate abruptly because of exception. We can provide any useful information to user regarding the exception.

A try block can be followed by multiple catch blocks.

throw

We can throw the exceptions manually either predefined or user defined, by using the keyword throw.

throws

When an exception causing method wants to return the Exception to its calling method it can be thrown by a throws clause and calling method should handle this exception.

finally

The finally block follows a try block or a catch block. A finally block of code always executes, irrespective of occurrence of an Exception. Using a finally block allows you to run any clean up statements that you want to execute like closing files, session or network connections

Example

:

consider the following statement which leads to

- Arithmetic Exception -divide by zero Exception

```
class TryCatchFinallyExample
{
    public static void main(String[] args)
    {
        System.out.println("start of the code");
        try
        {
            System.out.println("mid of the code");
            int data=50/0; //may throw exception
            System.out.println("rest of the code");
        }

        catch(ArithmeticException e)
        {
            System.out.println(e);
        }
        finally
        {
            System.out.println("End of the program block");
        }
    }
}
```

user defined Exceptions/ custom exceptions

User Defined Exception or custom exception is creating your own exception class and raising that exception using '**throw**' keyword. custom exception class can be created by extending the class Exception.

In the below example problem when amount required to with drawl exceeds the balance we are going to raise a user define exception called OutOfBalanceException

```
class Main
{
    public static void main(String args[])
    {
        Operations obj = new Operations();
        try
        {
            obj.withdrawl(20000);
        }
        catch(OutOfBalanceException e)
        {
            System.out.println(e);
        }
    }
}
```

User defined exception class

```
class OutOfBalanceException extends Exception  
{  
    public String toString()  
    {  
        return "insufficient funds can not perform  
        withdrawl";  
    }  
}
```


class Operations

```
{  
static int balance = 10000;  
void withdrawl(int req)throws OutOfBalanceException  
{  
if(balance < req )  
{  
throw new OutOfBalanceException();  
}  
else  
{  
balance = balance - req;  
}  
}  
}
```

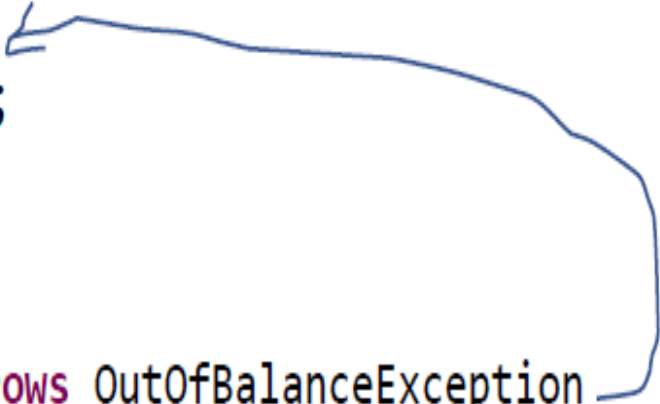
To raise the user defined exception we can use throw keyword

```
throw new OutOfBalanceException();
```

here the raised exception is transferred to the caller method using throws keyword, the exception should be handled there

```
try {  
    obj.withdrawl(20000);  
}
```

```
void withdrawl(int req) throws OutOfBalanceException  
{  
    throw new OutOfBalanceException();  
}
```



THANK YOU

