FJava Concepts

1. **Steps to install java**

Type java in google and then go to the site -> After downloading java - > open exe file and run. We need to set environment variables for our system to understand where the java is present.

Setting Environment variables:

**Step:1**

Go to program files and open java -> copy the entire path -> right click on my computer and click on properties -> click on advance system settings and select environment variables -> select new in system variables, then enter variable name, paste the above copied path and click on ok

**Step:2**

Go to program files and open java -> open jdk and finally open bin folder and copy the entire path -> right click on my computer and click on properties -> click on advance system settings and select environment variables -> edit the path from system variables by adding “;” at the end and paste the above copied path -> click on ok

**--** Now, open cmd and type java -version -> then if you get the details of java then it got successfully installed.

1. **Steps to install eclipse**

Type eclipse in google and then go to the site -> choose Eclipse 64 bit if you download java 64 bit and choose Eclipse 32 bit if you download java 32 bit -> Now download and run the exe file, it will get installed -> It will show the default workspace (we can even change it to different)

1. **Steps to create workspace**

Open Eclipse and click on File -> go to Switch workspace and click on other -> you can choose different workspace and click on ok -> it will refresh and Eclipse will restart again with new workspace.

1. **Steps to create project**

Open Eclipse and click on File -> go to new and select project -> In that we can select our desired project and click on next -> give project name, choose saving location of the project if you don’t wish to save in the default one’s and click on Finish

**-->** You can see src folder and java jars which got imported. package resides in src folder and in that package classes will be there.

**-->** If you directly created a class, then it will automatically saves in default package or we can create our own package first and also desired classes in that package.

1. **create .java file/class**

After creating the project right click on it -> go to new and click on class ->select the desired package -> Enter name and check the public static void main(String[] args) if you want main method -> click on Finish

(or)

Directly right click on the desired package -> go to new and click on class -> Enter name and check the public static void main(String[] args) if you want main method -> click on Finish

1. **how to create packages and what is best way to give name**

After creating the project right click on it -> go to new and click on package -> Enter name -> click on Finish (or) If you directly created a class, then it will automatically saves in default package.

Best way to give name: Always use lowercase letters for package in order to avoid conflicts between classes and interfaces.

1. **what is main method will do?**

**Public static void main(String[] args)**

It is like an entry point for execution.

**public -** it is an access specifier where each and every class have an access.

**static -** it is given because we can invoke without the need of creating an object.

**void** - it doesn’t return any value to the caller(JVM).

**main()** - it is required for the interpreter(JVM), without this it cannot run the program but compiler can convert “.class” file without main.

**String[] args** - it can accept all types of values as arguments.

1. **creating property/data members**

There are two types of data members :

1. Non-static:

They access with respect to object name so, they are like object level data members and memory space is created each and every time whenever an object is created. They are used to store specific values.

1. static:

They access with respect to class name so, they are like class level data members and memory space is created only once irrespective of the creation of objects, They are used to store common values.

code:

**package** Introduction;

**public** **class** DataMembers {

// Declaring static data members - they are fixed

**static** String *FullName* = "SowmyaLakkireddy";

**static** **int** *rollNo* = 25;

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

// Declaring Instance or Non-static data members - we can changethe

// values

String name = "Sowmya";

**int** age = 25;

**double** height = 5.4;

}

}

1. **what is data type and different data types**

Data type - It is a type of data/information which the variable holds.

Different data types:

1. **primitive data types:**

They are predefined by the language and specified by a keyword. They have fixed size.

Byte - made of 8bits and used to save space in Arrays, which is 4 times smaller than int.

Default value = 0, min = -128(-2^7), max = +127(+2^7-1)

Short - made of 16bit signed 2’s complement integer, which is 2 times smaller than int

Default value = 0, min = -32,768(-2^15), max = +32,767(+2^15-1)

Int - made of 32bit signed 2’s complement integer

Default value = 0, min =- 2,147,483,648(-2^31), max = 2,147,483,647(+2^31-1)

Long - made of 64bit signed 2’s complement integer

Default value = 0, min=-2^63,max=+2^63 -1

Float - It is a single-precision 32-bit IEEE 754 floating point

Default value = 0.0f

Double - It is a double-precision 64-bit IEEE 754 floating point

Default value = 0.0d

Boolean - False and true, Default value=false

Char- It a single 16-bit unicode character, min value = 0, max = 65,535

**b) Reference/Object data types:**

These represents arrays and class objects. So, whenever a new object is created then its reference will be stored in the variable instead of the object and while accessing the code these references will be helpful in locating the object/array

Default value = null

1. **What is variable?**

It is a storage location which holds some information (different data types) in the system’s memory. So, when you create a variable then it reserves some space in the memory based on the data type.

1. **creating method with void**

Void method: It doesn’t return any value

**Code:**

**package** Methods;

**import** java.util.Date;

**public** **class** DatatypeMethods {

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

// **TODO** Auto-generated method stub

**int** a = 1;

**int** b = 2;

**int** m = 10;

**int** n = 5;

*Person*(a, b);

MethodsReturn mr = **new** MethodsReturn();

Date date = **new** Date();

System.***out***.println(mr.add(m, n));

System.***out***.println(mr.substract(m, n));

System.***out***.println(mr.result());

System.***out***.println(mr.multiple(m, n));

System.***out***.println(date);

}

**public** **static** **void** Person(**int** a, **int** b) {

**int** sum = a + b;

System.***out***.println(sum);

}

}

1. **creating variable, we can create variables inside method**

We can create non-static variables inside the method but, we cannot create static variable inside the method because they belong to the class level.

**Example:**

**package** Introduction;

**public** **class** DataMembers {

// Declaring static data members - they are fixed

**static** String *FullName* = "SowmyaLakkireddy";

**static** **int** *rollNo* = 25;

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

// Declaring Instance or Non-static data members - we can change the

// values

String name = "Sowmya";

**int** age = 25;

**double** height = 5.4;

}

}

1. **creating method with return data type, int/string/double/float/date**

Step 1- create a class with all methods

**package** Methods;

**public** **class** MethodsReturn {

**public** **int** add(**int** m, **int** n) {

**return** m + n;

}

**public** String names() {

**return** "sowmya";

}

**public** **double** substract(**int** m, **int** n) {

**return** m - n;

}

**public** **float** multiple(**int** m, **int** n) {

**return** m \* n;

}

**public** **boolean** result() {

**return** **false**;

}

**public** String date(){

**return** **null**;

}

}

Step2: now create main class, then create object for the above class and invoke all the methods

import java.util.Date;  
  
**package** Methods;

**import** java.util.Date;

**public** **class** DatatypeMethods {

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

// **TODO** Auto-generated method stub

**int** a = 1;

**int** b = 2;

**int** m = 10;

**int** n = 5;

*Person*(a, b);

MethodsReturn mr = **new** MethodsReturn();

Date date = **new** Date();

System.***out***.println(mr.add(m, n));

System.***out***.println(mr.substract(m, n));

System.***out***.println(mr.result());

System.***out***.println(mr.multiple(m, n));

System.***out***.println(date);

}

**public** **static** **void** Person(**int** a, **int** b) {

**int** sum = a + b;

System.***out***.println(sum);

}

}

1. **method that will return hard coded value**

It returns hard coded value sowmya

**package** Methods;

**public** **class** MethodsReturn {

**public** String names() {

**return** "sowmya";

}

}

Output: sowmya

1. **method that will return property value**

Here the method add will return value based on the data members.

**package** Methods;

**public** **class** MethodsReturn {

**public** **int** add(**int** m, **int** n) {

**return** m + n;

}

}

Output: 15

**16. Create default/paramterzied constructors (overloaded constructor)**

**Constructor:**

A 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 required, or default values at the time of object creation. It is not mandatory for the coder to write a constructor for the class.

* It adds lot on readability and usability of class.
* We can use this () to refer the same class and super () to refer parent class
* If we don’t provide constructor then default constructor will be created and it will assign the default values for the variables.
* We cannot inherit constructors from parent to child.

**Rules:**

It has the same name as the class

It should not return a value not even void

Constructors cannot be abstract, static, final or synchronized

It can have all four accessibility modifiers: private , public, protected, default

It can have parameters

It can have throws clause: we can throw exception from constructor.

It can have logic, as part of logic it can have all java legal statement except return statement with value.

**Why constructor overloading?**

* Constructor overloading is done to construct object in different ways.
* Constructor overloading is a technique in Java in which a class can have any number of constructors that differ in parameter lists. The compiler differentiates these constructors by taking into account the number of parameters in the list and their type.
* useful when you want to use same class name to initialize variables of different types datatypes or different number of variables.

**Code:**

**public** **class** Constructors {

**private** String name;

**private** **int** age;

**private** **double** height;

**public** Constructors() { //default constructor

}

**public** Constructors(String name, **int** age) { //parameterized constructor with 2 parameters

**this**.name = name;

**this**.age = age;

}

**public** Constructors(String name, **int** age, **double** height) { ////parameterized constructor with 3 parameters

**this**.name = name;

**this**.age = age;

**this**.height = height;

}

**void** display(){

System.***out***.println(name + " " + age + " " + height);

}

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

Constructors d1 = **new** Constructors();

Constructors d2 = **new** Constructors("sowmya",25);

Constructors d3 = **new** Constructors("sruthi",30,5.4);

d1.display();

d2.display();

d3.display();

}

}

**Output:**

null 0 0.0

sowmya 25 0.0

sruthi 30 5.4

**Encapsulation – (hiding of data) – getter and setters**

* The whole idea behind encapsulation is to hide the implementation details from users. If a data member is private it means it can only be accessed within the same class. No outside class can access private data member (variable) of other class. However if we setup public getter and setter methods to update (for e.g. void setSSN(int ssn))and read (for e.g. int getSSN()) the private data fields then the outside class can access those private data fields via public methods. This way data can only be accessed by public methods thus making the private fields and their implementation hidden for outside classes. That’s why encapsulation is known as data hiding.
* The reason for using getters and setters instead of making your members public is that it makes it possible to change the implementation without changing the interface.
* **Encapsulation can be achieved by**: Declaring all the variables in the class as private and writing public methods in the class to set and get the values of variables.

**Advantages of Encapsulation:**

**Data Hiding**: The user will have no idea about the inner implementation of the class. It will not be visible to the user that how the class is storing values in the variables. He only knows that we are passing the values to a setter method and variables are getting initialized with that value.

**Increased Flexibility**: We can make the variables of the class as read-only or write-only depending on our requirement. If we wish to make the variables as read-only then we have to omit the setter methods like setName(), setAge() etc. from the above program or if we wish to make the variables as write-only then we have to omit the get methods like getName(), getAge() etc. from the above program

**Reusability**: Encapsulation also improves the re-usability and easy to change with new requirements.

**Testing code is easy**: Encapsulated code is easy to test for unit testing.

**Code:**

**Step-1:**

**package** Encapsulation;

**public** **class** Faculty {

**private** String name;

**private** **int** age;

**private** **double** height;

**public** Faculty(**int** age){

**this**.age=age;

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

System.***out***.println(name);

}

**public** **int** getAge() {

**return** age;

}

**public** **void** setAge(**int** age) {

**this**.age = age;

}

**public** **double** getHeight() {

**return** height;

}

**public** **void** setHeight(**double** height) {

System.***out***.println(**this**.height = height);

}

}

**Step-2:**

**package** Encapsulation;

**public** **class** TestingClass {

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

// **TODO** Auto-generated method stub

**int** age = 24;

Faculty f = **new** Faculty(age); //displaying direct values is not possible. we have to use get and set methods

f.setName("Lavanya");

f.setHeight(5.4);

System.***out***.println(age);

}

}

**Output:**

Lavanya

5.4

24

**Abstraction:**

abstraction is a process of hiding the implementation details from the user, only the functionality will be provided to the user. In other words, the user will have the information on what the object does instead of how it does it.

* In Java, abstraction is achieved using Abstract classes and interfaces.

**1.Abstract classes:**

* we have to declare an abstract method in abstract class only and and implementing all abstract methods in sub class is mandatory.
* We cannot create an object for abstract class (instantiation) but this can be achieved by extending the class (giving class object reference to it) Abstract methods don’t have body.

**Code:**

**Step-1:**

**package** Abstraction;

**public** **class** Faculty **extends** Students {

**private** **int** id;

**private** String department;

**public** Faculty(**int** id, String department,**int** rollNo,String name) {

**super**(rollNo,name);

**this**.id = id;

**this**.department = department;

}

**public** **int** gerId() {

**return** id;

}

**public** **void** setId(**int** id) {

**this**.id = id;

System.***out***.println(id);

}

**public** String getDepartment() {

**return** department;

}

**public** String getName(){

**return** **super**.getName();

}

**public** **void** count(){

System.***out***.println("jhkkh");

}

**public** **void** setDepartment(String Department) {

**this**.department = Department;

System.***out***.println(Department);

}}

**Step-2:**

**package** Abstraction;

**public** **abstract** **class** Students {

**private** **int** rollNo;

**private** String name;

**public** Students(**int** rollNo, String name) {

**this**.rollNo = rollNo;

**this**.name = name;

}

**public** **int** getRollNo() {

**return** rollNo;

}

**public** **void** setRollNo(**int** rollNo) {

**this**.rollNo = rollNo;

System.***out***.println(rollNo);

}

**public** String getName() {

**return** name;

}

**public** **void** setName(String name) {

**this**.name = name;

System.***out***.println(name);

}

**public** **abstract** **void** count();

}

**Step-3:**

**package** Abstraction;

**public** **class** AbstractionDemo {

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

// **TODO** Auto-generated method stub

//Students s= new Students(1,"sowmya"); // abstract class cannot instantiate

Students f = **new** Faculty(20,"Computers",12,"sowmya"); //we can give class object reference to it

System.***out***.println(f.getName());

f.count();

}

}

**2.Interfaces:**

* We cannot instantiate interfaces.
* Collection of abstract methods.
* Class can implement many interfaces and if any class does this it can use all the methods in that.
* Multiple Inheritance can be replaced with interfaces.

**Code:**

**Access Modifiers:**

They limit the scope of variables and methods depending on the type of modifier. They are 4 types

1. Private

2. Public

3. Protected

4. No access modifier

| Class | Package | Subclass | Subclass | World

| | |(same pkg)|(diff pkg)|

————————————+———————+—————————+——————————+——————————+————————

public | + | + | + | + | +

————————————+———————+—————————+——————————+——————————+————————

protected | + | + | + | + |

————————————+———————+—————————+——————————+——————————+————————

no modifier | + | + | + | |

————————————+———————+—————————+——————————+——————————+————————

private | + | | | |

+ : accessible

blank : not accessible

**Code:**

**Step:1**

**package** AccessModifiers;

**public** **class** Mango {

**int** count; //no access modifier is accessible in the same package

**private** **int** price; // within the class and use getter and setters to access the varibale

**public** String colour ="yellow"; //anywhere in java by importing package and class

**protected** **double** weight; //within the same package and sub classes of different packages

**public** **void** setPrice(**int** price){

System.***out***.println(price);

}

}

**Step:2**

**package** AccessModifiers;

**public** **class** Fruits {

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

// **TODO** Auto-generated method stub

Mango m = **new** Mango();

System.***out***.println(m.count = 14); // no access modifier variable

m.setPrice(40); // private access modifier accessed with setter

m.colour = "yellow"; // public access modifier

m.weight=55.2; // protected access modifier

}

}

**Output:**

14

40

**Step:3 (different package)**

**package** MethodsAndClasses;

**import** AccessModifiers.Mango;

**public** **class** FruitsDiffPackage **extends** Mango {

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

// **TODO** Auto-generated method stub

Mango m = **new** Mango();

System.***out***.println(m.colour);

}

}

**Output:**

Yellow

**creating method with return data type and parameter**

**public** **int** number(**int** m){

**return** m;

}

//here int is data type and m is parameter

**creating static property and static method:**

**package** Methods;

**public** **class** StaticPropertyAndMethod {

//static variables and methods can be declared before main method

//because they are class level but they can be accessed inside main method

**static** String *name* = "Sowmya";

**static** **int** *age*= 25;

**static** **void** output(){

System.***out***.println("I am static");

}

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

// **TODO** Auto-generated method stub

*output*(); // no need of object creation because it is static - accessed from any class

System.***out***.println(*name*);

System.***out***.println(*age*);

}

}

**Output:**

I am static

Sowmya

25

**Create static block**

* Static block is a set of statements, which will be executed by the JVM before execution of main method.
* At the time of class loading if we want to perform any activity we have to define that activity inside static block because this block execute at the time of class loading.
* In a class we can take any number of static block but all these blocks will be execute from top to bottom.

**Code:**

**package** Methods;

**public** **class** StaticBlock {

**static**{

System.***out***.println("Hello, I am static");

}

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

// **TODO** Auto-generated method stub

System.***out***.println("Done");

}

}

**Output:**

Hello, I am static

Done

**creating object**

**code:**

**class 1: MethodsReturn**

**package** Methods;

**public** **class** MethodsReturn {

**public** **int** add(**int** m, **int** n) {

**return** m + n;

}

**public** String names() {

**return** "sowmya";

}

**public** **double** substract(**int** m, **int** n) {

**return** m - n;

}

**public** **float** multiple(**int** m, **int** n) {

**return** m \* n;

}

**public** **boolean** result() {

**return** **false**;

}

**public** **int** number(**int** m){

**return** m;

}

**public** String date(){

**return** **null**;

}

**public** **void** Nothing(){

System.***out***.println("I don't have return and parameter");

}

}

**class 2: DatatypeMethods**

**Now we have create an object for class 1 in order to access their methods**

**public** **class** DatatypeMethods {

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

// **TODO** Auto-generated method stub

**int** m = 10;

**int** n = 5;

MethodsReturn mr = **new** MethodsReturn(); //object creation

}

}

**calling method void with no return and parameter**

**public** **class** DatatypeMethods {

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

mr.Nothing();

}

}

**Output:**

I am void with no return and parameter

**calling method with return and no parameter**

**public** **class** DatatypeMethods {

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

System.***out***.println(mr.result());

}

}

**Output:**

false

**calling method with return and parameter**

**public** **class** DatatypeMethods {

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

**int** m = 10;

System.***out***.println(mr.number(m));

}

}

**Output:**

10

**calling method with return and storing the return data**

**public** **class** DatatypeMethods {

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

System.***out***.println(mr.names());

}

}

**Output:**

sowmya

**create classes under multiple packages**

**class 1: ClassOfMethods under Methods package**

**package** Methods;

**public** **class** ClassOfMethods {

**private** String company;

**private** **int** totalCount;

**public** String dashBoard(String company,**int** totalCount){

**return** company + " " + totalCount;

}

}

**class 2: ClassOfIntroduction under Introduction package**

**package** Introduction;

**public** **class** ClassOfIntroduction {

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

// **TODO** Auto-generated method stub

}

}

**calling classes under different packages**

**Here we called class from other package by importing packageName.className from other package**

**package** Introduction;

**import** Methods.ClassOfMethods; //import classes from other package

**public** **class** ClassOfIntroduction {

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

// **TODO** Auto-generated method stub

ClassOfMethods cm = **new** ClassOfMethods();

System.***out***.println(cm.dashBoard("Saibersys", 1000));

}

}

**Output:**

Saibersys 1000

**write code to handle exceptions with try/catch/finally**

**package** Exceptions;

**public** **class** TryCatchFinally {

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

// **TODO** Auto-generated method stub

**int** noOfItems = 5;

String name = "Sowmya";

**int** price = 0;

**int** discount;

**try** {

discount = noOfItems / price;

System.***out***.println(discount);

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

System.***out***.println(name + " something went wrong" + e);

} **finally** {

System.***out***.println("Done shopping");

}

}

}

**Output:**

Sowmya something went wrongjava.lang.ArithmeticException: / by zero

Done shopping

**what is checked exception/unchecked exception**

**Checked exception:**

Checked exceptions are checked at compile-time. It means if a method is throwing a checked exception then it should handle the exception using try-catch block or it should declare the exception using throws keyword, otherwise the program will give a compilation error. It is named as checked exception because these exceptions are checked at Compile time.

**Code:**

**package** Exceptions;

**import** java.io.BufferedReader;

**import** java.io.FileReader;

**import** java.io.IOException;

**public** **class** CheckedException {

**public** **static** **void** main(String[] args) **throws** IOException {

// **TODO** Auto-generated method stub

FileReader file = **new** FileReader("C:\\test\\a.txt");

BufferedReader fileInput = **new** BufferedReader(file);

// Print first 3 lines of file "C:\test\a.txt"

**for** (**int** counter = 0; counter < 3; counter++)

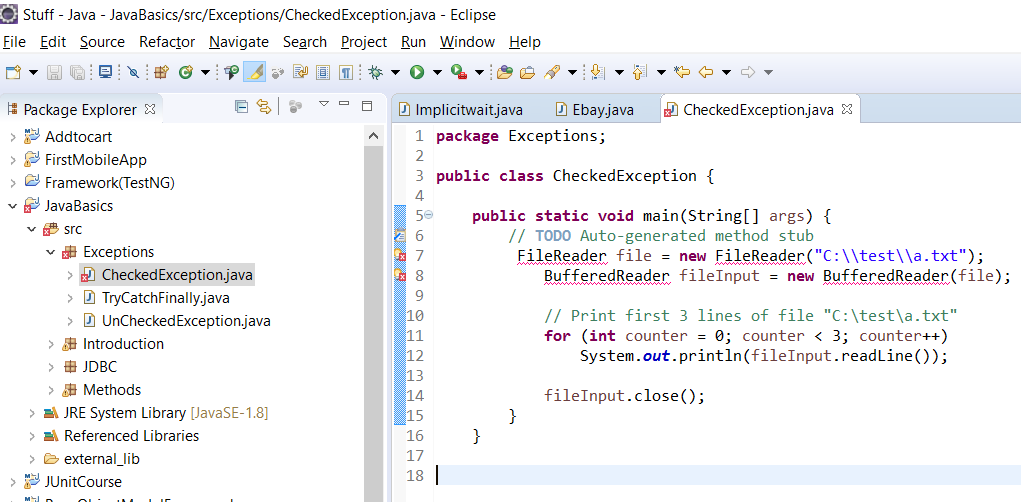
System.***out***.println(fileInput.readLine());

fileInput.close();

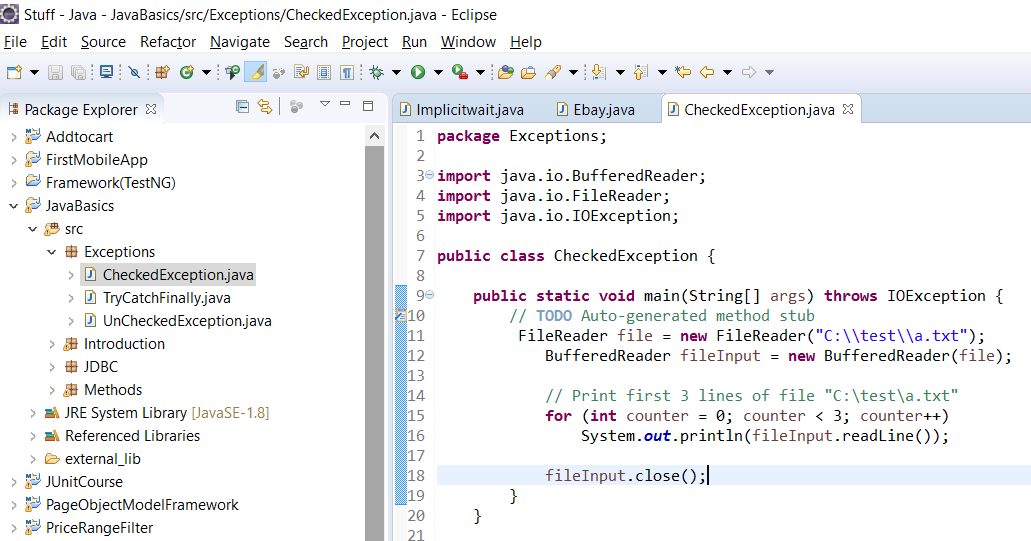
}

}

* Error which came during compilation time



* Now, after importing all the related packages it got solved.



**Unchecked exception**

Unchecked exceptions are not checked at compile time. It means if your program is throwing an unchecked exception and even if you didn’t handle/declare that exception, the program won’t give a compilation error. Most of the times these exception occurs due to the bad data provided by user during the user-program interaction. It is up to the programmer to judge the conditions in advance, that can cause such exceptions and handle them appropriately. All Unchecked exceptions are direct sub classes of RuntimeException class.

**Examples:**

* NullPointerException
* ArrayIndexOutOfBoundsException
* ArithmeticException
* IllegalArgumentException

**Code:**

**package** Exceptions;

**public** **class** UnCheckedException {

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

// **TODO** Auto-generated method stub

**int** a = 5;

**int** b = 0;

**int** c;

**try** {

c = a / b;

System.***out***.println(c);

} **catch** (Exception e) {

// **TODO** Auto-generated catch block

System.***out***.println(" something went wrong" + e);

} **finally** {

System.***out***.println("Done");

}

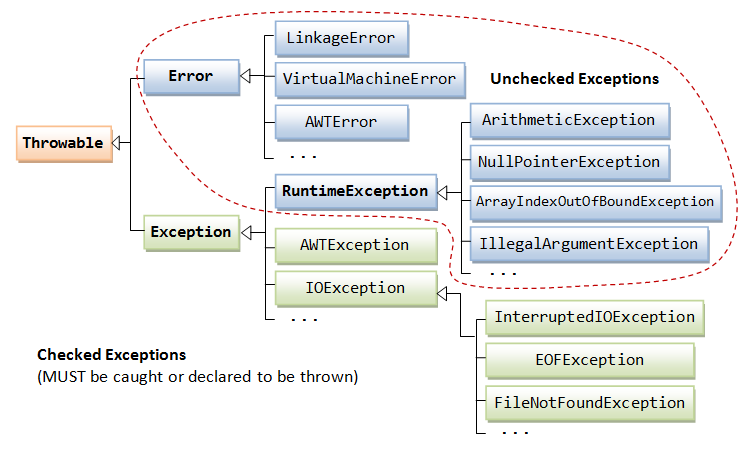
}

}

**Output:**

something went wrongjava.lang.ArithmeticException: / by zero

Done



**What is Final keyword?**

It is used to make the following ways:

**1) variable as a constant – creates constant variables**

**2) Restrict method overriding – Final Method**

**3) Restrict inheritance – Final class**

**Create final class, final method and final property**

**Final Class:**

When a class is declared with final keyword, it is called a final class, meaning that the class cannot be inheriting by other classes. When we want to restrict inheritance then make class as a final.

**Code:**

**package** MethodsAndClasses;

**final** **class** student {

}

**public** **class** DetailsClassFinal //extends student //it cannot be inherited because it restricts inheritance

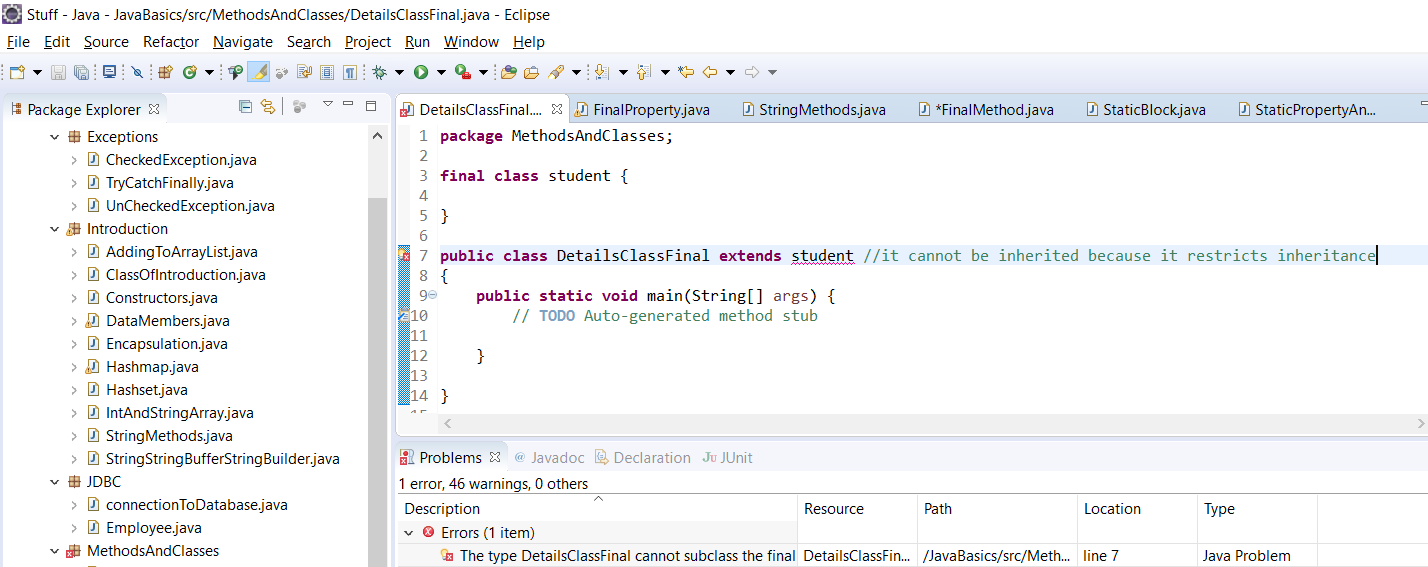
{

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

// **TODO** Auto-generated method stub

}

}



**Final Varaiable/Property:**

When a variable is declared with final keyword, it’s value can’t be modified, essentially, a constant. This also mean that you must initialize a final variable

**Code:**

**package** MethodsAndClasses;

**public** **class** FinalProperty {

**final** **int** rollNo = 34;

// void number() { //here we cannot change the variable value because it is constant/fixed

// rollNo = 50;

// }

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

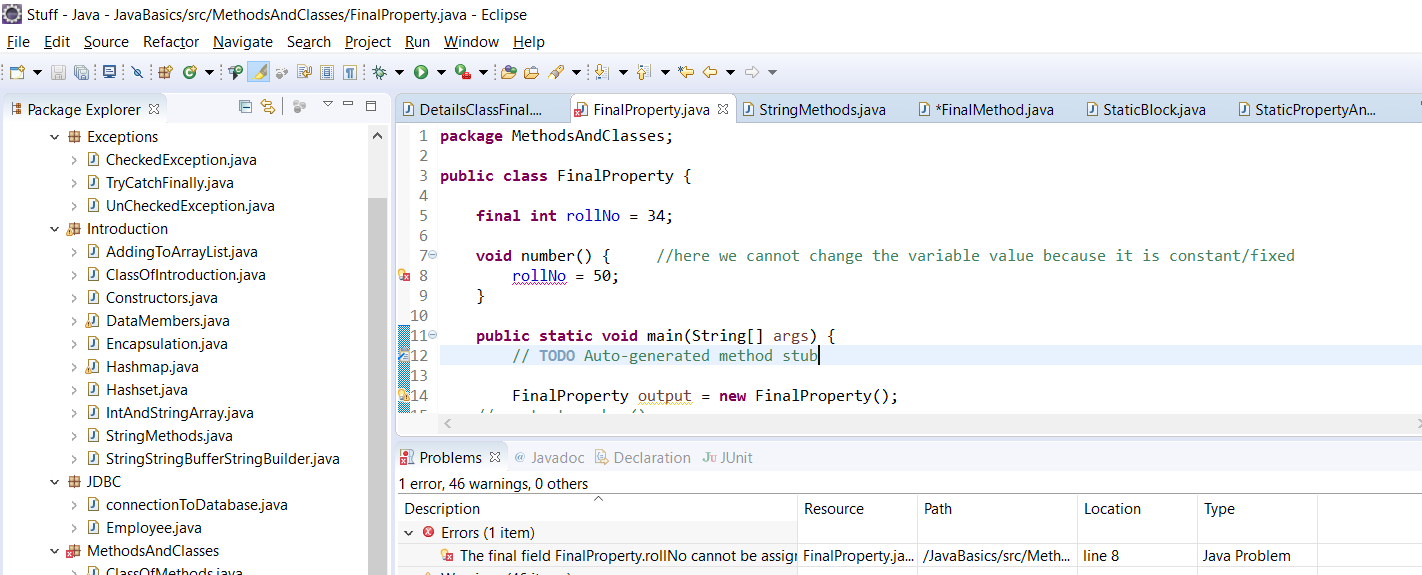
// **TODO** Auto-generated method stub

FinalProperty output = **new** FinalProperty();

// output.number();

}

}



**Final Method:**

When a method is declared with final keyword, it is called a final method. A final method cannot be overridden.

**Code:**

**package** MethodsAndClasses;

**class** school {

**final** **void** count() {

System.***out***.println("total 50");

}

}

**public** **class** DetailsFinalMethod **extends** school {

// void count() {

// System.out.println("total 100");

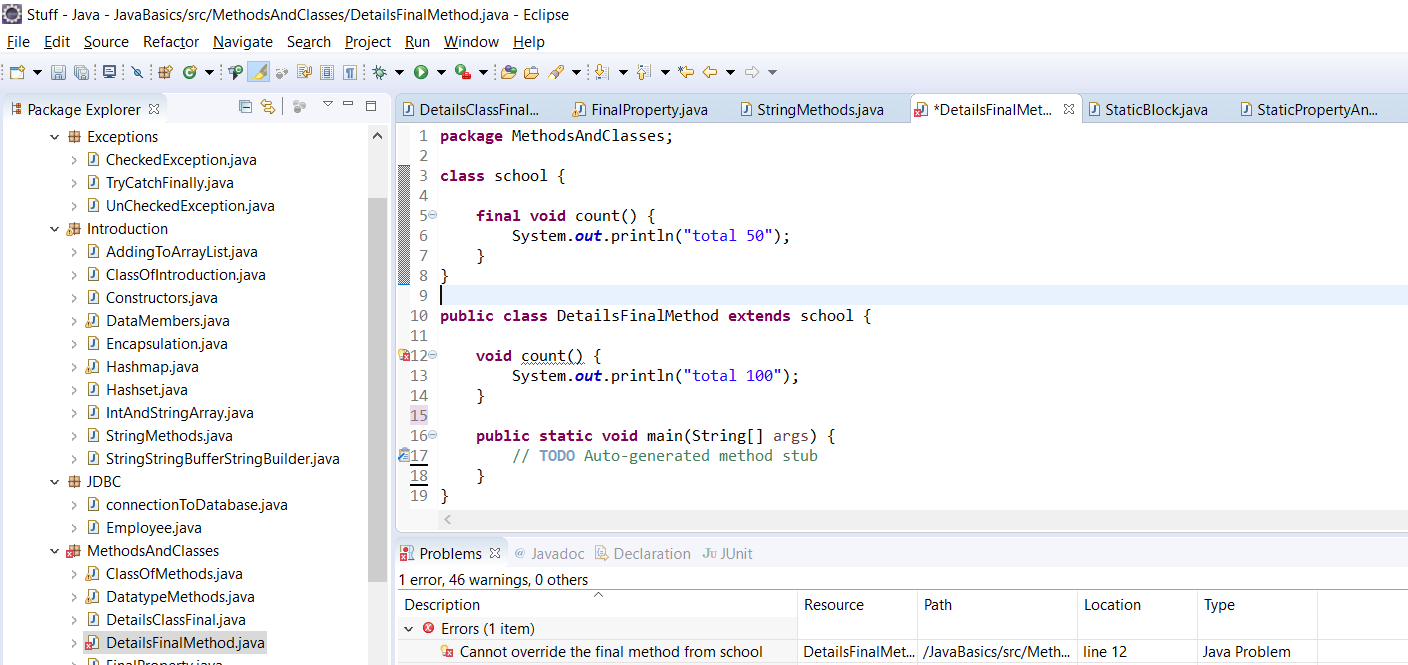
// }

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

// **TODO** Auto-generated method stub

}

}



| **BASIS FOR COMPARISON** | **STATIC** | **FINAL** |
| --- | --- | --- |
| Applicable | Static keyword is applicable to nested static class, variables, methods and block. | Final keyword is applicable to class, methods and variables. |
| Initialization | It is not compulsory to initialize the static variable at the time of its declaration. | It is compulsory to initialize the final variable at the time of its declaration. |
| Modification | The static variable can be reinitialized. | The final variable can not be reinitialized. |
| Methods | Static methods can only access the static members of the class, and can only be called by other static methods. | Final methods can not be inherited. |
| Class | Static class's object can not be created, and it only contains static members only. | A final class can not be inherited by any class. |
| Block | Static block is used to initialize the static variables. | Final keyword supports no such block. |

**How to do inheritance in java (using extend keyword)**

Inheritance in java is a mechanism in which one object acquires all the properties and behaviors of parent object.

* Inheritance represents the **IS-A** relationship, also known as parent-child relationship. It doesn’t support multiple inheritance.
* For invoking base class constructor we have to use super keyword for example super (argument in constructor) and for methods we have to use super. (methodName)

Advantages:

* For Method Overriding (so runtime polymorphism can be achieved).
* For Code Reusability.

**Coding:**

**Step-1**

**package** Inheritance;

**public** **class** Car {

**int** speed;

**public** Car(**int** startSpeed) {

**this**.speed = startSpeed;

}

**public** **void** increaseSpeed() {

System.***out***.println(speed++);

}

**public** **void** decreaseSpeed() {

System.***out***.println(speed--);

}

**public** String toString() {

**return** "Total speed is";

}

}

**Step-2**

**package** Inheritance;

**public** **class** Celerio **extends** Car {

**public** Celerio(**int** startSpeed) {

**super**(startSpeed);

}

**public** **void** decreaseSpeed(){

**super**.decreaseSpeed();

}

**public** String toString(){

**return** **super**.toString()+ "=" + speed--;

}

}

**Step-3**

**package** Inheritance;

**public** **class** TestingClass {

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

// **TODO** Auto-generated method stub

**int** speed=11;

Car c=**new** Car(speed);

System.***out***.println(c.toString());

Celerio cl = **new** Celerio(speed);

cl.increaseSpeed();

cl.decreaseSpeed();

System.***out***.println(cl.toString());

}

}

**Output:**

Total speed is

11

12

Total speed is=11

**18.write code to add items to integer, string array**

1. Adding items into integer array

**int**[] sum = **new** **int**[3];

sum[0] = 1;

sum[1] = 2;

sum[2] = 3;

(or)

Int[] values = {1,2,3};

1. Adding items into string array

String[] values = **new** String[4];

values[0] = "sowmya";

values[1] = "roja";

values[2] = "pallavi";

values[3] = "vrundha";

(0r)

String[] names = {“Sowmya”,”Roja”,”Pallavi”,”Vrundha”}

**19.** **write code to retrieve items from integer, string array**

Retrieving items from integer array

**for** (**int** s : sum) {

System.***out***.println(s);

}

Output:

1

2

3

Retrieving items from string array

Arrays.*sort*(values);

**for** (String v : values) {

System.***out***.println(v);

}

Output:

Pallavi

Roja

Sowmya

Vrundha

**20.write code to add items to ArrayList collection**

ArrayList<String> animals = **new** ArrayList<String>();

animals.add("dog");

animals.add("cat");

animals.add("monkey");

ArrayList<Integer> values = **new** ArrayList<Integer>();

values.add(1);

values.add(2);

values.add(3);

values.add(4);

**21.write code to retrieve items from arraylist (using for each loop)**

System.***out***.println(animals);

System.***out***.println(animals.size());

**for** (String a : animals) {

System.***out***.println(a);

}

Output:

[dog, cat, monkey]

3

dog  
cat  
monkey

System.***out***.println(values);

**for** (**int** v : values) {

System.***out***.println(v);

}

Output:

[1, 2, 3, 4]  
1  
2  
3  
4

**Note: Class implements Interfaces**

**Class extends Classes**

**Interface extends Interfaces**

**HashSet:**

* Implements Set Interface.
* Underlying data structure for HashSet is hashtable.
* As it implements the Set Interface, duplicate values are not allowed.
* Objects that you insert in HashSet are not guaranteed to be inserted in same order. Objects are inserted based on their hash code.
* NULL elements are allowed in HashSet.
* It is not synchronized means not thread safe (2 threads can call methods simultaneous)
* HashSet also implements Searlizable and Cloneable interfaces.

**Adding items into HashSet**

**code:**

**package** Introduction;

**import** java.util.HashSet;

**import** java.util.Iterator;

**public** **class** Hashset {

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

// **TODO** Auto-generated method stub

HashSet<String> hs = **new** HashSet<String>();

hs.add("sowmya");

hs.add("pallavi");

hs.add("Roja");

**Retrieving items from HashSet**

Iterator<String> i = hs.iterator(); // it will iterate elements one by

One element in the set

**while** (i.hasNext()) {

System.***out***.println(i.next());

}

}

}

**HashMap:**

* HashMap maintains key and value pairs and often denoted as HashMap<Key, Value> or HashMap<K, V>.
* HashMap implements Map interface.
* It is not synchronized means not thread safe (2 threads can call methods simultaneous)
* HashMap is similar to Hashtable with two exceptions – HashMap methods are unsynchornized and it allows null key and null values unlike Hashtable.

**Add items to HashMap**

**package** Introduction;

**import** java.util.HashMap;

**import** java.util.Set;

**import** java.util.Iterator;

**import** java.util.Map;

**public** **class** Hashmap {

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

// **TODO** Auto-generated method stub

HashMap<Integer, String> hmap = **new** HashMap<Integer, String>();

hmap.put(1, "sowmya");

hmap.put(2, "pallavi");

hmap.put(3, "roja");

**Retrieve items to HashMap**

System.***out***.println(hmap.get(1)); // it will give values for the given

// key

Iterator it = hmap.entrySet().iterator();

**while** (it.hasNext()) {

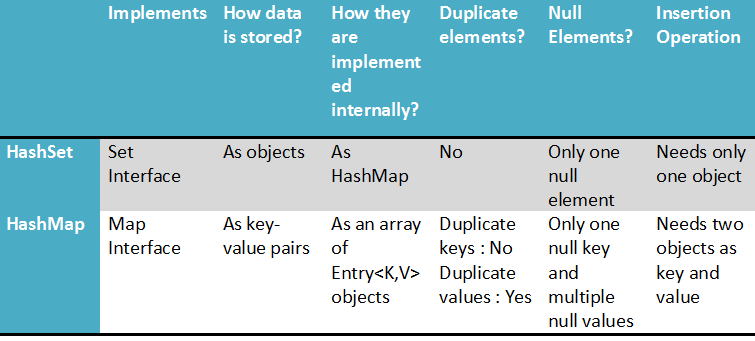
Map.Entry me = (Map.Entry) it.next();

System.***out***.println("key = " + me.getKey() + " Value = " + me.getValue());

}

}

}



**Hash Table:**

* A Hashtable is an array of list. Each list is known as a bucket. The position of bucket is identified by calling the hashcode() method. A Hashtable contains values based on the key.
* It contains only unique elements.
* It may have not have any null key or value.
* It is synchronized means thread safe (2 threads cannot call methods simultaneous)

**HashMap vs ConcurrentHashMap**

**1.  Thread -Safe :**  
  
ConcurrentHashMap is thread-safe that is the code can be accessed by single thread at a time .      
     while HashMap is not thread-safe .  
[](https://images-blogger-opensocial.googleusercontent.com/gadgets/proxy?url=http://2.bp.blogspot.com/-WsUtI3c7wrQ/Uu9tohX5HLI/AAAAAAAAAJs/7rhptkM80Yo/s1600/java-collections-tutorial.png&container=blogger&gadget=a&rewriteMime=image/*)  
**2.  Synchronization Method :**  
  
 HashMap can be synchronized by using      
    synchronizedMap(HashMap)  method .  By using this    
    method we get a HashMap object which is equivalent   
    to the HashTable object . So every modification  is performed      
    on  Map is locked on Map object.

**import** **java.util.\***;

**public** **class** **HashMapSynchronization** {

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

// create map

Map<String,String> map = **new** HashMap<String,String>();

// populate the map

map.put("1","ALIVE ");

map.put("2","IS");

map.put("3","AWESOME");

// create a synchronized map

Map<String,String> syncMap = Collections.synchronizedMap(map);

System.out.println("Synchronized map :"+syncMap);

}

}

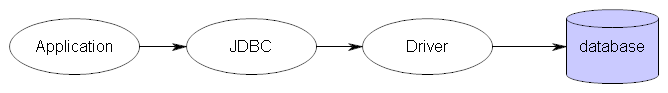
   ConcurrentHashMap synchronizes or locks on the certain portion of the Map . To optimize  
   the performance of ConcurrentHashMap , Map is divided into different partitions depending  
   upon the Concurrency level . So that we do not need to synchronize the whole Map Object.  
  
  
**3.  Null Key**  
  
  
     ConcurrentHashMap does not allow NULL values . So the key can not be null in  
     ConcurrentHashMap .While In HashMap there can only be one null key .  
  
   
**4.  Performance**   
  
     In multiple threaded environment HashMap is usually faster than ConcurrentHashMap . As      
     only single thread can access the certain portion of the Map and thus reducing the performance .   
     While in HashMap any number of threads can access the code at the same time .  
  
     Please  write in comments  in case if you have any doubts .

**22.write code to connect to JDBC to get rows from employee table**

**JDBC: (Java Database Connectivity)**

* JDBC API consists of set of classes and interfaces that help us to connect with database, insert, update and delete the entries in the table.
* With this we can write SQL queries in java code.
* One must import **java.sql** package to run JDBC.

**Flow**:



Application calls JDBC Library -> JDBC loads the driver -> driver talks with database.

**Steps to follow:**

* + Load the driver
  + use DriverManager to get a Connection
  + create a Statement to query the database

3 kinds of statements –

1. **statement**- for executing simple queries without parameters and one time

executeQuery – can be used for running SQLqueries

executeUpdate – can be used for crud operations

2.**prepared statements** – executing queries with parameters many times

3.**callable statements** – executing stored procedures and accept input parameters

* + execute the statement
  + process the ResultSet
  + close the Statement
  + close the Connection

1. **Driver class:**The driver class for the mysql database is **com.mysql.jdbc.Driver**.

At first, we have to load the driver that is Class.forName**(“com.mysql.jdbc.driver”);**

1. Use DriverManager to get a connection. For this we should have
   1. **URL: (database service)**

The connection URL for the mysql database is **jdbc:mysql://localhost:3306/sonoo**  where jdbc is the API, mysql is the database, localhost is the server name on which mysql is running, we may also use IP address, 3306 is the port number and sonoo is the database name. We may use any database, in such case, you need to replace the sonoo with your database name.

* 1. **Username:**The default username for the mysql database is **root**.
  2. **Password:**Password is given by the user at the time of installing the mysql database. In this example, we are going to use root as the password.

1. Now, once the connection is done we should query the database (to send sql queries to database) for that we have to create a statement

Statement st = conn.createStatement();

1. Once a statement is created we should execute the statement using the statement object which is “**st**”.

st.executeQuery(query) [query = Select \* from employee]

1. Now, whatever the data which we got when we execute the query is sent to the result set so, for that we should create a **ResultSet** object and send the data which we got.

ResultSet **rs** = st.executeQuery(select \* from employee)

1. Now, we have to process the **ResultSet** for that we have many methods like first, beforeFirst, last, afterLast, next, previous. For our scenario, we have to take next method (it moves the cursor to the next row and returns false when there are no more rows )
2. We can also have ResultSet methods for getting the data like dbResultSet.getString(“column name”) – it will return the value of the column name form the current row.

(or)

dbResultSet.getString(2)- returns the value of field in column2 for the current row (indexing starts from 1).

1. Close the statement and close the connection.

**Important:** we have to download jdbc jar file because jdbc is an API and we have to import all jar files to make a connection between java application and database mysql.

**Code:**

package JDBC;

import java.sql.Connection;

import java.sql.DriverManager;

import java.sql.ResultSet;

import java.sql.Statement;

import java.util.ArrayList;

public class connectionToDatabase {

ArrayList<Employee> list = new ArrayList<Employee>();

public static void main(String[] args) {

try{

//Class.forName("com.mysql.jdbc.driver"); // loads the driver class

Connection con = DriverManager.getConnection("jdbc:mysql://localhost:3306/demo", "root", "root"); //creates conneection with the database

Statement st = con.createStatement();// create statement

ResultSet result = st.executeQuery("select \* from employee ");//execute the statement and return the result into the ResultSet

while(result.next())//processing the ResultSet

{

System.out.println(result);

}

result.close();

st.close();

con.close();

}catch(Exception ex){

System.out.println("Message " + ex);

}

}

}

**Add employee class to list collection**

ArrayList<Employee> list = new ArrayList<Employee>();

**Create Employee class**

**package** JDBC;

**public** **class** Employee {

**private** **int** idemployee;

**private** String Name;

**private** **int** Age;

**private** String State;

**public** Employee(**int** id, String name, **int** age, String state) {

**this**.idemployee = id;

**this**.Name = name;

**this**.Age = age;

**this**.State = state;

}

**public** **int** getId(**int** id) {

**return** idemployee;

}

**public** **void** setId(**int** idemployee) {

**this**.idemployee = idemployee;

}

**public** String getName(String name) {

**return** Name;

}

**public** **void** setName(String name) {

**this**.Name = name;

}

**public** **int** getAge(**int** age) {

**return** Age;

}

**public** **void** setAge(**int** age) {

**this**.Age = age;

}

**public** String getState(String state) {

**return** State;

}

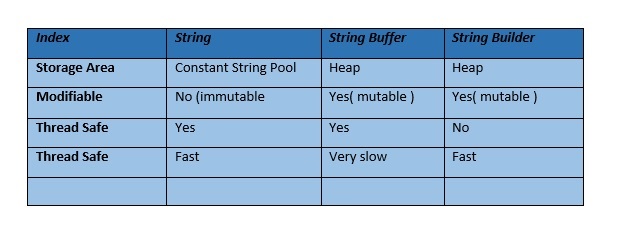
**public** **void** setState(String state) {

**this**.State = state;

}

}

**Difference between string, string buffer and string builder with example**



**Code:**

**package** Introduction;

**public** **class** StringStringBufferStringBuilder {

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

// create space in string constant pool as it is String literal

String str = "Saibersys";

str += "hello";

System.***out***.println(str);

// create space in heap memory

// creates another ref as it is immutable

String a = **new** String("Amensys");

a += "Hi";

System.***out***.println(a);

// create space in heap memory

// doesn't create another space as it is mutable

StringBuffer b = **new** StringBuffer("Good");

b.append("work");

System.***out***.println(b);

// same like StringBuffer but it is not thread safe

StringBuilder c = **new** StringBuilder("Keep");

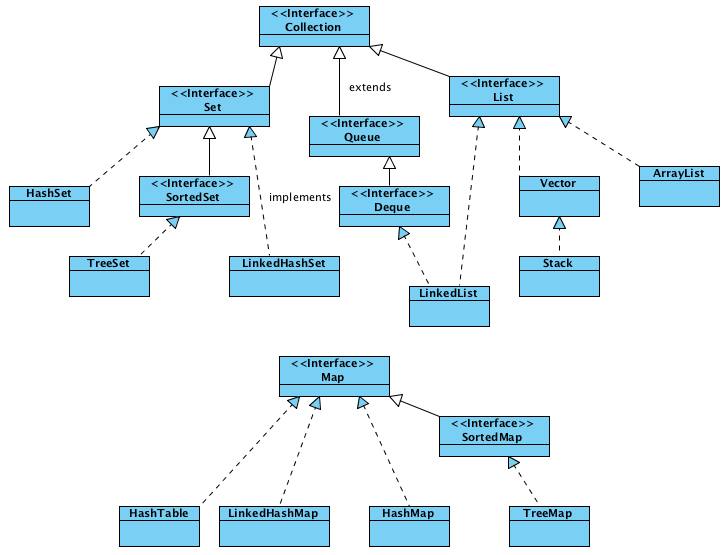
c.append(" it up");

System.***out***.println(c);

}

}

List of interfaces/classes in collection package



**What is a singleton in Java?**

This property is useful to create **singleton** class in **java**. **Singleton** pattern helps us to keep only one instance of a class at any time. The purpose of **singleton** is to control object creation by keeping private constructor.