

Core java

What is Java?

Java is a popular programming language, created in 1995.

It is owned by Oracle, and more than 3 billion devices run Java. It is used for:
Mobile applications (specially Android apps)
Desktop applications
Web applications
Web servers and application servers
Games
Database connection

Why Use Java?

Java works on different platforms (Windows, Mac, Linux, RaspberryPi, etc.)
It is one of the most popular programming language in the world
It is easy to learn and simple to use

Yess InfoTech Pvt. Ltd.

Transforming Career

It is open-source and free

It is secure, fast and powerful

It has a huge community support (tens of millions of developers)

Java is an object oriented language which gives a clear structure to programs and allows code to be reused, lowering development costs

Java Syntax.

In the previous chapter, we created a Java file called Main.java, and we used the following code to print "Hello World" to the screen:

main.java

```
public class Main {  
  
    public static void main(String[] args) {  
  
        System.out.println("Hello World");  
    }  
}
```

Yess InfoTech Pvt. Ltd.

Transforming Career

Example explained

Every line of code that runs in Java must be inside a class. In our example, we named the class Main. A class should always start with an uppercase first letter.

Note: Java is case-sensitive: "MyClass" and "myclass" has different meaning.

The main Method

The main() method is required and you will see it in every Java program:

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

Any code inside the main() method will be executed. You don't have to understand the keywords before and after main. You will get to know them bit by bit while reading this tutorial.

For now, just remember that every Java program has a class name which must match the filename, and that every program must contain the main() method.

`System.out.println()`

Inside the main() method, we can use the `println()` method to print a line of text to the screen:

```
public static void main(String[] args) {System.out.println("Hello World");}
```

Java Comments

Comments can be used to explain Java code, and to make it more readable. It can also be used to prevent execution when testing alternative code.

Single-line Comments

Single-line comments start with two forward slashes (`//`).

Any text between `//` and the end of the line is ignored by Java (will not be executed). This example uses a single-line comment before a line of code:

```
// This is a comment System.out.println("Hello World")
```

Java Multi-line Comments

Multi-line comments start with /* and ends with */. Any text between /* and */ will be ignored by Java.



Yess InfoTech Pvt. Ltd.

Transforming Career

This example uses a multi-line comment (a comment block) to explain the code:

```
/* The code below will print the words Hello World to the screen, and it is amazing
 */ System.out.println("Hello World");
```

Java Variables

Variables are containers for storing data values.

In Java, there are different types of variables, for example:

String - stores text, such as "Hello". String values are surrounded by double quotes
int - stores integers (whole numbers), without decimals, such as 123 or -123

Yess InfoTech Pvt. Ltd.

Transforming Career

float - stores floating point numbers, with decimals, such as 19.99 or -19.99

char - stores single characters, such as 'a' or 'B'. Char values are surrounded by single quotes

boolean - stores values with two states: true or false

Java Data Types

As explained in the previous chapter, a variable in Java must be a specified data type:

```
int myNum = 5; // Integer (whole number)
float myFloatNum = 5.99f; // Floating point number
char myLetter = 'D'; // Character
boolean myBool = true; // Boolean
String myText = "Hello"; // String
```

Data types are divided into two groups:

Primitive data types - includes byte, short, int, long, float, double, boolean and char. **Non-primitive data types** - such as String, Arrays and Classes (you will learn more about these in a later chapter)

Primitive Data Types

A primitive data type specifies the size and type of variable values, and it has no additional methods.

There are eight primitive data types in Java:

Data TypeSize Description



Yess InfoTech Pvt. Ltd.

Transforming Career

byte 1 byte Stores whole numbers from -128 to 127

short 2 bytes Stores whole numbers from -32,768 to 32,767

int 4 bytes Stores whole numbers from -2,147,483,648 to 2,147,483,647

long 8 bytes Stores whole numbers from -9,223,372,036,854,775,808 to 9,223,372,036,854,775,807

float 4 bytes Stores fractional numbers. Sufficient for storing 6 to 7 decimal digits

double 8 bytes Stores fractional numbers. Sufficient for storing 15 decimal digits

boolean 1 bit Stores true or false values

char 2 bytes Stores a single character/letter or ASCII values

Numbers



Yess InfoTech Pvt. Ltd.

Transforming Career

Primitive number types are divided into two groups:



Yess InfoTech Pvt. Ltd.

Transforming Career

Integer types stores whole numbers, positive or negative (such as 123 or -456), without decimals. Valid types are byte, short, int and long. Which type you should use, depends on the numeric value.

Floating point types represents numbers with a fractional part, containing one or more decimals. There are two types: float and double.

Integer Types

Byte

The byte data type can store whole numbers from -128 to 127. This can be used instead of int or other integer types to save memory when you are certain that the value will be within -128 and 127:

```
byte myNum = 100; System.out.println(myNum);  
Short
```

The short data type can store whole numbers from -32768 to 32767:

```
short myNum  
= 5000;  
System.out.println(myNum);
```

Int

The int data type can store whole numbers from -2147483648 to 2147483647. In general, and in our tutorial, the int data type is the preferred data type when we create variables with a numeric value.

```
int myNum = 100000; System.out.println(myNum);
```

Long

The long data type can store whole numbers from -9223372036854775808 to 9223372036854775807. This is used when int is not large enough to store the value. Note that you should end the value with an "L":

```
long myNum = 15000000000L; System.out.println(myNum);
```

Floating Point Types

You should use a floating point type whenever you need a number with a decimal, such as 9.99 or 3.14515.

Float

The float data type can store fractional numbers from 3.4e-038 to 3.4e+038. Note that you should end the value with an "f":



Yess InfoTech Pvt. Ltd.

Transforming Career

```
float myNum = 5.75f; System.out.println(myNum);
```

Double

The double data type can store fractional numbers from $1.7e-308$ to $1.7e+308$. Note that you should end the value with a "d":

```
double myNum = 19.99d; System.out.println(myNum);
```

Booleans

A boolean data type is declared with the boolean keyword and can only take the values true or false:

```
boolean isJavaFun = true; boolean isFishTasty = false;  
System.out.println(isJavaFun); // Outputs true  
System.out.println(isFishTasty); // Outputs false
```

Characters

The char data type is used to store a single character. The character must be surrounded by single quotes, like 'A' or 'c':

```
char myGrade = 'B';
```

Strings

The String data type is used to store a sequence of characters (text). String values must be surrounded by double quotes:

```
System.out.println(myGrade); String greeting = "Hello World";  
System.out.println(greeting);
```

Non-Primitive Data Types

Non-primitive data types are called reference types because they refer to objects. The main difference between primitive and non-primitive data types are:

Primitive types are predefined (already defined) in Java.

Non-primitive types are created by the programmer and is notdefined by Java (except for String).

Non-primitive types can be used to call methods to perform certainoperations, while primitive types cannot.

A primitive type has always a value, while non-primitive types canbe null.

A primitive type starts with a lowercase letter, while non-primitivetypes starts with an uppercase letter.

The size of a primitive type depends on the data type, whilenon-primitive types have all the same size.

Java Type Casting

Type casting is when you assign a value of one primitive data type toanother type.
In Java, there are two types of casting:

Widening Casting (automatically) - converting a smaller type to alarger type size
byte -> short -> char -> int -> long -> float ->double

Narrowing Casting (manually) - converting a larger type to a smallersize type
double -> float -> long -> int -> char -> short ->byte

Java Operators

Operators are used to perform operations on variables and values.

In the example below, we use the + operator to add together two values:
`int x = 100 + 50;`

Although the + operator is often used to add together two values, like inthe example above, it can also be used to add together a variable and a value, or a variable and another variable:

```
int sum1 = 100 + 50; // 150 (100 + 50)
```

```
int sum2 = sum1 + 250; // 400 (150 + 250)
```

```
int sum3 = sum2 + sum2; // 800 (400 + 400)
```

Java divides the operators into the following groups:

- Arithmetic operators
- Assignment operators
- Comparison operators
- Logical operators
- Bitwise operators

Yess InfoTech Pvt. Ltd.

Transforming Career

Arithmetic Operators

Arithmetic operators are used to perform common mathematical operations.

Operator	Name	Description	Example	Try it
+	Addition	Adds together two values	$x + y$	Try it »
-	Subtraction	Subtracts one value from another	$x - y$	Try it »
*	Multiplication	Multiplies two values	$x * y$	Try it »
/	Division	Divides one value by another	x / y	Try it »

% Modulus Returns the division remainder
x % y Try it
»

++ Increment Increases the value of a variable by 1
++x Try it
»

-- Decrement Decreases the value of a variable by 1
--x Try it
»

Yess InfoTech Pvt. Ltd.

Transforming Career

Java Assignment Operators

Assignment operators are used to assign values to variables.

In the example below, we use the assignment operator (=) to assign the value 10 to a variable called x:

```
int x = 10;
```

The addition assignment operator (+=) adds a value to a variable:

```
int x = 10;
```

```
x += 5;
```

A list of all assignment operators:

Operator Example Same As Try it

= `x = 5` `x = 5` Try it

»

Yess InfoTech Pvt. Ltd.

+= `x += 3` `x = x + 3` Try it

»

Transforming Career

-= `x -= 3` `x = x - 3` Try it

»

*= `x *= 3` `x = x * 3` Try it

`/= x /= 3 x = x / 3 Try it`



Yess InfoTech Pvt. Ltd.

Transforming Career

```
%=  x %= 3      x = x % 3  Try it
&=  x &= 3      x = x & 3  Try it
»
```

```
|=  x |= 3      x = x | 3  Try it
»
```

```
^=  x ^= 3      x = x ^ 3  Try it
»
```

```
>>= x >>= 3   x = x >> 3 Try it
»
```

```
<<= x <<= 3   x = x << 3
```

Java Comparison Operators

Comparison operators are used to compare two values:

Operator Name	Example	Try it
---------------	---------	--------

==	Equal to	x == y	Try it
----	----------	--------	--------

»

!=	Not equal	x != y	Try it
----	-----------	--------	--------

»

>	Greater than	x > y	Try it
---	--------------	-------	--------

»

< Less than x < y Try it

»



Yess InfoTech Pvt. Ltd.

Transforming Career

\geq Greater than or equal to

$x \geq y$ Try it

»

\leq Less than or equal to $x \leq y$ Try it



Java Logical Operators

Logical operators are used to determine the logic between variables or values

Yess InfoTech Pvt. Ltd.

Transforming Career

Operator	Name	Description	Example	Try it
<code>&&</code>	Logical and statements are true	Returns true if both	<code>x < 5 && x < 10</code>	Try it »
<code> </code>	Logical or is true	Returns true if one of the statements is true	<code>x < 5 x < 4</code>	Try it »
<code>!</code>	Logical not if the result is true	Reverse the result, returns false if the result is true	<code>!(x < 5 && x < 10)</code>	Try it »

Java Conditions and If Statements

Java supports the usual logical conditions from mathematics:

Less than: `a < b`

Less than or equal to: `a <= b`

Greater than: `a > b`

Greater than or equal to: `a >= b`

Equal to: `a == b`

Not Equal to: `a != b`

You can use these conditions to perform different actions for different decisions. Java has the following conditional statements:

Use `if` to specify a block of code to be executed, if a specified condition is true

Use else to specify a block of code to be executed, if the same condition is false
Use else if to specify a new condition to test, if the first condition is false
Use switch to specify many alternative blocks of code to be executed

The if Statement

Use the if statement to specify a block of Java code to be executed if a condition is true.

Syntax

```
if (condition) {  
    // block of code to be executed if the condition is true  
}
```

Java Switch Statements

Use the switch statement to select one of many code blocks to be executed.

SYNTAX

Transforming Career

```
switch(expression) {case x:  
// code block  
break;  
case y:  
  
// code block  
break; default:  
// code block  
  
}
```

This is how it works:

The switch expression is evaluated once.

The value of the expression is compared with the values of each case.

If there is a match, the associated block of code is executed.

The break and default keywords are optional, and will be described later in this chapter.



Yess InfoTech Pvt. Ltd.

Transforming Career

The example below uses the weekday number to calculate the weekdayname:

Example

```
int day = 4; switch (day) {case 1:  
System.out.println("Monday");break;  
case 2: System.out.println("Tuesday");break;  
case 3: System.out.println("Wednesday");break;  
case 4: System.out.println("Thursday");
```



Yess InfoTech Pvt. Ltd.

Transforming Career

break;case 5:

```
System.out.println("Friday");break;  
case 6: System.out.println("Saturday");break;  
case 7: System.out.println("Sunday");break;  
}
```

Loops

Loops can execute a block of code as long as a specified condition is reached. Loops are handy because they save time, reduce errors, and they make code more readable.

Yess InfoTech Pvt. Ltd.

Transforming Career

Java While Loop

The while loop loops through a block of code as long as a specified condition is true:

Syntax

```
while (condition) {  
    // code block to be executed  
}
```

In the example below, the code in the loop will run, over and over again, as long as a variable (i) is less than 5:

Example

```
int i = 0; while (i < 5) {  
    System.out.println(i);
```

Yess InfoTech Pvt. Ltd.

Transforming Career

```
i++;
```

```
}
```

Note: Do not forget to increase the variable used in the condition, otherwise the loop will never end!

The Do/While Loop

The do/while loop is a variant of the while loop. This loop will execute the code block once, before checking if the condition is true, then it will repeat the loop as long as the condition is true.

Syntax

```
do {  
    // code block to be executed  
}
```

Yess InfoTech Pvt. Ltd.

Transforming Career

```
}
```

```
while (condition);
```

The example below uses a do/while loop. The loop will always be executed at least once, even if the condition is false, because the codeblock is executed before the condition is tested:

```
do { System.out.println(i); i++;  
}
```

```
while (i < 5);
```

Java For Loop

When you know exactly how many times you want to loop through a block of code, use the for loop instead of a while loop:

```
for (statement 1; statement 2; statement 3) {
```

```
// code block to be executed
```

```
}
```

Statement 1 is executed (one time) before the execution of the code block. Statement 2 defines the condition for executing the code block.

Statement 3 is executed (every time) after the code block has been executed.

The example below will print the numbers 0 to 4:

Example



Yess InfoTech Pvt. Ltd.

Transforming Career

```
for (int i = 0; i < 5; i++) {System.out.println(i);}
```



Yess InfoTech Pvt. Ltd.

Transforming Career

}

Example explained

Statement 1 sets a variable before the loop starts (int i = 0).

Statement 2 defines the condition for the loop to run (i must be less than 5). If the condition is true, the loop will start over again, if it is false, the loop will end.

Statement 3 increases a value (i++) each time the code block in the loop has been executed.

OOPS IN JAVA

What is Object-Oriented Programming ?

The object-oriented programming model revolves around the concept of Objects.

What is an Object?

An object is an instance of a Class. It contains properties and functions. They are like real-world objects. For example, your car, house, laptop, etc. are all objects. They have some specific properties and methods to perform some action.

What is a Class?

The Class defines the blueprint of Objects. They define the properties and functionalities of the objects. For example, Laptop is a class and your laptop is an instance of it.

Core OOPS concepts are:

Inheritance

Inheritance is a mechanism in which one class acquires the property of another class. For example, a child inherits the traits of his/her parents. With inheritance, we can reuse the fields and methods of the existing class. Hence, inheritance facilitates Reusability and is an important concept of OOPs.

JAVA INHERITANCE is a mechanism in which one class acquires the property of another class. In Java, when an "Is-A" relationship exists between two classes, we use Inheritance. The parent class is called a super class and the inherited class

is called a subclass. The keyword extends is used by the sub class to inherit the features of super class.



Yess InfoTech Pvt. Ltd.

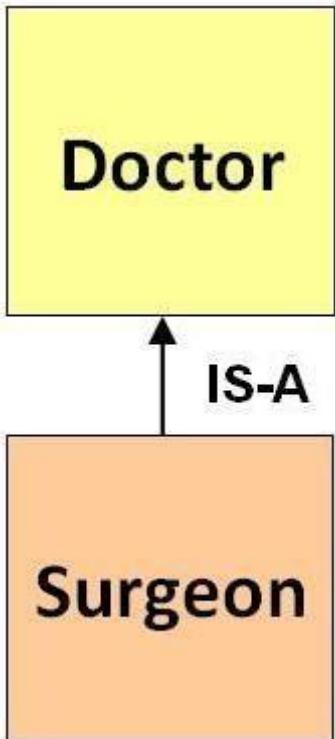
Transforming Career

Inheritance is important since it leads to the reusability of code.

Java Inheritance Syntax:

```
class subClass extends superClass  
{  
    //methods and fields  
}
```

Java Inheritance Example



Ye FoTech Pvt. Ltd.
Transforming Career

```
class Doctor {  
void Doctor_Details() {  
    System.out.println("Doctor Details...");  
}  
}  
}
```

```
class Surgeon extends Doctor { void Surgeon_Details() {  
    System.out.println("Surgeon Detail...");  
}  
}  
}
```

```
public class Hospital {  
public static void main(String args[]) {  
    Surgeon s = new Surgeon();  
    s.Doctor_Details();  
    s.Surgeon_Details();  
}  
}
```

Super Keyword

The super keyword is similar to "this" keyword.

The keyword super can be used to access any data member or methods of the parent class.

Super keyword can be used at variable, method and constructor level.

Syntax:

```
super.<method-name>();
```

What is this Keyword in Java?

this keyword in Java is a reference variable that refers to the current object of a method or a constructor. The main purpose of using this keyword in Java is to remove the confusion between class attributes and parameters that have same names.

Following are various uses of 'this' keyword in Java:

It can be used to refer instance variable of current class

It can be used to invoke or initiate current class constructor



Yess InfoTech Pvt. Ltd.

Transforming Career

It can be passed as an argument in the method call
It can be passed as argument in the constructor call
It can be used to return the current class instance

What is Constructor in Java?

CONSTRUCTOR is a special method that is used to initialize a newly created object and is called just after the memory is allocated for the object. It can be used to initialize the objects to desired values or default values at the time of object creation. It is not mandatory for the coder to write a constructor for a class.

If no user-defined constructor is provided for a class, compiler initializes member variables to its default values.

numeric data types are set to 0

char data types are set to null character('\'0')

reference variables are set to nullIn this tutorial, you will learn-

Rules for creating a Constructor

Yess InfoTech Pvt. Ltd.

Transforming Career

Constructor Overloading in Java

Constructor Chaining

Rules for creating a Java Constructor

It has the **same name** as the class

It should not return a value not even **void**

Example 1: Create your First Constructor in Java

Step 1) Type the following constructor program in Java editor.

```
class Demo{  
    int value1;  
    int value2;  
    Demo(){  
        value1 = 10;  
        value2 = 20;  
        System.out.println("Inside Constructor");  
    }  
}
```

```
    public void display(){  
        System.out.println("Value1 === "+value1);  
        System.out.println("Value2 === "+value2);  
    }
```

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

```
    Demo d1 = new Demo();  
    d1.display();
```

```
}
```

```
}
```

Step 2) Save , Run & Compile the constructor program in Java and observe the output.

Output:

```
Inside Constructor Value1 === 10  
Value2 === 20
```



Constructor Overloading in Java

Java Constructor overloading is a technique in which a class can have any number of constructors that differ in parameter list. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.

Examples of valid constructors for class Account are

```
Account(int a); Account (int a,int b); Account (String a,int b);
```

Example 2: To understand Constructor Overloading in Java

Step 1) Type the code in the editor.

```
class Demo{  
    int value1;  
    int value2;  
    /*Demo(){  
        value1 = 10;  
        value2 = 20;
```

Xocs InfoTech Pvt. Ltd.
Transforming Career

```
System.out.println("Inside 1st Constructor");
} */
Demo(int a){
    value1 = a;
    System.out.println("Inside 2nd Constructor");
}
Demo(int a,int b){
    value1 = a;
    value2 = b;
    System.out.println("Inside 3rd Constructor");
}
public void display(){
    System.out.println("Value1 === "+value1);
    System.out.println("Value2 === "+value2);
}
public static void main(String args[]){
    Demo d1 = new Demo();
    Demo d2 = new Demo(30);
    Demo d3 = new Demo(30,40);
    d1.display();
    d2.display();
    d3.display();
}
```

Step 2) Save, Compile & Run the Code.

Step 3) Error = ?. Try and debug the error before proceeding to next step of Java constructor overloading

Step 4) Every class has a default Constructor in Java. Default overloaded constructor Java for **class Demo** is **Demo()**. In case you do not provide this constructor the compiler creates it for you and initializes the variables to default values. You may

choose to override this default constructor and initialize variables to your desired values as shown in Example 1.

But if you specify a parametrized constructor like Demo(int a), and want to use the default constructor Java Demo(), it is mandatory for you to specify it.



Yess InfoTech Pvt. Ltd.

Transforming Career

In other words, in case your overloading constructor in Java is overridden, and you want to use the default constructor Java, its need to be specified.

Step 5) Uncomment line # 4-8. Save, Compile & Run the code.

What is Polymorphism in Java?

Polymorphism in Java occurs when there are one or more classes or objects related to each other by inheritance. It is the ability of an object to take many forms. Inheritance lets users inherit attributes and methods, and polymorphism uses these methods to

Yess InfoTech Pvt. Ltd.

Transforming Career

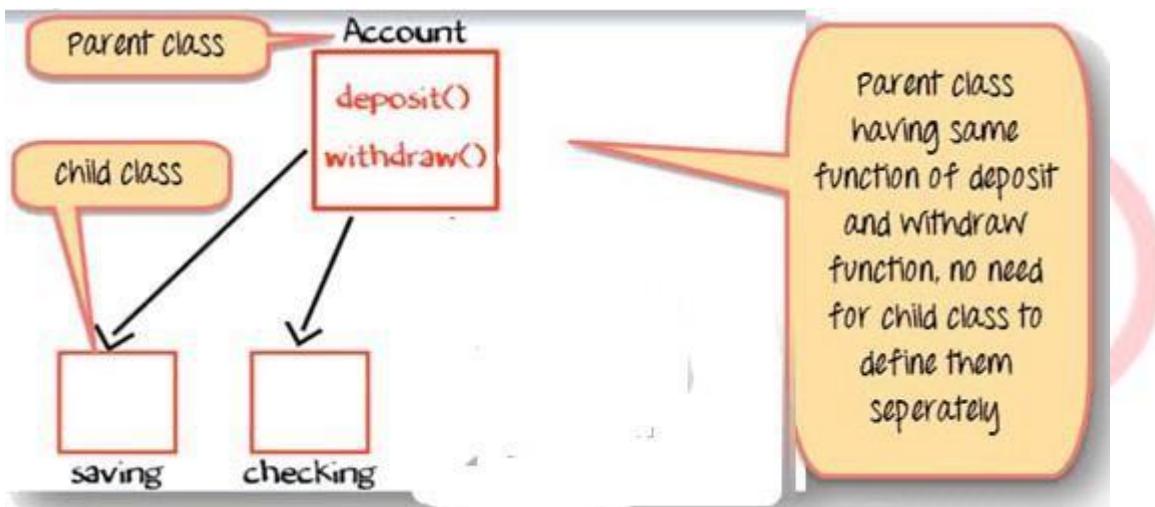
perform different tasks. So, the goal is communication, but the approach is different. For example, you have a smartphone for communication. The communication mode you choose could be anything. It can be a call, a text message, a picture message, mail, etc. So, the goal is common that is communication, but their approach is different. This is called **Polymorphism**. Now, we will learn Polymorphism in Java with

Java Polymorphism in OOPs with Example

We have one parent class, ‘Account’ with function of deposit and withdraw. Account has 2 child classes

The operation of deposit and withdraw is same for Saving and Checking accounts. So the inherited methods from Account class will work.

Yess InfoTech Pvt. Ltd.
Transforming Career



Java Polymorphism Example

Change in Software Requirement

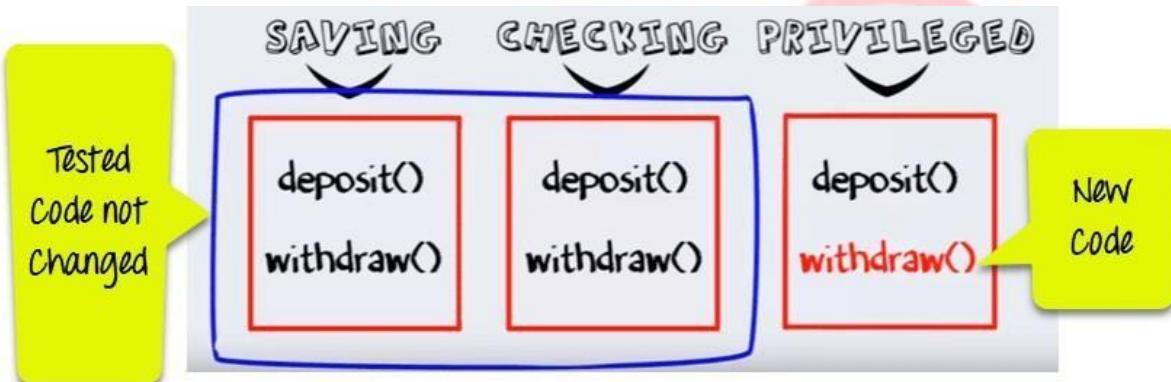
There is a change in the requirement specification, something that is so common in the software industry. You are supposed to add functionality privileged Banking Account with Overdraft Facility.

For a background, overdraft is a facility where you can withdraw an amount more than available in your account.

Yess InfoTech Pvt. Ltd.

Transforming Career

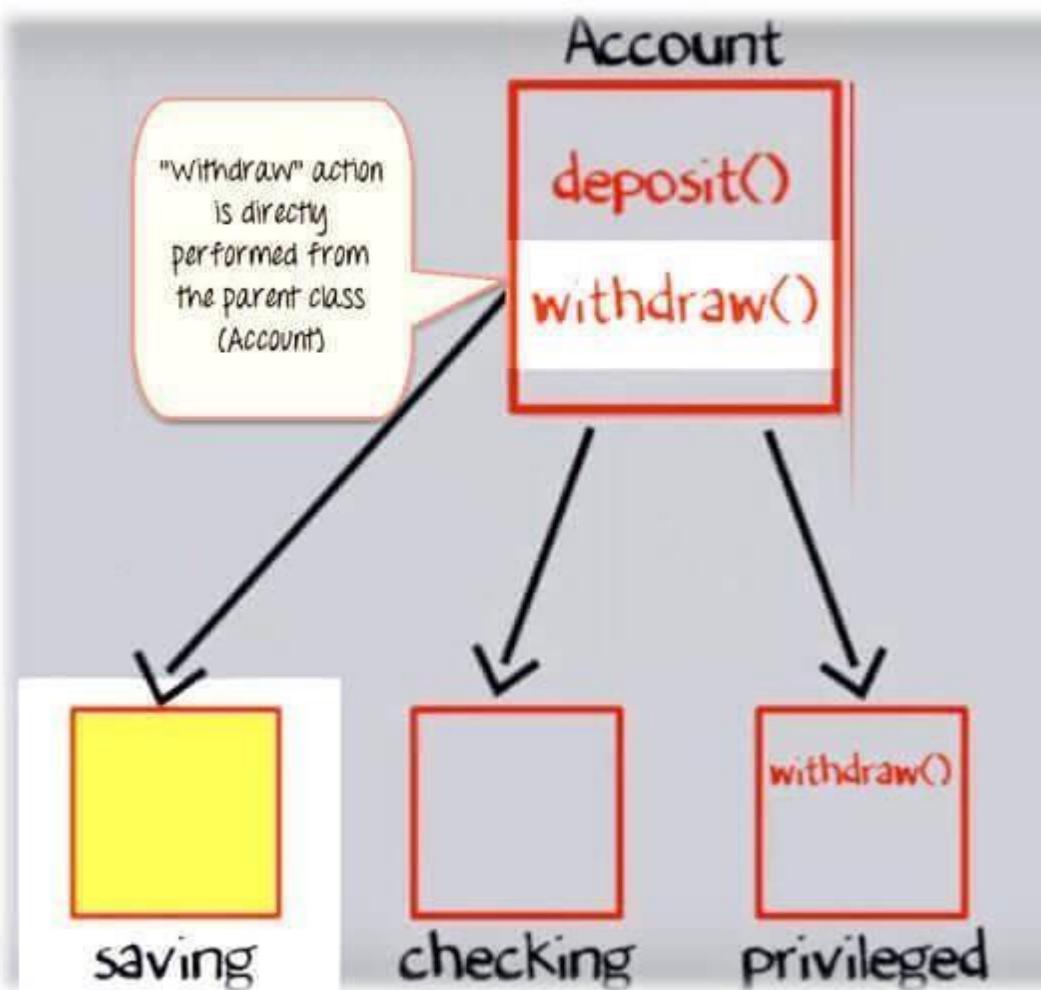
So, withdraw method for privileged needs to implemented afresh. But you do not change the tested piece of code in Savings and Checking account. This is advantage of OOPS



Step 1) Such that when the "withdrawn" method for saving account is called a method from parent account class is executed.

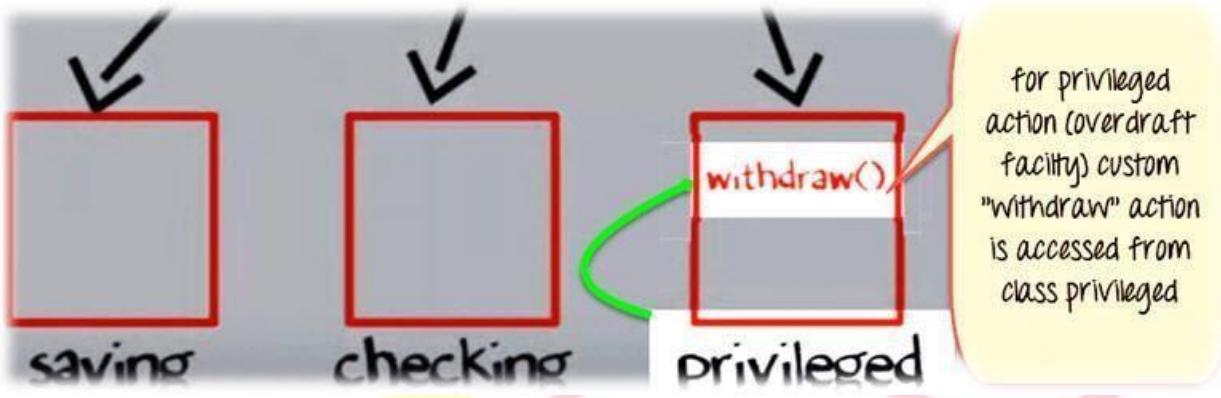
Yess InfoTech Pvt. Ltd.

Transforming Career



Step 2) But when the "Withdraw" method for the privileged account (overdraft facility) is called withdraw method defined in the privileged class is executed. This is **Polymorphism in OOPs**.

Transforming Career



Method Overriding in Java

Method Overriding is redefining a super class method in a subclass.

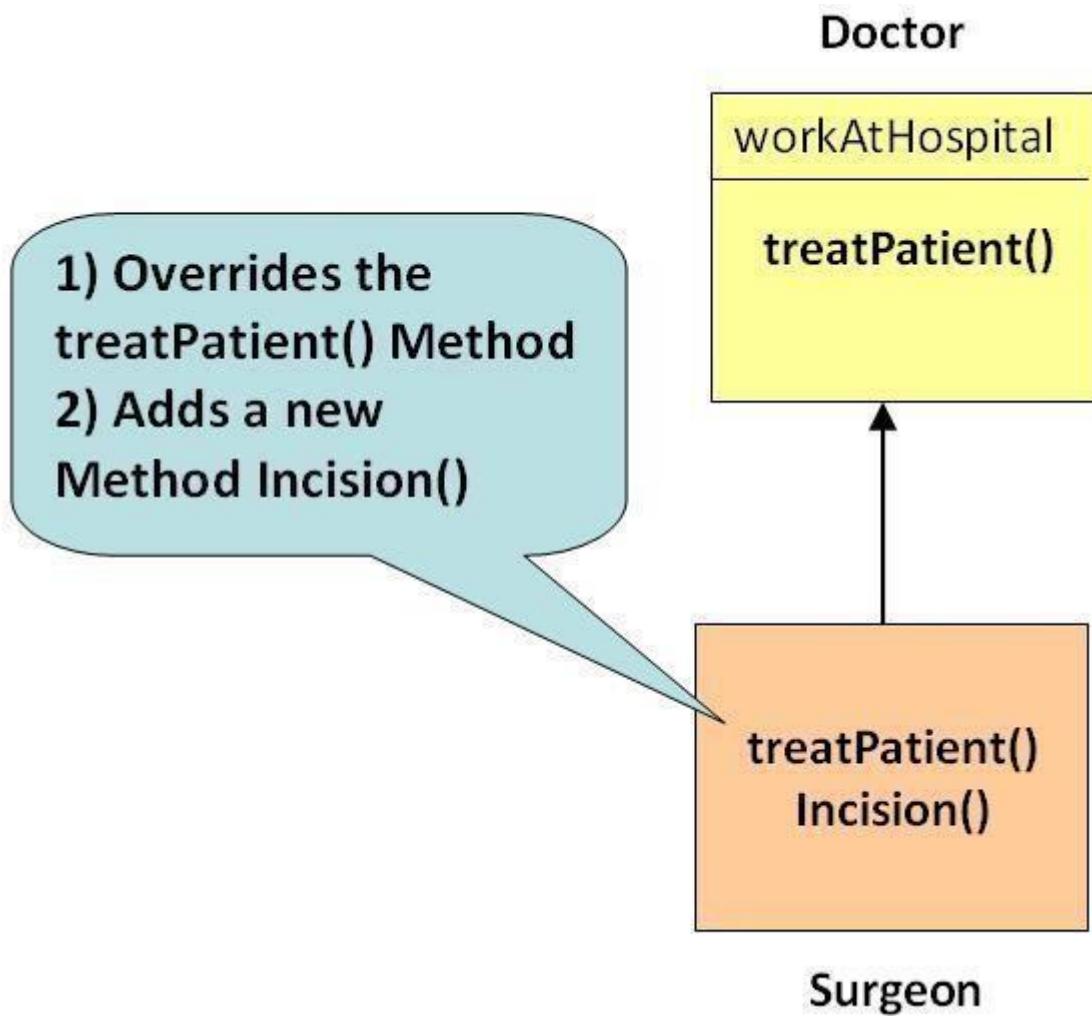
Rules for Method Overriding

The method signature i.e. method name, parameter list and return type have to match exactly.

The overridden method can widen the accessibility but not narrow it, i.e. if it is private in the base class, the child class can make it public but not vice versa.

Yess InfoTech Pvt. Ltd.

Transforming Career



Example

```
class Doctor{  
    public void treatPatient(){  
        // treatPatient method  
    }  
}
```

```
}
```

```
class Surgeon extends Doctor{
```

```
    public void treatPatient(){
```

```
        // treatPatient method
```

```
}
```

```
}
```

```
Class run{
```

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

```
        Doctor doctorObj = new Doctor()
```

```
        // treatPatient method in class Doctor will be executed
```

```
doctorObj.treatPatient();
```

```
Surgeon surgeonObj = new Surgeon();
```

```
// treatPatient method in class Surgeon will be executed
```

```
surgeonObj.treatPatient();
```

```
}
```

```
}
```

Difference between Overloading and Overriding

Yess InfoTech Pvt. Ltd.

Transforming Career

Method Overloading

Method Overriding

Method overriding is when one of the methods in the same class is redefined in the subclass, where more than one method exists. In this case, the signature of the methods have the same name but different method remains the same.

signatures.

Ex:

Ex:

```
class X{
    public int sum(){
        // some code
    }
}
```

```
void sum (int a , int b); void sum }
```

```
(int a , int b, int c);void sum (float a,}
```

```
double b);
```

```
class Y extends X{
    public int sum(){
        //overridden method
    }
}
```

Difference between Static & Dynamic Polymorphism

Static Polymorphism in Java is a type of polymorphism that collects the

information for calling a method at compilation time, whereas Dynamic



Yess InfoTech Pvt. Ltd.

Transforming Career

Polymorphism is a type of polymorphism that collects the information for calling a method at runtime.



Yess InfoTech Pvt. Ltd.

Transforming Career

Static Polymorphism

Dynamic Polymorphism

It relates to methodoverloading. It relates to methodoverriding.

Errors, if any, are resolved at compile time. Since the code is not executed during compilation, hence the name static.

Ex:

```
void sum (int a , int b); void sum (float dynamic.
```

```
a, double b);
```

```
int sum (int a, int b); //compiler gives
```

error. *Access Inherited Methods* //reference of parent pointing to child object

```
Doctor obj = new Surgeon();
```

```
// method of child called obj.treatPatient();
```

Abstraction

Abstraction is the concept of hiding the internal details and describing things in simple terms. For example, a method that adds two integers. The internal processing of the method is hidden from the outer world. There are many ways to achieve abstraction in object-oriented programming, such as encapsulation and inheritance. A Java program is also a great example of abstraction. Here Java takes care of converting simple statements to machine language and hides the inner implementation details from the outer world.

What is Interface in Java?

An **Interface in Java** programming language is defined as an abstract type used to specify the behavior of a class. A Java interface contains static constants and abstract methods. A class can implement multiple interfaces. In Java, interfaces are declared using the interface keyword. All methods in the interface are implicitly public and abstract.

Now, we will learn how to use interface in Java.

Syntax for Declaring Interface

To use an interface in your class, append the keyword "implements" after your class name followed by the interface name.

```
interface {  
//methods  
}
```

Example for Implementing Interface

Now, let's understand interface in Java with example:

```
class Dog implements Pet
```

interface RidableAnimal extends Animal, Vehicle

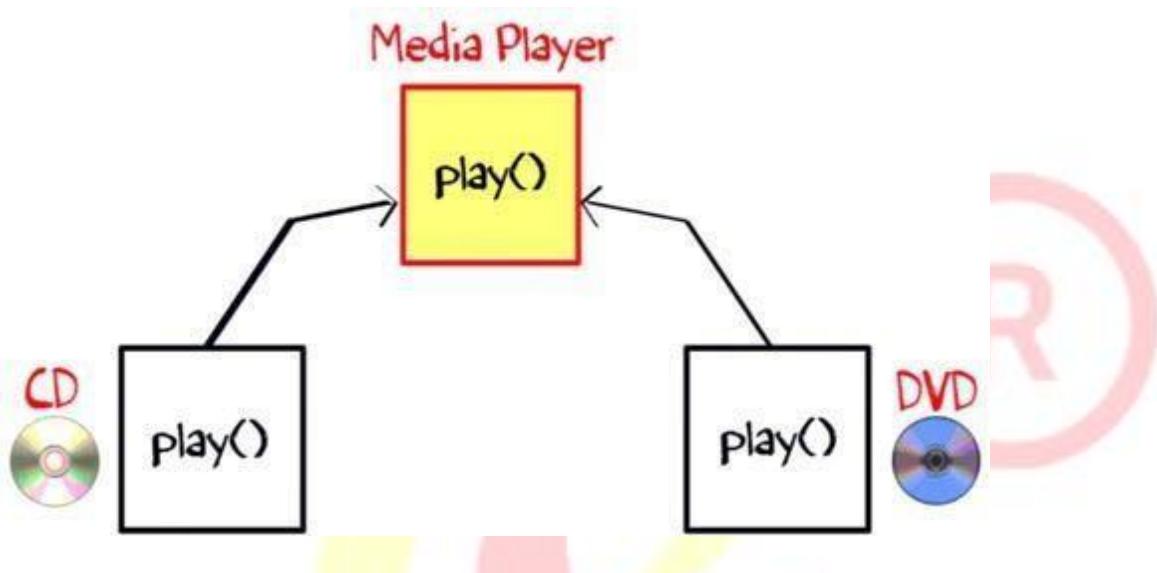
[Click here if the video is not accessible](#)

Why is an Interface required?

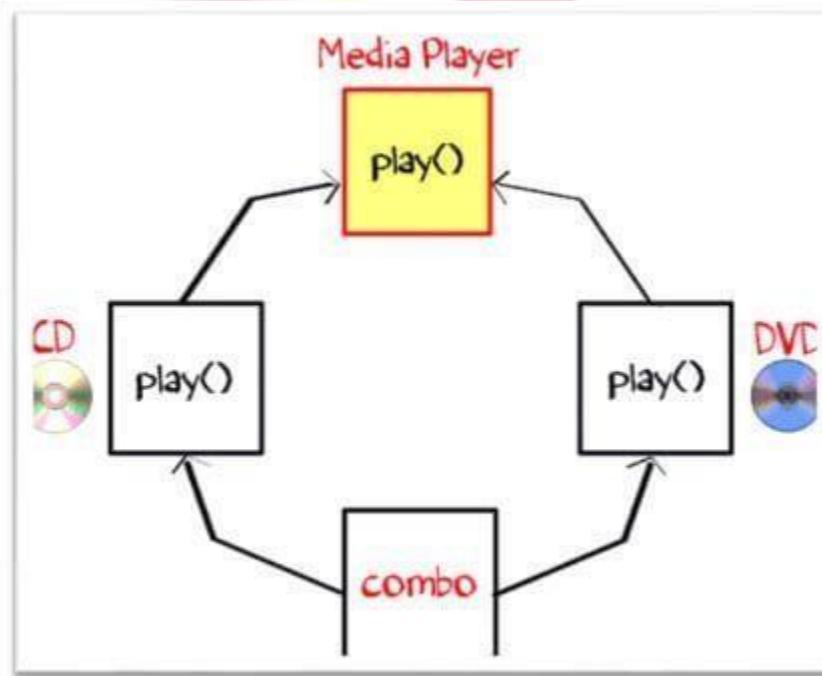
To understand the use of interface in Java better, let see an Java interface example. The class "Media Player" has two subclasses: CD player and DVD player. Each having its unique interface implementation in Java method to play music.

Yess InfoTech Pvt. Ltd.

Transforming Career

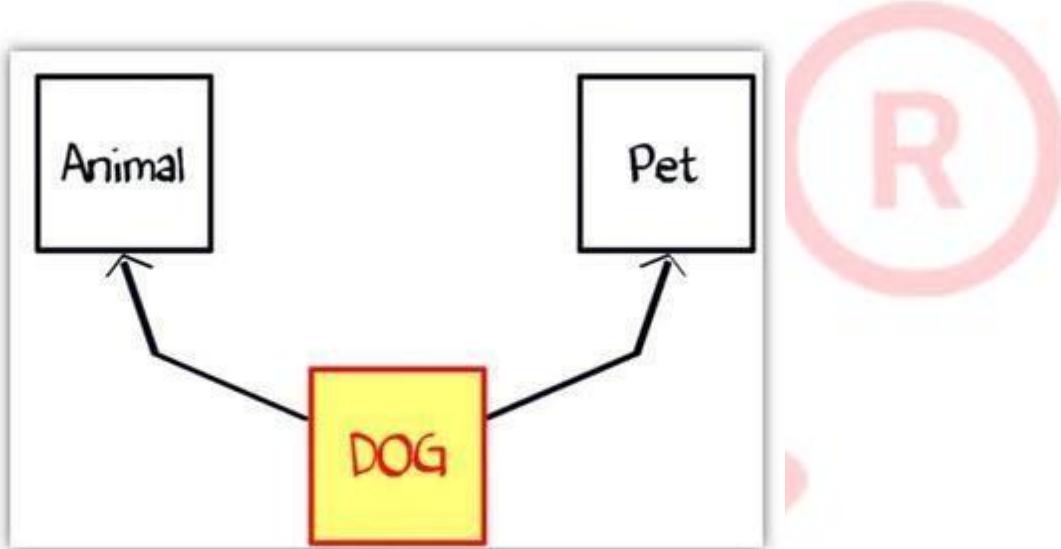


Another class "Combo drive" is inheriting both CD and DVD (see image below). Which play method should it inherit? This may cause serious design issues. And hence, Java does not allow multiple inheritance.



Now let's take another example of Dog.

Suppose you have a requirement where class "dog" inheriting class "animal" and "Pet"(see image below). But you cannot extend two classes in Java. So what would you do?The solution is Interface.



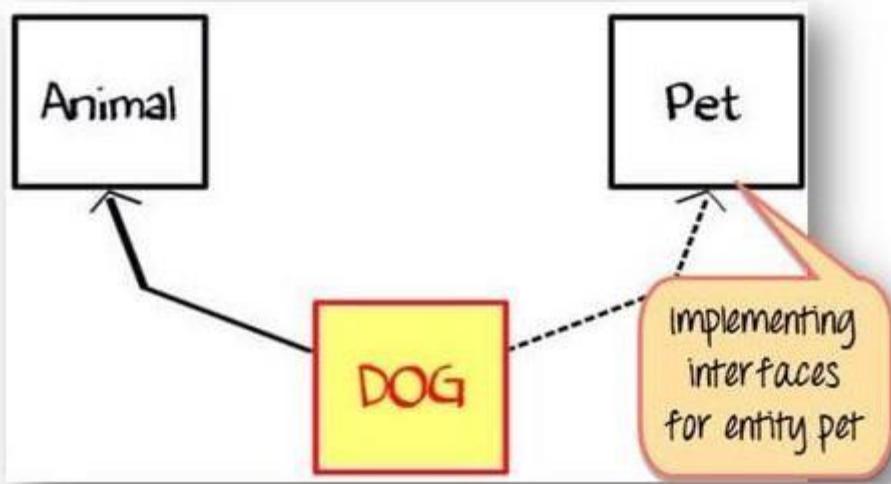
The rulebook for interface says,

A Java implement interface is 100% abstract class and has only abstractmethods.
Class can implement any number of interfaces.

Class Dog can extend to class "Animal" and implement interface as "Pet".

Yess InfoTech Pvt. Ltd.

Transforming Career



Java Interface Example:

Let's understand the below interface program in Java:

Step 1) Copy following code into an editor.

```
interface Pet{  
    public void test();  
}  
class Dog implements Pet{  
    public void test(){  
        System.out.println("Interface Method Implemented");  
    }  
    public static void main(String args[]){  
        Pet p = new Dog();  
        p.test();  
    }  
}
```

```
}
```

Step 2) Save , Compile & Run the code. Observe the Output of the interface in Java program.

Difference between Class and Interface

Class	Interface

In class, you can instantiate variable and create an object. In an interface, you can't instantiate variable and create an object.

Class can contain concrete(with implementation) methods The interface cannot contain concrete(with implementation)methods

The access specifiers used with classes are private, protected and public. In Interface only one specifier is used-Public.

Transforming Career

When to use Interface and Abstract Class?

Use an abstract class when a template needs to be defined for a group of subclasses
Use an interface when a role needs to be defined for other classes, regardless of the inheritance tree of these classes

Must know facts about Interface

A Java class can implement multiple Java Interfaces. It is necessary that the class must implement all the methods declared in the interfaces.

Class should override all the abstract methods declared in the interface

The interface allows sending a message to an object without concerning which classes it belongs.

Class needs to provide functionality for the methods declared in the interface.

All methods in an interface are implicitly public and abstract

An interface cannot be instantiated

An interface reference can point to objects of its implementing classes

An interface can extend from one or many interfaces. A class can extend only one class but implement any number of interfaces

An interface cannot implement another Interface. It has to extend another interface if needed.

An interface which is declared inside another interface is referred as nested interface

At the time of declaration, interface variable must be initialized. Otherwise, the compiler will throw an error.

The class cannot implement two interfaces in java that have methods with same name but different return type.

Summary:

The class which implements the interface needs to provide functionality for the methods declared in the interface

All methods in an interface are implicitly public and abstract

An interface cannot be instantiated

An interface reference can point to objects of its implementing classes

An interface can extend from one or many interfaces. A class can extend only one class but implement any number of interfaces

Encapsulation

Encapsulation is the technique used to implement abstraction in object-oriented programming. Encapsulation is used for access restriction to class members and methods. Access modifier keywords are used for encapsulation in object oriented programming. For example, encapsulation in java is achieved using private, protected and public keywords.

There are four types of Java access modifiers:

Private: The access level of a private modifier is only within the class. It cannot be accessed from outside the class.

Default: The access level of a default modifier is only within the package. It cannot be accessed from outside the package. If you do not specify any access level, it will be the default.

Protected: The access level of a protected modifier is within the package and outside the package through child class. If you do not make the child class, it cannot be accessed from outside the package.

Public: The access level of a public modifier is everywhere. It can be accessed from within the class, outside the class, within the package and outside the package.

Understanding Java Access Modifiers

Access Modifier	within class	within package	outside package by subclass only	outside package
Private	Y	N	N	N
Default	Y	Y	N	N
Protected	Y	Y	Y	N
Public	Y	Y	Y	Y

Java Strings

Strings are used for storing text.



Yess InfoTech Pvt. Ltd.

Transforming Career

A String variable contains a collection of characters surrounded by doublequotes:



Yess InfoTech Pvt. Ltd.

Transforming Career

```
String greeting = "Hello";
```

String Length

A String in Java is actually an object, which contain methods that can perform certain operations on strings. For example, the length of a string can be found with the length() method:

Example

```
String txt = "ABCDEFGHIJKLMNPQRSTUVWXYZ";
```

```
System.out.println("The length of the txt string is: " +txt.length());
```

More String Methods

There are many string methods available, for example toUpperCase() and toLowerCase():

Example

```
String txt = "Hello World";
```

```
System.out.println(txt.toUpperCase()); // Outputs "HELLO WORLD"  
System.out.println(txt.toLowerCase()); // Outputs "hello world"
```

Finding a Character in a String

The indexOf() method returns the index (the position) of the first occurrence of a



Yess InfoTech Pvt. Ltd.

Transforming Career

specified text in a string (including whitespace):

Example

```
String txt = "Please locate where 'locate' occurs!";
System.out.println(txt.indexOf("locate")); // Outputs 7
```

String Concatenation

The + operator can be used between strings to combine them. This is called concatenation:

Example

```
String firstName = "John"; String lastName = "Doe";
System.out.println(firstName + " " + lastName);
```

You can also use the concat() method to concatenate two strings:

Example

```
String firstName = "John "; String lastName = "Doe";
System.out.println(firstName.concat(lastName));
```

What is an Array in Java?

An array refers to a data structure that contains homogeneous elements. This means that all the elements in the array are of the same data type. Let's take an example:

Elements
Index
40
55
23
87
21
11
94

This is an array of seven elements. All the elements are integers and homogeneous.

The green box below the array is called the index, which always starts from zero



Yess InfoTech Pvt. Ltd.

Transforming Career

and goes up to n-1 elements. In this case, as there are seven elements, the index is from zero to six. There are three main features of an array:

Dynamic allocation: In arrays, the memory is created dynamically, which reduces the amount of storage required for the code.

Elements stored under a single name: All the elements are stored under one name. This name is used any time we use an array.

Occupies contiguous location: The elements in the arrays are stored at adjacent positions. This makes it easy for the user to find the locations of its elements.

Advantages of Arrays in Java

Java arrays enable you to access any element randomly with the help of indexes
It is easy to store and manipulate large data sets

Disadvantages of Arrays in Java

The size of the array cannot be increased or decreased once it is declared—arrays have a fixed size

Java cannot store heterogeneous data. It can only store a single type of primitives
Now that we understand what Java arrays are- let us look at how arrays in Java are declared and defined.

Define an Array in Java

Arrays in Java are easy to define and declare. First, we have to define the array.

The syntax for it is:

```
type var-name[];
```

OR

```
type[] var-name;
```

Here, the type is int, String, double, or long. Var-name is the variable name of the array.

Declare an Array in Java

These are the two ways that you declare an array in Java. You can assign values to elements of the array like this:



```
1 public class Apple{  
2     //creating a no-arg constructor  
3     Apple(){  
4         System.out.println("This is a no-arg constructor of the Apple class.");  
5     }  
6     //the main method  
7     public static void main (String args[]){  
8         //creating an object to invoke the constructor  
9         Apple a = new Apple();  
10    }  
11 }  
12 }
```

Execute Mode: Version: Inputs & Arguments

Result

CPU Time: 0.12 seconds. Memory: 10464 kilobytes.

This is a no-arg constructor of the Apple class.

We have declared an array arr of type integer. The size of the array is 5, meaning that it can have five elements. The array is assigned with elements for each of the index positions. We'll run a for loop to print the elements in the array. A counter variable "i" is used to increment the index position after checking if the current index position is less than the length of the array.

```
public static void main(String[] args) {  
    int[] arr;  
    arr = new int[5];  
    arr[0] = 10;  
    arr[1] = 20;  
    arr[2] = 30;  
    arr[3] = 40;  
    arr[4] = 50;  
    for(int i = 0;i<arr.length;i++){  
        System.out.println("Element at index "+i+" is "+arr[i]);  
    }  
}
```

Exception Handling

What is Exception in Java?

Exception in Java is an event that interrupts the execution of program instructions and disturbs the normal flow of program execution. It is an object that wraps an



Yess InfoTech Pvt. Ltd.

Transforming Career

errorevent information that occurred within a method and it is passed to the runtime system. In Java, exceptions are mainly used for indicating different types of error conditions.

There are two types of errors:

Compile time errors
Runtime errors

Compile time errors can be again classified again into two types:

Syntax Errors
Semantic Errors

Syntax Errors Example:

Instead of declaring int a; you mistakenly declared it as in a; for which compiler will throw an error.

Yess InfoTech Pvt. Ltd.
Transforming Career

Example: You have declared a variable int a; and after some lines of code you again declare an integer as int a;. All these errors are highlighted when you compile the code.

Runtime Errors Example

A Runtime error is called an **Exceptions** error. It is any event that interrupts the normal flow of program execution.

Example for exceptions are, arithmetic exception, Nullpointer exception, Divide byzero exception, etc.

Exceptions in Java are something that is out of developers control.

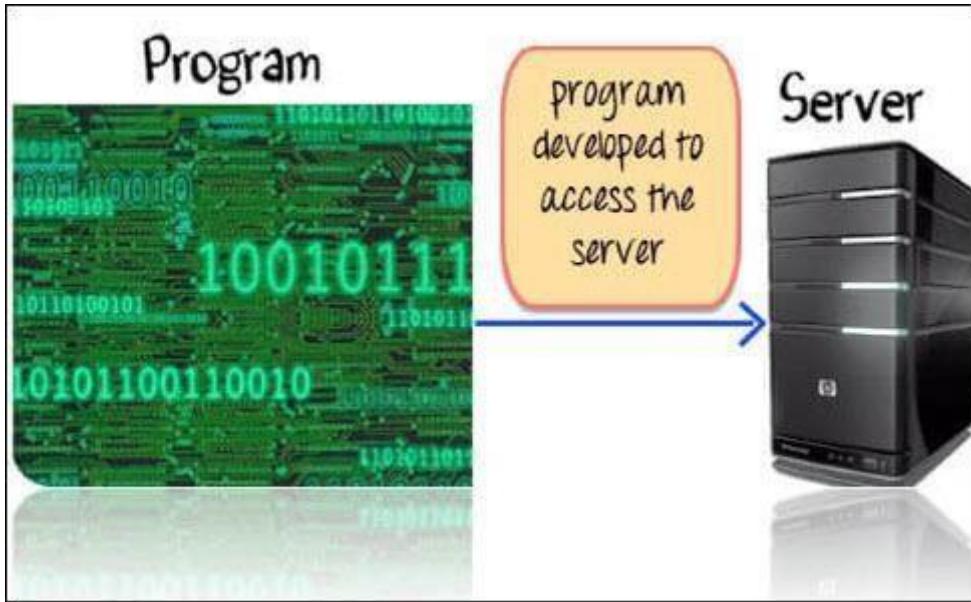
[Click here if the video is not accessible](#)

Why do we need Exception?

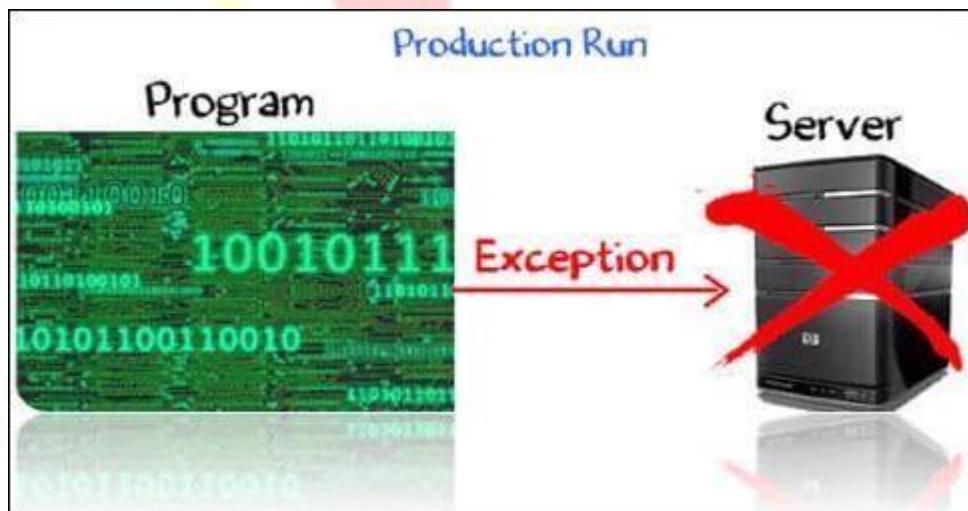
Suppose you have coded a program to access the server. Things worked fine while you were developing the code.

Yess InfoTech Pvt. Ltd.

Transforming Career



During the actual production run, the server is down. When your program tried to access it, an exception is raised.



Yes

Ltd.

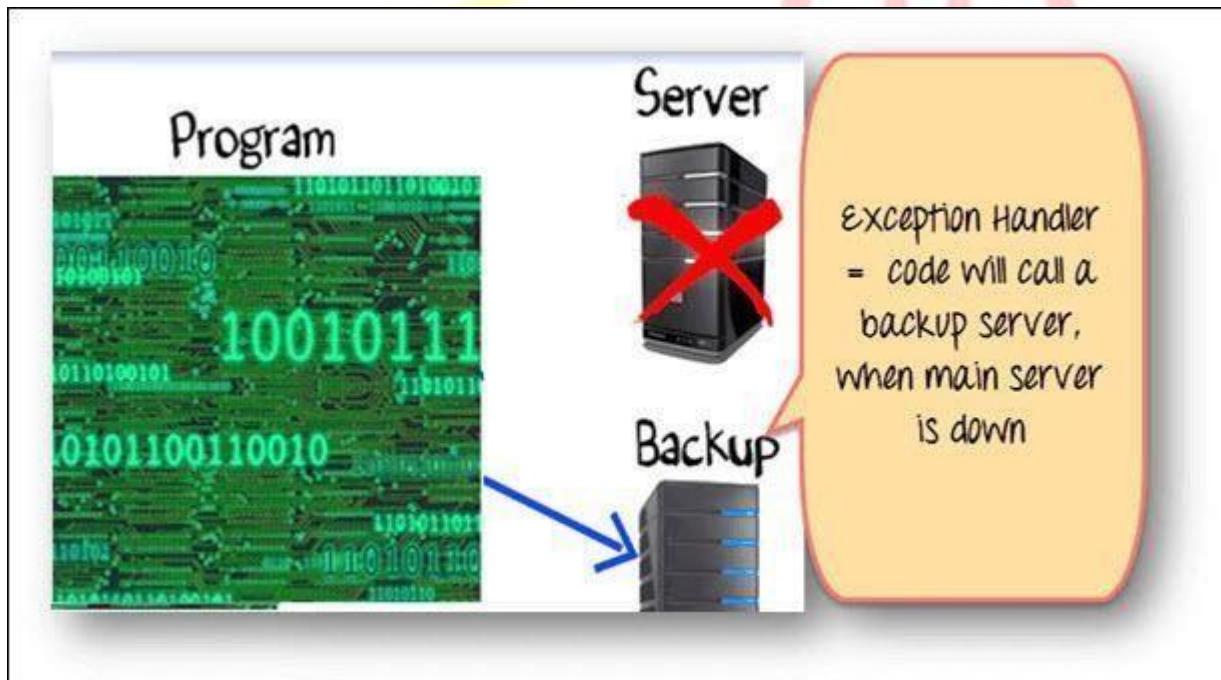
How to Handle Exception

So far we have seen, exception is beyond developer's control. But blaming your code failure on environmental issues is not a solution. You need a Robust Programming,

Transforming Career

which takes care of exceptional situations. Such code is known as **Exception Handler**.

In our example, good exception handling would be, when the server is down, connect to the backup server.



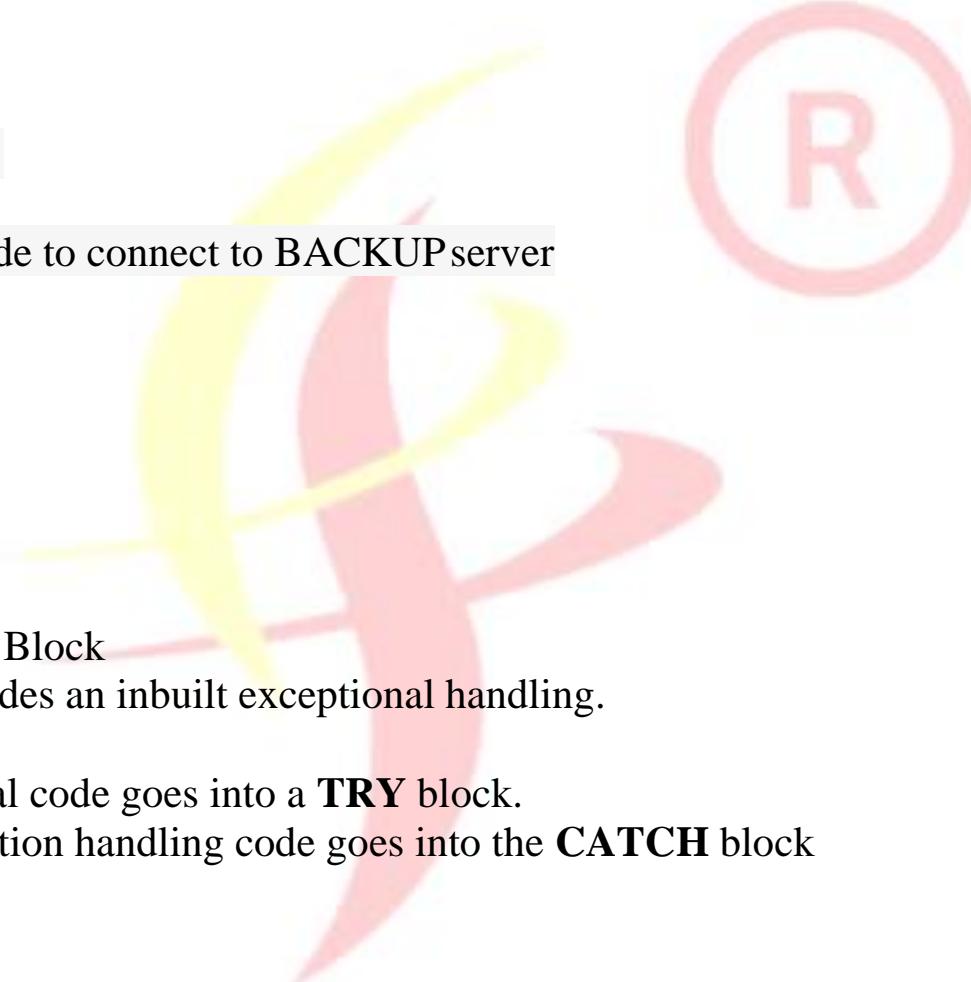
To implement this, enter your code to connect to the server (Using traditional if and else conditions).

You will check if the server is down. If yes, write the code to connect to the backup server.

Such organization of code, using "if" and "else" loop is not effective when your code has multiple java exceptions to handle.

```
class connect{
```

```
if(Server Up){  
    // code to connect to server  
}  
  
else{  
    // code to connect to BACKUP server  
}  
}
```



Try Catch Block
Java provides an inbuilt exceptional handling.

The normal code goes into a **TRY** block.
The exception handling code goes into the **CATCH** block

Yess InfoTech Pvt. Ltd.
Transforming Career

```
class connect{  
    1 try{  
        //Code to connect to server  
    }  
    2 catch{  
        //Code to connect to Backup  
        server  
    }  
}
```

Try block - Normal code

catch block - exception handling code

In our example, TRY block will contain the code to connect to the server. CATCH block will contain the code to connect to the backup server.

In case the server is up, the code in the CATCH block will be ignored. In case the server is down, an exception is raised, and the code in catch block will be executed.

```
class connect{  
    try{  
        //Code to connect to server  
    }
```

Exception
catch{

if server is down, an exception is raised.
The code in the catch block will be executed

So, this is how the exception is handled in Java.

Syntax for using try & catch

```
try{
```

```
    statement(s)
```

```
}
```

```
catch (exceptiontype name){
```

statement(s)

}

Example

Step 1) Copy the following code into an editor

```
class JavaException {  
    public static void main(String args[]){  
        int d = 0;  
        int n = 20;  
        int fraction = n/d;  
        System.out.println("End Of Main");  
    }  
}
```

Yess InfoTech Pvt. Ltd.

Transforming Career

Step 2) Save the file & compile the code. Run the program using command, java JavaException

Step 3) An Arithmetic Exception - divide by zero is shown as below for line # 5 and line # 6 is never executed

Step 4) Now let's see examine how try and catch will help us to handle this exception.

We will put the exception causing the line of code into a **try** block, followed by a **catch** block. Copy the following code into the editor.

```
class JavaException {  
  
    public static void main(String args[]) {  
  
        int d = 0;  
  
        int n = 20;  
  
        try {  
  
            int fraction = n / d;  
  
            System.out.println("This line will not be Executed");  
  
        } catch (ArithmaticException e) {  
    }  
}
```

Transforming Career

```
    System.out.println("In the catch Block due to Exception = "
+ e);

}

System.out.println("End Of Main");

}
```



Step 5) Save, Compile & Run the code. You will get the following output

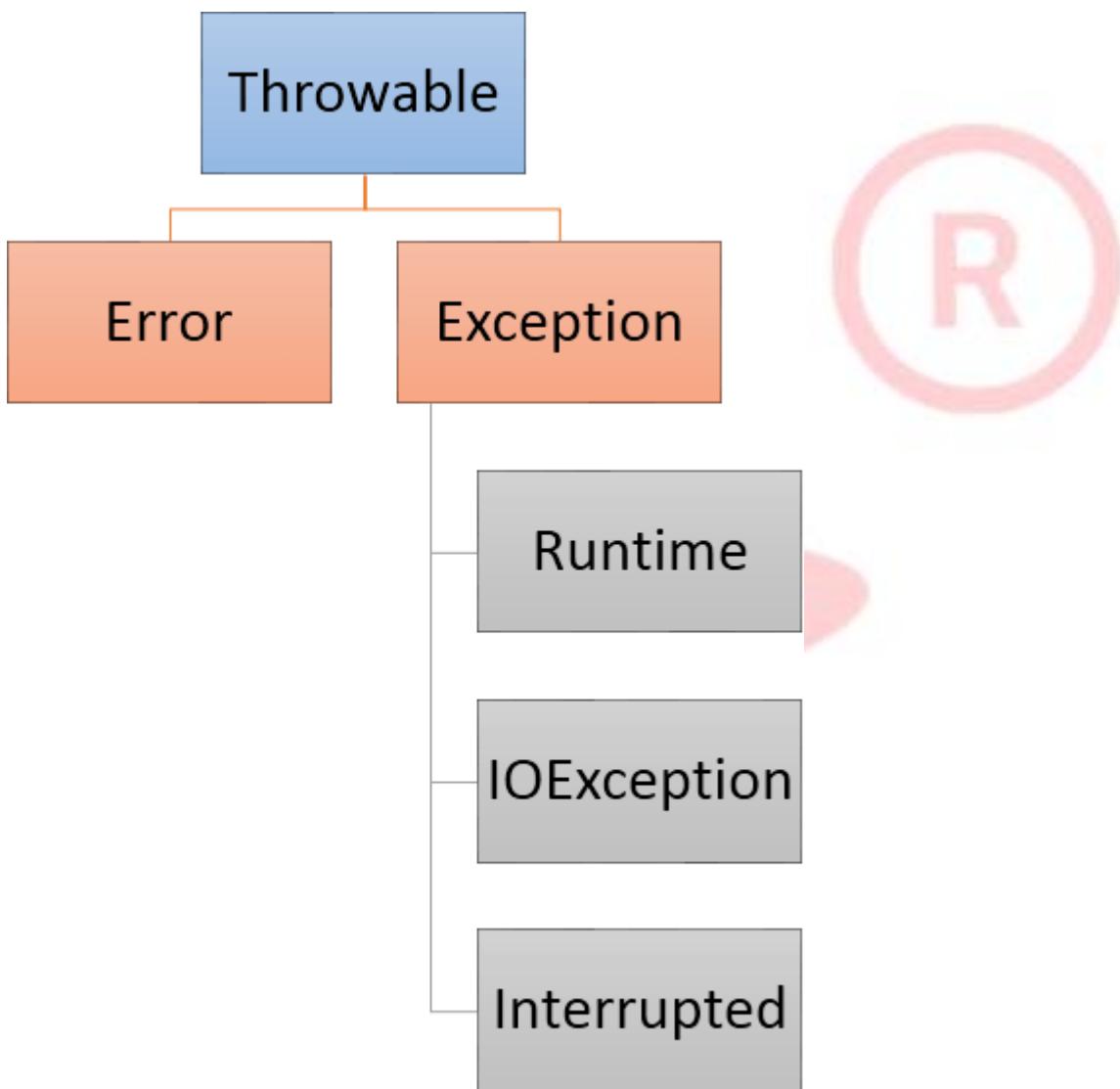
```
C:\workspace>java JavaException
In the catch clock due to Exception = java.lang.ArithmetricException: / by zero
End Of Main
```

As you observe, the exception is handled, and the last line of code is also executed. Also, note that Line #7 will not be executed because **as soon as an exception is raised the flow of control jumps to the catch block.**

Note: The ArithmetricException Object "e" carries information about the exception that has occurred which can be useful in taking recovery actions.

Java Exception class Hierarchy

After one catch statement executes, the others are bypassed, and execution continues after the try/catch block. The nested catch blocks follow Exception hierarchy.



Exception Hierarchy

All exception classes in Java extend the class ‘**Throwable**’. **Throwable** has two subclasses, **Error** and **Exception**

The **Error** class defines the exception or the problems that are not expected to occur under normal circumstances by our program, example Memory error, Hardware error, JVM error, etc

The Exception class represents the exceptions that can be handled by our program, and our program can be recovered from this exception using try and catch block. A Runtime exception is a sub-class of the exception class. The Exception of these type represents exception that occur at the run time and which cannot be tracked at the compile time. An excellent example of same is divide by zero exception, or null pointer exception, etc.

IO exception is generated during input and output operations. Interrupted exceptions in Java, is generated during multiple threading.

Example: To understand nesting of try and catch blocks

Step 1) Copy the following code into an editor.

```
class JavaException {  
    public static void main(String args[]) {  
        try {  
            int d = 1;  
            int n = 20;  
            int fraction = n / d;
```

```
int g[] = {  
    1  
};  
g[20] = 100;  
}  
  
/*catch(Exception e){  
    System.out.println("In the catch block due to Exception  
= "+e);  
}  
*/  
catch (ArithmaticException e) {  
    System.out.println("In the catch block due to Exception = "  
+ e);  
}  
} catch (ArrayIndexOutOfBoundsException e) {  
    System.out.println("In the catch block due to Exception = "  
+ e);
```

Transforming Career

```
    }  
  
    System.out.println("End Of Main");  
  
}  
  
}
```



Step 2) Save the file & compile the code. Run the program using command, **java JavaException**.

Step 3) An `ArrayIndexOutOfBoundsException` is generated. Change the value of `int dto 0`. Save, Compile & Run the code.

Step 4) An `ArithmaticException` must be generated.

Step 5) Uncomment line #10 to line #12. Save, Compile & Run the code.

Step 6) Compilation Error? This is because `Exception` is the base class of `ArithmaticException`. Any `Exception` that is raised by `ArithmaticException` can be handled by `Exception` class as well. So the catch block of `ArithmaticException` will never get a chance to be executed which makes it redundant. Hence the compilation error.

Java Finally Block

Transforming Career

The finally block is **executed irrespective of an exception being raised** in the try block. It is **optional** to use with a try block.

```
try {  
    statement(s)  
}  
  
catch (ExceptionType name)  
{  
    statement(s)  
}  
  
finally {  
    statement(s)  
}
```

In case, an exception is raised in the try block, finally block is executed after the catchblock is executed.

Transforming Career

Example

Step 1) Copy the following code into an editor.

```
class JavaException {  
    public static void main(String args[]){  
        try{  
            int d = 0;  
            int n = 20;  
            int fraction = n/d;  
        }  
        catch(ArithmaticException e){  
            System.out.println("In the catch block due to Exception="+e);  
        }  
        finally{  
    }
```

```
        System.out.println("Inside the finally block");

    }

}

}
```

Step 2) Save, Compile & Run the Code.

Step 3) Expected output. Finally block is executed even though an exception is raised.

Step 4) Change the value of variable d = 1. Save, Compile and Run the code and observe the output. Bottom of Form

Summary:

An **Exception is a run-time error** which interrupts the normal flow of program execution. Disruption during the execution of the program is referred as error or exception.

Errors are classified into two categories

Compile time errors – Syntax errors, Semantic errors

Runtime errors- Exception

A **robust program should handle all exceptions** and continue with its normal flow of program execution. Java provides an inbuilt exceptional handling method. Exception Handler is a set of code that **handles an exception**. Exceptions can be handled in Java using try & catch.

Try block: Normal code goes on this block.

The Java Collection Framework

The **collections framework** consists of several interfaces and classes (in the `java.util` package) that define data structures to store data, search for and retrieve data, and perform other operations on a collection efficiently. An example is the `ArrayList` class.

This class implements the `List` interface in the *collections framework*. There are two primary interfaces in the collections framework from which all other interfaces are derived: `Collection` and `Map`. The `Collection` interface is extended by the `Set` and `List` interfaces.

The List, Set, and Map interfaces described as follows:

- **The List interface:** A list is a specific arrangement of elements in a collection. The elements in a list can access via their indices. This interface defines methods to read, add, and remove elements from the specified indices. The `ArrayList` and `LinkedList` classes implement this interface.
- **The Set interface:** The main feature of a set is that it cannot duplicate elements. It distinguishes it from a list in which duplicates are allowed. For example, consider a set of fingerprints in which each fingerprint is unique. The `Set` interface declares methods to check whether an element is present in a set and add and remove a specified element from a set. The `HashSet`, `TreeSet`, and `LinkedHashSet` classes implement this interface.
- **The Map interface:** A map associates a key with a value. It makes it possible to find and retrieve objects efficiently from a collection. For example, consider a world map. The key is the place's coordinates on a map, and the value is the place's name. Once you know the coordinates, you can locate the place quickly on the map. The classes that implement the `Map` interface are `TreeMap`, `HashMap`, and `LinkedHashMap`.

Need for Collection Framework:

JDK earlier than 1.2 or (Before Collection Framework), the standard library methods are built-in `Java` methods that are readily available for grouping Java objects (or collections) were `Arrays` or `Vectors` or `Hashtables`. All of these collections objects had no standard interface. Accessing elements of these Java objects was a hassle as each had a different method (and syntax) for accessing its members:

advantages of the Collections Framework in Java are as follows:

- **Consistent API:** The first benefit of the Collections Framework in Java is a



Yess InfoTech Pvt. Ltd.

Transforming Career

consistent and regular API. The API provides a standard interface, Collection, Set, List, or Map. Java Collections Framework provides basic classes (ArrayList, LinkedList, Vector, etc.) and utility methods that implement these interfaces.

- **Reduces programming effort:**

It gives core collection classes that reduce development effort, due to which we need not write our collection classes.

- **Increases program speed and quality:**

Using the Java Collection Framework provides high-performance implementations of the interfaces that increase the performance.

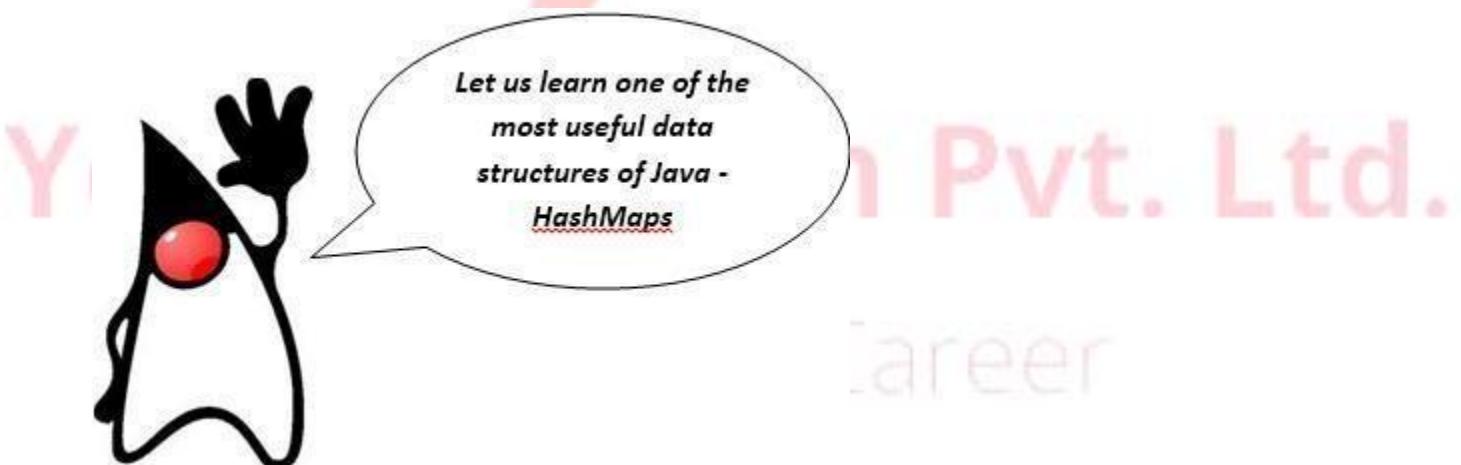
Disadvantages of collections framework:

- Collections Framework must typecast objects or references to correct data types.
- Collections Framework can't do compile-time type checking.

HashMap in Java Learn with Example

What is Hashmap in Java?

A HashMap basically designates **unique keys** to corresponding **values** that can be retrieved at any given point.



Features of Java Hashmap

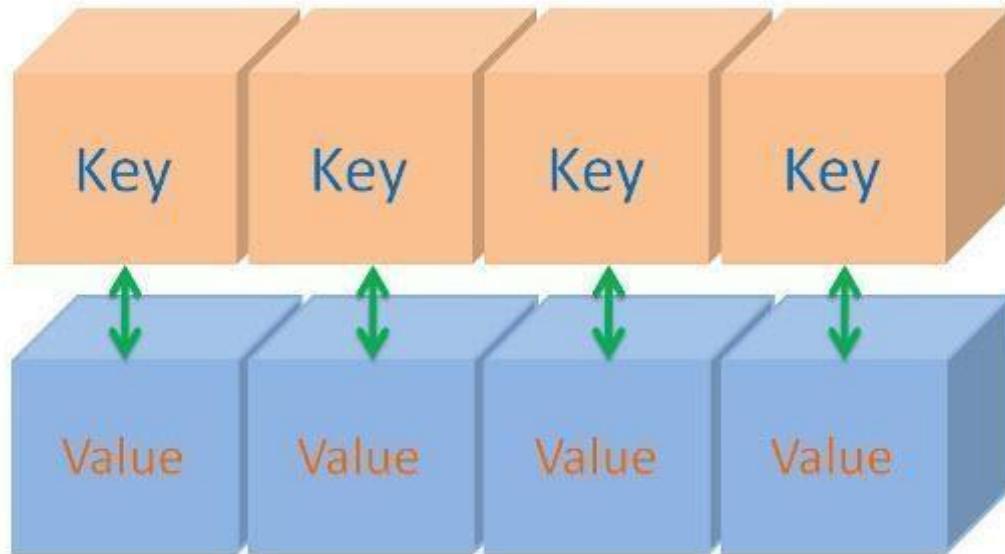
The **values** can be stored in a map by forming a **key-value** pair. The value can be retrieved using the key by passing it to the correct method.

If **no element** exists in the Map, it will throw a '**NoSuchElementException**'.

HashMap stores only **object references**. That is why, it is impossible to use **primitive data types** like double or int. Use wrapper class (like Integer or Double) instead.

Yess InfoTech Pvt. Ltd.
Transforming Career

Key-Value Pairs



Using HashMaps in Java Programs:

Following are the two ways to declare a Hash Map:
HashMap<String, Object> map = new HashMap<String, Object>();
HashMap x = new HashMap();
Important Hashmap Methods

get(Object KEY) – This will return the value associated with a specified key in this Java hashmap.

Transforming Career

put(Object KEY, String VALUE) – This method stores the specified value and associates it with the specified key in this map.

Java Hashmap Example

Following is a sample implementation of java Hash Map:

```
import java.util.HashMap;import java.util.Map;
public class Sample_TestMaps{

    public static void main(String[] args){

        Map<String, String> objMap = new HashMap<String, String>();

        objMap.put("Name", "Suzuki");

        objMap.put("Power", "220");

        objMap.put("Type", "2-wheeler");

        objMap.put("Price", "85000");

        System.out.println("Elements of the Map:");

        System.out.println(objMap);
```

```
}
```

```
}
```

Output:

Example 2: Remove a value from HashMap based on key
import java.util.*;

```
public class HashMapExample {
```

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

```
        // create and populate hash map
```

```
        HashMap<Integer, String> map = new HashMap<Integer, String>();
```

```
        map.put(1, "Java");
```

```
        map.put(2, "Python");
```

```
        map.put(3, "PHP");
```

```
        map.put(4, "SQL");
```

Transforming Career

```
map.put(5, "C++");

System.out.println("Tutorial in Guru99: "+ map);

// Remove value of key 5

map.remove(5);

System.out.println("Tutorial in Guru99 After Remove: "+ map);

}

}
```

this Keyword in Java

Garbage Collection in Java

What is Garbage Collection in Java?

Garbage Collection in Java is a process by which the programs perform memory management automatically. The Garbage Collector(GC) finds the unused objects and deletes them to reclaim the memory. In Java, dynamic memory allocation of objects is achieved using the new operator that uses some memory and the memory remains allocated until there are references for the use of the object.

When there are no references to an object, it is assumed to be no longer needed, and the memory, occupied by the object can be reclaimed. There is no explicit need to destroy an object as Java handles the de-allocation automatically. The technique that accomplishes this is known as **Garbage Collection**. Programs that do not de-allocate memory can eventually crash when there is no memory left in the system to allocate. These programs are said to have *memory leaks*.

Garbage collection in Java happens automatically during the lifetime of the program, eliminating the need to de-allocate memory and thereby avoiding memory leaks.

In C language, it is the programmer's responsibility to de-allocate memory allocated dynamically using `free()` function. This is where Java memory management leads.

Note: All objects are created in **Heap** Section of memory.

Yess InfoTech Pvt. Ltd.
Transforming Career

Example: To Learn Garbage Collector Mechanism in Java

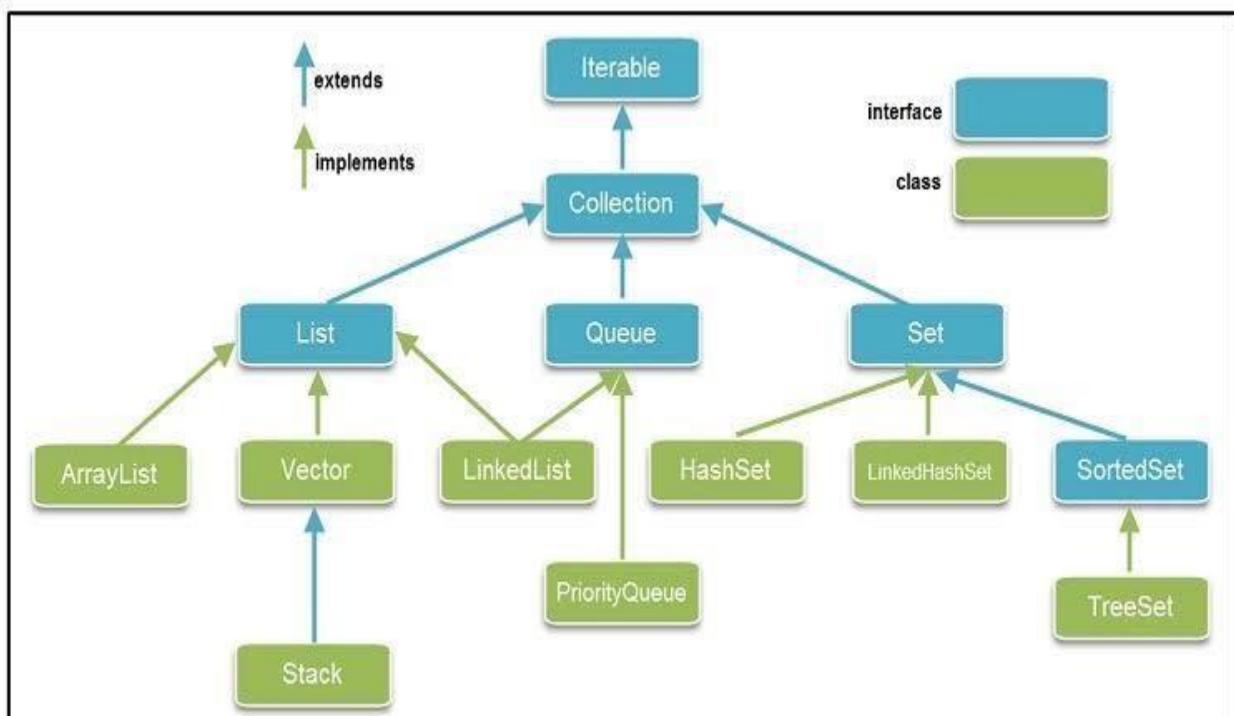
1. class Student{

```
2. int a;  
3. int b;  
4.  
5. public void setData(int c,int d){  
6.     a=c;  
7.     b=d;  
8. }  
9. public void showData(){  
10.    System.out.println("Value of a = "+a);  
11.    System.out.println("Value of b = "+b);  
12. }  
13. public static void main(String args[]){  
14.    Student s1 = new Student();  
15.    Student s2 = new Student();  
16.    s1.setData(1,2);  
17.    s2.setData(3,4);  
18.    s1.showData();  
19.    s2.showData();  
20.    //Student s3;  
21.    //s3=s2;  
22.    //s3.showData();  
23.    //s2=null;
```

```
24. //s3.showData();  
25. //s3=null;  
26. //s3.showData();  
27. }  
28.}
```

Hierarchy of Collection Framework

The Collection framework's hierarchy consists of four core interfaces such as Collection, List, Set, Map, and two specialized interfaces named SortedSet and SortedMap for sorting. The `java.util` package contains all the interfaces and classes for the collection framework.



Methods of Collection interface

There are many methods declared in the Collection interface. They are as follows:

No.	Method	Description
1	public boolean add(Object obj)	A method that adds the object obj to the end of the list.
2	public boolean addAll(Collection<? extends E> c)	This method is used for adding all the elements of a list to the another list.
3	public boolean remove(Object element)	Removes the first occurrence of the specified element from Vector If the Vector does not contain the element, it is unchanged.
4	public boolean removeAll(Collection<?> c)	The removeAll() method is used to remove all the elements from a list that are contained in the specified collection.
5	default boolean removeIf(Predicate<? super E> filter)	Removes all of the elements of this collection that satisfy the given predicate.
6	public boolean retainAll(Collection<?> c)	The retainAll() method is used to remove it's elements from a list that are not contained in the specified collection.
7	public int size()	The size() method is used to get the number of elements in an ArrayList object.
8	public void clear()	The clear() method is used to remove all of the elements from a list.
9	public boolean contains(Object element)	The return type is boolean which means this method returns true or false.
10	public boolean containsAll(Collection<?> c)	Returns true if this collection contains all of the elements in the specified collection.
11	public Iterator iterator()	Returns an iterator over the elements in this list.
12	public Object[] toArray()	This method returns an array containing all the elements of a list in proper sequence.
13	public <T> T[] toArray(T[] a)	Returns an array containing all of the elements in this list in proper sequence (from first to last element);
14	public boolean isEmpty()	This method returns true if the length of the string is 0.
15	default Stream<E> parallelStream()	Returns a possibly parallel Stream with this collection as its source.
16	default Stream<E> stream()	The method stream() of the Collection interface can be called on an existing collection.
17	default Spliterator<E> spliterator()	Creates a Spliterator over the elements in this collection.
18	public boolean equals(Object obj)	The equals() method returns a Boolean value.
19	public int hashCode()	

Iterator interface

An iterator is an object that can be used to move sequentially through the data stored in a collection. You have already seen how the enhanced for loop can be used for the same purpose. An iterator is used when the elements in a collection have to be removed during traversal. Sets and lists provide an iterator, whereas maps do not. The iterator method returns an iterator to access the elements of a set.

Methods of Iterator interface

There are only three methods in the Iterator interface. They are:

No.	Method	Description
1	public boolean hasNext()	Returns true if there is another element in the collection after the current element.
2	public Object next()	Returns the element that appears next in the iteration.
3	public void remove()	Remove the element that was returned by the last call to method next.

The

Collection Interface



Yess InfoTech Pvt. Ltd.

Transforming Career

Java specifies an interface called Collection that declares methods to manipulate a group of objects. Some methods in this interface are shown in the Figure below.

List Interface

The List interface specifies methods to find and modify elements based on their position. Objects also have a given order in a Queue, but the ordering scheme is different from that in List.

The ArrayList and LinkedList classes implement the List interface. Like ArrayList, a LinkedList maintains the order in which elements are added to it. Another similarity is that in both types of lists, elements can access via their indices. The difference between the two is that an array list stores its elements in an array, whereas a linked list uses a group of connected nodes. Each node stores object data and linked to the previous and next node. (This configuration is analogous to how the cars in a train are connected.) A linked list is traversed by starting from either end and following the links from one node to the next. Array lists generally offer better performance than linked lists and tend to be used more often. Array lists provide faster random access of elements because, in them, an element can be accessed directly using its index, whereas, in a linked list, the intermediate nodes must be traversed before the required elements can be accessed. However, when elements need to be inserted or removed frequently

from the middle of the List, a linked list can be faster than an array list.

No.	Method	Description
1	public boolean add(Object obj)	A method that adds the object obj to the end of the list.
2	public boolean addAll(Collection<? extends E> c)	This method is used for adding all the elements of a list to the another list.
3	public boolean remove(Object element)	Removes the first occurrence of the specified element from Vector If the Vector does not contain the element, it is unchanged.
4	public boolean removeAll(Collection<?> c)	The removeAll() method is used to remove all the elements from a list that are contained in the specified collection.
5	default boolean removeIf(Predicate<? super E> filter)	Removes all of the elements of this collection that satisfy the given predicate.
6	public boolean retainAll(Collection<?> c)	The retainAll() method is used to remove it's elements from a list that are not contained in the specified collection.
7	public int size()	The size() method is used to get the number of elements in an ArrayList object.
8	public void clear()	The clear() method is used to remove all of the elements from a list.
9	public boolean contains(Object element)	The return type is boolean which means this method returns true or false.
10	public boolean containsAll(Collection<?> c)	Returns true if this collection contains all of the elements in the specified collection.
11	public Iterator iterator()	Returns an iterator over the elements in this list.
12	public Object[] toArray()	This method returns an array containing all the elements of a list in proper sequence.
13	public <T> T[] toArray(T[] a)	Returns an array containing all of the elements in this list in proper sequence (from first to last element);
14	public boolean isEmpty()	This method returns true if the length of the string is 0.
15	default Stream<E> parallelStream()	Returns a possibly parallel Stream with this collection as its source.
16	default Stream<E> stream()	The method stream() of the Collection interface can be called on an existing collection.
17	default Spliterator<E> spliterator()	Creates a Spliterator over the elements in this collection.
18	public boolean equals(Object obj)	The equals() method returns a Boolean value.
19	public int hashCode()	

ArrayList and LinkedList classes implement this interface. Examples of using an ArrayList are shown here:

```

ArrayList<Double>      list1      = new      ArrayList<Double>();
list1.add(10.5);          //           list1      = [10.5]
list1.add(30.5);          //           list1      = [10.5,      30.5]
list1.add(1,    20.5);    //           list1      = [10.5,    20.5,      30.5]
list1.set(2,    50.5);    //           list1      = [10.5,    20.5,    50.5]
list1.remove(0);
System.out.println(list1); // list1 = [20.5, 50.5]

```

A LinkedList can be used similarly:

```

LinkedList<Double>    list2      = new      LinkedList<Double>();
list2.add(10.5);          //           list2      = [10.5]

```

```
list2.add(1, 20.5); // list2 = [10.5, 20.5]
```



Yess InfoTech Pvt. Ltd.

Transforming Career

```
list2.remove(1);  
System.out.println(list2); // list2 = [10,5]
```

The ArrayList Class

The `ArrayList` class in `java.util` package used when we want an array whose size changes while the program executes. You know that arrays have a fixed size that must be specified when the array is created. How can an element be added to an array that filled? One way to do this is for the programmer to create a more extensive array and copy all of the smaller array elements into the larger one, with the remaining space for new elements. An alternative is to use the `ArrayList` class for arrays whose sizes change frequently. An instance of this class contains an array that grows and shrinks automatically as elements are added or removed to relieve the programmer from the burden of doing this. The figure shows some of the constructors and methods in this class.

The following statements show how to use the add method:

```
ArrayList list = new ArrayList();  
list.add(100);  
list.add(200);  
list.add(300);
```

This prints out the size of list:

```
System.out.println(list.size());
```

The size is 3 because list has three elements: 100, 200, and 300. Note that the length field is used to obtain the array size for a regular array, whereas the size method is used for an instance of the `ArrayList` class.

This removes the element at index 1:

```
list.remove(1);
```

The list now contains the elements 100 and 300, and has size 2.

LinkedList class

The `LinkedList` class is built in class where data is stored as a separate objects i.e. each list element is stored along with a pointer to the object that precedes it and the object that follows it. With the help of these pointers, we can navigate through the entire list. As said earlier, it is because of these pointers that we can easily insert or delete an element from a linked list (just by adjusting the previous and next pointers of the element).

Constructor:

LinkedList <E> () Creates an empty linked list using the specified type.



Yess InfoTech Pvt. Ltd.

Transforming Career

Common methods of the LinkedList class

No.	Method	Description
1	ArrayList()	Constructor creates an empty array of size ten.
2	ArrayList(int n)	Constructor creates an empty array of size n.
3	boolean add(object obj)	Adds the object obj to the end of the array and returns true.
4	Object remove(int index)	Removes and returns the object at the given index.
5	int size()	Returns the number of elements in this array.
6	boolean isEmpty()	Returns true if this array does not have any elements; otherwise, returns false.
7	Object get(int i)	Returns (without removing) the element at index i from the array .
8	Object set(int i, Object obj)	Changes the element at index i to obj and returns the previous element at this index.

Vector Class

The Vector class provides the capability to implement a growable array. The array grows larger as more elements are added to it. The array may also be reduced in size, after some of its elements have been deleted. This is accomplished using the trimToSize() method.

Vector operates by creating an initial storage capacity and then adding to this capacity as needed. It grows by an increment defined by the increment variable. The initial storage capacity and increment can be specified in Vector's constructor. There are three types of constructors:

Vector()

Vector(intsize)

Vector(int size, int increment)

Stack Class

The Stack class provides the capability to create and use stacks within the Java programs. Stacks are storage objects that store information by pushing it onto a stack and remove and retrieve information by popping it off the stack. Stacks implement a last-in-first-out storage capability: The last object pushed on a stack is the first object that can be retrieved from the stack. The Stack class extends the Vector class.

The Stack class provides a single default constructor, Stack(), that is used to create

an empty stack.
Objects are placed on the stack using the pushO method and retrieved from the



Yess InfoTech Pvt. Ltd.

Transforming Career

stack using the `pop()` method.

search() – It allows us to search through a stack to see if a particular object is contained on the stack.

peek() – It returns the top element of the stack without popping it off.

empty() – It is used to determine whether a stack is empty.

The `pop()` and `peek()` methods both throw the `EmptyStackException` if the stack is empty. Use of the `empty()` method can help to avoid the generation of this exception.

The Set Interface

The Set interface also has special methods (inherited from Collection) to perform operations such as finding the intersection and union of the elements of two sets. All methods that modify the contents of a collection (such as `add` and `remove` in the Set interface) are optional methods. A class that extends an interface must implement all the methods in that interface; however, it is not necessary for a class to implement optional methods. If an instance of a class invokes a method that is not implemented in that class, an `UnsupportedOperationException` is thrown.

What is Thread?

A thread is a single sequential flow of control within a program. It differs from a process in that a process is a program executing in its own address space whereas a thread is a single stream of execution within a process.

A thread can be defined as a process in execution within a program. A thread by itself is not a complete program and cannot run on its own. It runs within a program; however each thread has a beginning, and an end, and a sequential flow of execution at a single point of time. At any given instant within the run-time of the program, only one thread will be executed by the processor.

Multithreading allows a program to be structured as a set of individual units of control that run in parallel; however, the Central Processing Unit (CPU) can only run one process at a time. The CPU time is divided into slices and a single thread will run in a given time slice. Typically, the CPU switches between multiple

threads and runs so fast that it appears as if all threads are running at the same



Yess InfoTech Pvt. Ltd.

Transforming Career

time. It is better, as a programming practice, to identify different parts of the program that can perform in parallel and implement them into independent threads.

Multithreading differs from multitasking: multitasking allows multiple tasks (which can be processes or programs) to run concurrently whereas multithreading allows multiple units within a program (threads) to run concurrently.

Unlike the processes in some operating systems (for example, Unix), the threads in Java are ‘light-weight processes’ as they have relatively low overheads and share common memory space. This facilitates an effective and inexpensive communication between threads.

Thread Creation

When a Java program begins execution, it always has atleast one thread, i.e., the main thread. In regular Java application program, this thread starts at the beginning of main () method that means that when your program creates a thread, it is in addition to the main thread of execution that created it. Java provides two ways to make your class behave like a thread.

- By extending a java .lang. Thread class and override its run () method
- By implementing the Runnable interface and implementing its run() method

Now the question arises which of the two thread creating techniques one should use. The first technique can only be used for classes that do not already extend any other class. Therefore, we stick with the second approach which is always applicable.

Before creating a class that extends the Thread class, you need to understand the java.lang.Thread class. The Thread class lets you create an object that can be run as a thread in a multithreaded Java application. It is a direct subclass of the object class, and it implements the Runnable interface. A Thread object controls a thread running under the JVM. The Thread class contains the following constructors for creating threads and methods for controlling threads.

Constructors	Purpose
Thread ()	Creates an instance of the Thread class.
Thread(String name)	Creates a Thread object and assigns a specified name to the thread.

Thread(Runnable target)	Creates a new Thread object that uses the run () method of the
-------------------------	--



Yess InfoTech Pvt. Ltd.

Transforming Career

specified target.

Method	Purpose
static Thread currentThread ()	Returns the reference of the currently executing Thread object.
void final start()	Causes the associated thread to start execution. The JVM calls the run () method of this thread.
void run ()	If the thread was created using separate Runnable object then that Runnable object's run ()method is called, otherwise this method does nothing and returns.
String getName() void setName(String name)	Returns the name of the thread. Changes the name of the thread to the one specified in the argument.
static void sleep (long milliseconds)	Cause the currently executing thread to sleep for the specified number of milliseconds. This method throws InterruptedException exception and must be used within the try catch block.
static void yield()	Causes the currently running thread to temporarily pause and allow other threads to execute.
final void setPriority (int newPriority)	Changes the priority of the thread with the value specified in the argument.
int getpriority()	Returns threads priority.
final boolean isAlive()	Tests if the thread is alive.
Method	Purpose

void setDaemon Marks the thread as either a user thread or a daemon



Yess InfoTech Pvt. Ltd.

Transforming Career

(boolean on)	thread.
boolean isDaemon()	Tests whether a thread is Daemon thread or not.
final void join()	Waits for the thread to die.
void interrupt()	Interrupts a thread.
void stops ()	Stops the thread.

Example of Thread

```

class NewThreadDemo implements Runnable {
    Thread t;
    NewThreadDemo(String threadName){
        t = new Thread(this, threadName);
        System.out.println("New userdefined child thread created"+threadName);
        t.start();
    }
    public void run(){
        try{
            for(int i =1; i <=10; i++){
                System.out.println("Child Thread: [ "+Thread.currentThread().getName()+" ] "+ i);
                Thread.sleep(500);
            }
        } Transforming Career
        catch(InterruptedException e){
            System.out.println("UseDefine Child interrupted.");
        }
        System.out.println("Exiting UserDefined child thread.");
    }
}

```

}



Yess InfoTech Pvt. Ltd.

Transforming Career

```

}

class MultiThreadDemo {
    public static void main(String args[]){
        new NewThreadDemo("Child_One");
        new NewThreadDemo("Child_Two");
        new NewThreadDemo("Child_Three");
        new NewThreadDemo("Child_Four");
        try{
            for(int i =1; i <10; i++){
                System.out.println("Main Thread: " + i);
                Thread.sleep(500);
            }
        }
        catch(InterruptedException e){
            System.out.println("Main thread interrupted.");
        }
        System.out.println("Main thread exiting.");
    }
}

```

Yess InfoTech Pvt. Ltd.

Let's see the difference between multitasking and multithreading:

S.NO	Multitasking	Multithreading
1.	In multitasking, users are allowed to perform many tasks by CPU.	While in multithreading, many threads are created from a process through which computer power is increased.

S.NO	Multitasking	Multithreading
2.	Multitasking involves often CPU switching between the tasks.	While in multithreading also, CPU switching is often involved between the threads.
3.	In multitasking, the processes share separate memory.	While in multithreading, processes are allocated the same memory.
4.	The multitasking component involves multiprocessing.	While the multithreading component does not involve multiprocessing.
5.	In multitasking, the CPU is provided in order to execute many tasks at a time.	While in multithreading also, a CPU is provided in order to execute many threads from a process at a time.
6.	In multitasking, processes don't share the same resources, each process is allocated separate resources.	While in multithreading, each process shares the same resources.
7.	Multitasking is slow compared to multithreading.	While multithreading is faster.
8.	In multitasking, termination of a process takes more time.	While in multithreading, termination of thread takes less time.
9.	Isolation and memory protection exist in multitasking.	Isolation and memory protection does not exist in multithreading.
10.	It helps in developing efficient	It helps in developing efficient

programs.

operating systems.



Yess InfoTech Pvt. Ltd.

Transforming Career

Multithreading Example with Synchronization

Here is the same example which prints counter value in sequence and every time we run it, it produces the same result.

Example

```
class PrintDemo {  
    public void printCount() {  
        try {  
            for(int i = 5; i > 0; i--) {  
                System.out.println("Counter --- " + i);  
            }  
        } catch (Exception e) {  
            System.out.println("Thread interrupted.");  
        }  
    }  
  
    class ThreadDemo extends Thread {  
        private Thread t;  
        private String threadName;  
        PrintDemo PD;  
  
        ThreadDemo( String name, PrintDemo pd) {  
            threadName = name;  
            PD = pd;  
        }  
  
        public void run() {  
            synchronized(PD) {  
                PD.printCount();  
            }  
            System.out.println("Thread " + threadName + " exiting.");  
        }  
  
        public void start () {  
    }
```

```
System.out.println("Starting " + threadName );  
if (t == null) {
```



Yess InfoTech Pvt. Ltd.

Transforming Career

```
t = new Thread (this, threadName);
t.start ();
}

}

public class TestThread {

public static void main(String args[]) {
PrintDemo PD = new PrintDemo();

ThreadDemo T1 = new ThreadDemo( "Thread - 1 ", PD );
ThreadDemo T2 = new ThreadDemo( "Thread - 2 ", PD );

T1.start();
T2.start();

// wait for threads to end
try {
    T1.join();
    T2.join();
} catch ( Exception e) {
    System.out.println("Interrupted");
}
}
}
```

Yess InfoTech Pvt. Ltd.

Transforming Career

Now, let's take a look at difference between these types of system:

No	Characteristic	Multiprogramming	Multiprocessing	Multithreading	Multitasking
1	What it is:	The concurrent residency of more than one program in the main memory is called as multiprogramming.	The availability of more than one processor per system, which can execute several set of instructions in parallel is called as multiprocessing.	A process is divided into several different subprocesses called as threads, which has its own path of execution. This concept is called as multithreading.	The execution of more than one task simultaneously is called as multitasking.
2	Number of CPU:	One	More than one	Can be one or more than one	One
3	Job processing time:	More time is taken to process the jobs.	Less time is taken for job processing.	Moderate amount of time is taken for job processing.	Moderate amount of time.
4	Number of process being executed:	One process is executed at a time.	More than one process can be executed at a time	Various components of the same process are being executed at a time.	One by one job is being executed at a time.
5	Economical:	It is economical.	Is less economical.	Is economical.	It is economical.
6	Number of users:	One at a time.	Can be one or more than one.	Usually one.	More than one.
7	Throughput:	Throughput is less.	Throughput is maximum.	Moderate.	Throughput is moderate.
8	Efficiency:	Less	Maximum	Moderate	Moderate
9	Categories:	No further divisions	Symmetric & Asymmetric.	No further divisions.	Single User & Multiuser.

What is Serialization in Java?

Serialization in Java is the concept of representing an object's state as a byte stream. The byte stream has all the information about the object. Usually used in Hibernate, JMS, JPA, and EJB, serialization in Java helps transport the code from one [JVM](#) to another and then de-serialize it there.

Deserialization is the exact opposite process of serialization where the byte [data type](#) stream is converted back to an object in the memory. The best part about these mechanisms is that both are JVM-independent, meaning you serialize on one JVM and de-serialize on another.

What are the Advantages of Serialization?

Serialization offers a plethora of benefits. Some of its primary advantages are:

- Used for marshaling (traveling the state of an object on the network)
- To persist or save an object's state
- JVM independent
- Easy to understand and customize

Points to Note About Serialization in Java?

To serialize an object, there are a few conditions to be met. Some other key points

need to be highlighted before you proceed further in the article. These are the conditions and points to remember while using serialization in Java.



Yess InfoTech Pvt. Ltd.

Transforming Career

- Serialization is a marker interface with no method or data member
- You can serialize an object only by implementing the Serializable interface
- All the fields of a class must be serializable; otherwise, use the transient keyword (more about it later)
- The child class doesn't have to implement the Serializable interface, if the parent class does
- The serialization process only saves non-static data members, but not static or transient data members

By default, the String and all wrapper classes implement the Serializable interface

FILE INPUT OUTPUT STREAM

The java.io package contains nearly every class you might ever need to perform input and output (I/O) in Java. All these streams represent an input source and an output destination. The stream in the java.io package supports many data such as primitives, object, localized characters, etc.

Stream

A stream can be defined as a sequence of data. There are two kinds of Streams –

- **InPutStream** – The InputStream is used to read data from a source.
- **OutPutStream** – The OutputStream is used for writing data to a destination.



Java provides strong but flexible support for I/O related to files and networks but this tutorial covers very basic functionality related to streams and I/O. We will see the most commonly used examples one by one –

Byte Streams

Java byte streams are used to perform input and output of 8-bit bytes. Though there are many classes related to byte streams but the most frequently used classes



Yess InfoTech Pvt. Ltd.
Transforming Career

are, **FileInputStream** and **FileOutputStream**. Following is an example which makes use of these two classes to copy an input file into an output file –

Example

```
import java.io.*;
public class CopyFile {

    public static void main(String args[]) throws IOException {
        FileInputStream in = null;
        FileOutputStream out = null;

        try {
            in = new FileInputStream("input.txt");
            out = new FileOutputStream("output.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        } finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Character Streams

Java **Byte** streams are used to perform input and output of 8-bit bytes, whereas Java **Character** streams are used to perform input and output for 16-bit unicode. Though there are many classes related to character streams but the most frequently

used classes are, **FileReader** and **FileWriter**. Though internally FileReader uses



Yess InfoTech Pvt. Ltd.

Transforming Career

FileInputStream and FileWriter uses FileOutputStream but here the major difference is that FileReader reads two bytes at a time and FileWriter writes two bytes at a time.

We can re-write the above example, which makes the use of these two classes to copy an input file (having unicode characters) into an output file –

Example

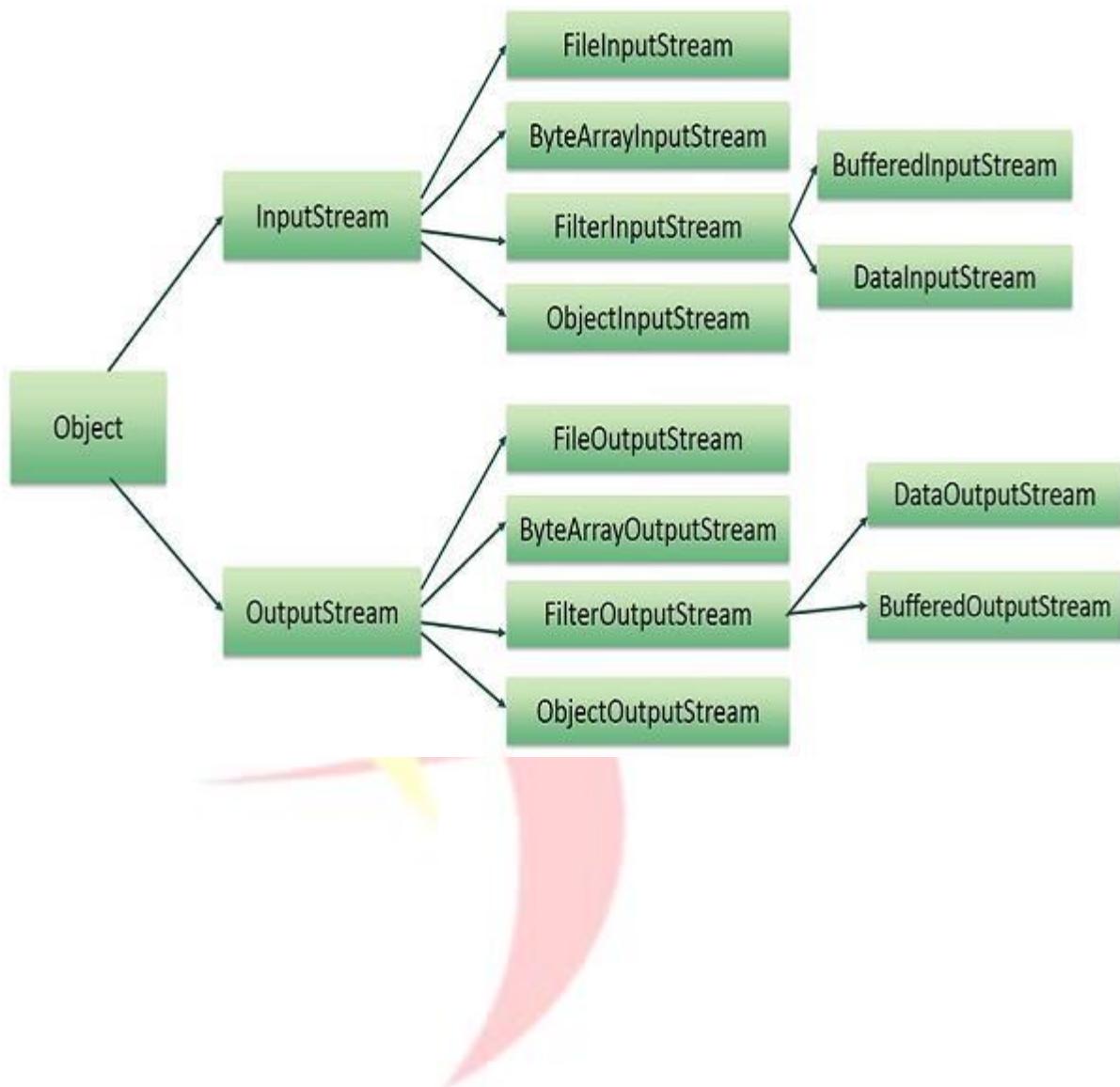
```
import java.io.*;
public class CopyFile {

    public static void main(String args[]) throws IOException {
        FileReader in = null;
        FileWriter out = null;

        try {
            in = new FileReader("input.txt");
            out = new FileWriter("output.txt");

            int c;
            while ((c = in.read()) != -1) {
                out.write(c);
            }
        }finally {
            if (in != null) {
                in.close();
            }
            if (out != null) {
                out.close();
            }
        }
    }
}
```

Here is a hierarchy of classes to deal with Input and Output streams.



Yess InfoTech Pvt. Ltd.

Transforming Career