**OOPs(Object Oriented Programming system)**

<http://beginnersbook.com/2013/04/oops-concepts/>

<https://www.youtube.com/watch?v=Mp4Fy9TDU_Y&list=PLBKgXPzyseMzVJxZT9NnpfnesKThrqslL>

1. **Data Hiding:**

Outside person can’t access our internal data directly or our internal data shouldn’t go out directly. This OOP feature is called Data hiding.

After validation or authentication outside person can access our internal data.

Example: 1) After providing proper user name and password, we can abele to access our GMAIL inbox information.

Example : 2) Even though we are valid customers of the bank, we can able to able to access our account information and we can’t access others account information.

How can we achieve Data hiding?

By declaring data member(variable) as a private, we can achieve data hiding.

Public class Account {

Private Double balance;

Public Double getBalance(){

//Validation whether logged in user is valid or not

return balance;

}

}

**The main advantage of Data hiding is Security.**

**Note:** It is highly recommended to declare data member(variable) as a **private**

1. **Abstraction**

Hide internal implementation and just highlight set of services we are offering, is a concept of **abstraction**.

Example: ATM GUI screen, services like WITHDRAW, CHECK BALANCE, MINI STATEMENT etc., without highlighting internal implementation.

**Advantages**:

a) Outside person doesn’t aware of internal implementation i.e., we achieve security.

b) No impact for the end user, since we can change internally if requires. Enhancement will become easy.

c) Abstraction improves easiness to the system. Ex: We no need to know internal implementation of the ATM to perform operations, we just need to know how to use ATM card.

d) Maintainability will be easy.

**By using Interfaces and Abstract classes, we can implement Abstraction.**

1. **Encapsulation**

The process of binding data and corresponding methods into a single unit is nothing but encapsulation.

Example: Every Java class is an example of Encapsulation (Data members + methods).

If any component follows Data Hiding and Abstraction, such type of component is called Encapsulation.

**Encapsulation = Data Hiding + Abstraction**

A simple POJO class is an example

**Advantages**:

a) We can achieve security

b) Enhancement will become easy

c) It improves maintainability of the application

**Disadvantages**:

Slows down the execution.

1. **Tightly coupled**

A class is said to be Tightly encapsulated, if and only if each and every variable declared as **private.** Whether class contains corresponding getter and setters are not and whether these methods are declared as public or not, these things we are not required to check.

Example: A Pojo class

**Note:** If a Parent class is not tightly coupled, then child classes will not be tightly coupled.

1. **IS-A relationship (Inheritance)**

The main advantage of IS-A relationship is code reusability.

By using **EXTENDS** keyword, we can implement IS-A relationship.

Class Parent { m1}

Class Child extends Parent {m2}

1. Whatever methods Parents has by default available to the child. And hence on the child reference we can call both Parent and Child class methods.

Child c = new Child();

c.m1();

c.m2();

1. Whatever methods child has by default not available to the Parent. Hence on the Parent reference we can’t call child specific methods.

Parent p = new Parent();

p.m1();

p.m2(); -- Throws compile time error

1. Parent reference can be used to hold Child Object. But using that reference we cannot call child specific methods. But, we can call the methods present in Parent class.

Parent p = new Child();

p.m1();

p.m2(); throws compile time error

1. Parent reference can be used to hold Child Object, but child reference cannot be use to hold parent Object.

Child c = new Parent(); Compile time error

Most common methods which are applicable for any type of child, we have to define in Parent class. The specific methods which are applicable for a particular child we have to define in child class.

Total Java API is implemented based on Inheritance concept. The most common methods which are applicable for any Java object are defined in **Object** class. And hence every class in Java is the child class of Object, either directly or in-directly.

So that, Object class methods by default available to every Java class without re-writing. Due to this Object class acts as root for all Java classes.

**Throwable** class defines the most common methods which are required for every Exception and Error classes.

**Java won’t support for multiple Inheritance.**

**Multiple Inheritance:** A Java class can’t extend more than one class at a time. Hence, Java won’t provide support for multiple inheritance in classes.

**Note:** If our class doesn’t extend any other class, then only our class is direct child class of Object.

If our class extends any other class, then our class is indirect child class of Object.

Either directly or indirectly, Java won’t provide support for inheritance with respected to classes.

**Java supports only Multilevel Inheritance.**

**Why Java won’t provide support for Multiple Inheritance?**

There may be a chance of Ambiguity Problem. Hence, Java won’t provide support for Multiple Inheritance.

P1{m1}

P2{m1}

C extends P1, P2 { c.m1} – Compile time error. Compiler don’t know which class m1 it calls.

But Interface can extend any number of Interfaces simultaneously. Hence, Java provide support for Multiple Inheritance with respect to Interfaces.

**Why Ambiguity problem won’t be there in Interfaces?**

Even though multiple method declarations are available, but implementation is unique. Hence there is no chance of ambiguity problem in Interfaces.

**Note:** Strictly speaking, through Interfaces we won’t get any Inheritance.

**Cyclic Inheritance:** It is not allowed in Java. Of course, it is not required.

Example class A extends A – A is child of A.

Class A extends B and Class B extends A

Throws compile time error in both scenarios (Cyclic Inheritance involving A).

**Advantages**: Reusability of code. Code redundancy is reduced.

1. **HAS-A relationship:**

HAS-A relationship is also known as Composition/Aggregation.

There is no specific keyword to implement HAS-A relation, but most of the times, we are depending on **new** keyword.

The main advantage of HAS-A relationship is reusability of the code.

Ex: Class car { Engine e = new Engine();}

Car **has a** engine reference.

**Difference between Composition and Aggregation?**

Ex: University has a multiple branch. University is called Container Object and Branches Are Contained Object.

If University is closed, then there are no branches. In same way, if Container Object is destroyed, then Contained Objects also get destroyed.

The bonding between Container and Contained objects are very strong. This is called **Composition.**

**Aggregation:** Without existing Container Object, if there is a chance of Existing Contained Object, then container and contained objects are weakly associated. And this weak association is nothing but **Aggregation**.

Ex: Department consists of several professors, without existing department there may be chance of existing professor objects. Hence, department and professor objects are weakly associated and this weak association is nothing but **Aggregation**.

**Note**: In Composition, Container Object holds directly contained Objects where as in Aggregation Container Object holds just references of Contained Objects.

**IS-A vs HAS-A:**

If we want total functionality of a class automatically then we should go for IS-A relationship.

Ex: Person class total is required for Student class.

If we want part of the functionality, then we should go for HAS-A relationship.

Ex: Test class contain 100 methods, within Demo class I need 1 or 2 methods, then create an object of Test in Demo class.

1. **Method Signature**

In Java, method signature consists of method names followed by argument types.

Public static int m1(int i, float f)

Method signature for above is: m1(int, float)

**Note**: Return type is not part of method signature in Java.

Compiler will use method signature to resolve method calls.

1. **Overloading**

2 methods are said to be Overloaded if and only if both methods having same name but different argument types.

**Note**: In Overloading method resolution is always takes care by compiler based on reference type. Hence, Overloading is also considered as **Compile Time Polymorphism** or **Static Polymorphism** or **Early Binding.**

In Overloading, method resolution is taken care by Compiler based on Reference Type.

**Case 1: Automatic promotion in overloading**

While resolving overloaded methods if exact matched method is not available then we won’t get any compile time error immediately. First it will promote argument to the next level and check whether method is available or not. If matched method is available, then it will be considered. If matched method is not available, then compiler promotes argument once again to the next level. This process will be continued until all possible promotions. Still if the method is not available then we will get compile time error.

The following are all possible promotions in overloading.

Byte-> short->int->long->float->double

Char-> int->long->float->double.

This process is called Automatic Promotion in overloading.

**Case 2:**

Class Test { public void m1(String str){} public void m1(Object obj){}

p.s.v.m(String args[]) {Test t = new Test();

t.m1(null) – calls String argument method}

While resolving overloaded methods compiler will always give the precedence for child type argument when compared with Parent type argument.

**Case 3:**

Class Test { public void m1(StringBuffer sb){} public void m1(String s){}

p.s.v.m(String args[]) {Test t = new Test();

t.m1(null) – Compile time error}

Because String and String buffer are child of Object. Compiler don’t know to call which method to call. Hence, it shows ambiguity error.

**Case 4:**

**C**lass Test { public void m1(int i, float f){} public void m1(float f, int i){}

p.s.v.m(String args[]) {Test t = new Test();

t.m1(10, 10) – Compile time error. Reference to m1 is ambiguous.

t.m1(10.5f, 10.5f) – Compile time error}

**Case 5:**

**C**lass Test { public void m1(int i){} public void m1(int … x){}

p.s.v.m(String args[]) {Test t = new Test();

t.m1() – Var args

t.m1(10) general method

}

In general, vararg method will get least priority, that is if no other matched then only vararg method will get the chance. It is exactly same as default case inside switch.

**Case 6:**

Class Animal{} Class Monkey extends Animal{}

Class Test {

Public void m1(Animal a)

Public void m1(Monkey m)

p.s.v.m {

Test t = new Test();

Animal a1 = new Monkey();

t.m1(a1) – Animal version

}

}

In overloading method resolution always takes care by compiler based on reference type.

1. **Overriding**

Whatever methods Parent has by default available to the child through inheritance. If child class not satisfied with Parent class implementation, then child is allowed to re-define that method based on its requirement.

This process is called **Overriding**.

The parent class method which is Overridden is called Overridden method and the child class method which is Overriding is called Overriding method.

In Overriding method resolution is always based on runtime object which is takes care by **JVM**. Hence, it is called **Runtime Polymorphism** or **Late Binding** or **Dynamic Polymorphism**.

**Rules for Overriding:**

* + - * + In Overriding method names and argument types must be matched i.e., method signatures must be same.
        + In Overriding return types must be same, but this rule is applicable until 1.4 version only. From 1.5 version onwards we can take co-variant return types. According to this child class method return type need not to be same as parent method return type. Its child type also allowed.
        + Co-variant return type applied only for wrapper classes not to primitive types
        + Parent class private methods not available to the child. And hence Overriding concept not applicable for Private methods.
        + Based on requirement, we can define exactly same private method in child class, it is valid but not overriding
        + We can’t override parent class final methods in child classes. If we are trying to override we will get compile time error.
        + Parent class abstract methods we should Override in child class to provide implementation.
        + We can override **non-abstract method** as **abstract.** The main advantage of this approach is we can stop the availability of Parent method implementation to the next level child classes.
        + In Overriding the following modifiers won’t keep any restriction.

Synchronized

Native

Strictfp

* + - * + While Overriding we can’t reduce scope of Access Modifier. But we can increase the scope (default->protected->public) – (private<default<protected<public). If a method is Public, outsiders can access the method. While overriding it shouldn’t impact outsiders.
        + Error and its child classes and Runtime Exception and its child classes are **Unchecked Exceptions**. Other than these classes remaining all are **Checked Exceptions**.
        + If Child class method throws any **Checked Exception** Compulsory Parent class method should throw the **same Checked Exception** or **its Parent**. Otherwise, we will get **Compile time error.** But there are no restrictions for **Unchecked Exceptions**.
        + Cases on Overriding in Exceptions:

Parent: public void m1() throws Exception

Child: public void m1()

This is a valid scenario

Parent: public void m1()

Child: public void m1() throws Exception

This is an invalid scenario

Parent: public void m1() throws Exception

Child: public void m1() throws IOException

This is a valid scenario

Parent: public void m1() throws IOException

Child: public void m1() throws Exception

This is an invalid scenario

Parent: public void m1() throws IOException

Child: public void m1() throws FileNotFoundException, EOFException

This is a valid scenario

Parent: public void m1() throws IOException

Child: public void m1() throws EOFException, InterruptedException

This is an invalid scenario

Parent: public void m1() throws IOException

Child: public void m1() throws NPE, ArithematicException

This is a valid scenario

* If Child Class method exception should always lower than Parent because, outsiders who are calling the method shouldn’t be impacted.

**Overriding w.r.t static methods:**

* We can’t override a static method as non-static. Otherwise we will get compile time error
* Similarly, we can’t override a non-static method as static
* If both Parent and Child class methods are static, then we won’t get any compile time error. It seems Overriding concept applicable for static methods. **But it is not overriding and it is method hiding.**

**Method Hiding:**

All rules of ***method hiding are exactly same as Overriding*,** except the following differences:

|  |  |
| --- | --- |
| **Method Hiding** | **Overriding** |
| Both Parent and Child class methods should be static | Both Parent and Child class methods should be non-static |
| Compiler is responsible for method resolution based on Reference Type | JVM is always responsible for method resolution based on Runtime Object. |
| It is also known as Compile time Polymorphism or Static Polymorphism or Early binding | It is also known as Runtime Polymorphism or Dynamic Polymorphism or Late binding |
|  |  |

Class P {public static void m1()}

Class C extends C {public static void m1()}

Class Test {

p.s.v.m() {

P p = new C();

p.m1(); ----- Calls m1 method of P because of static method (Compiler is responsible for method resolution based on Reference type).

}

}

If both Parent and Child class methods are non-static, then it will become Overriding.

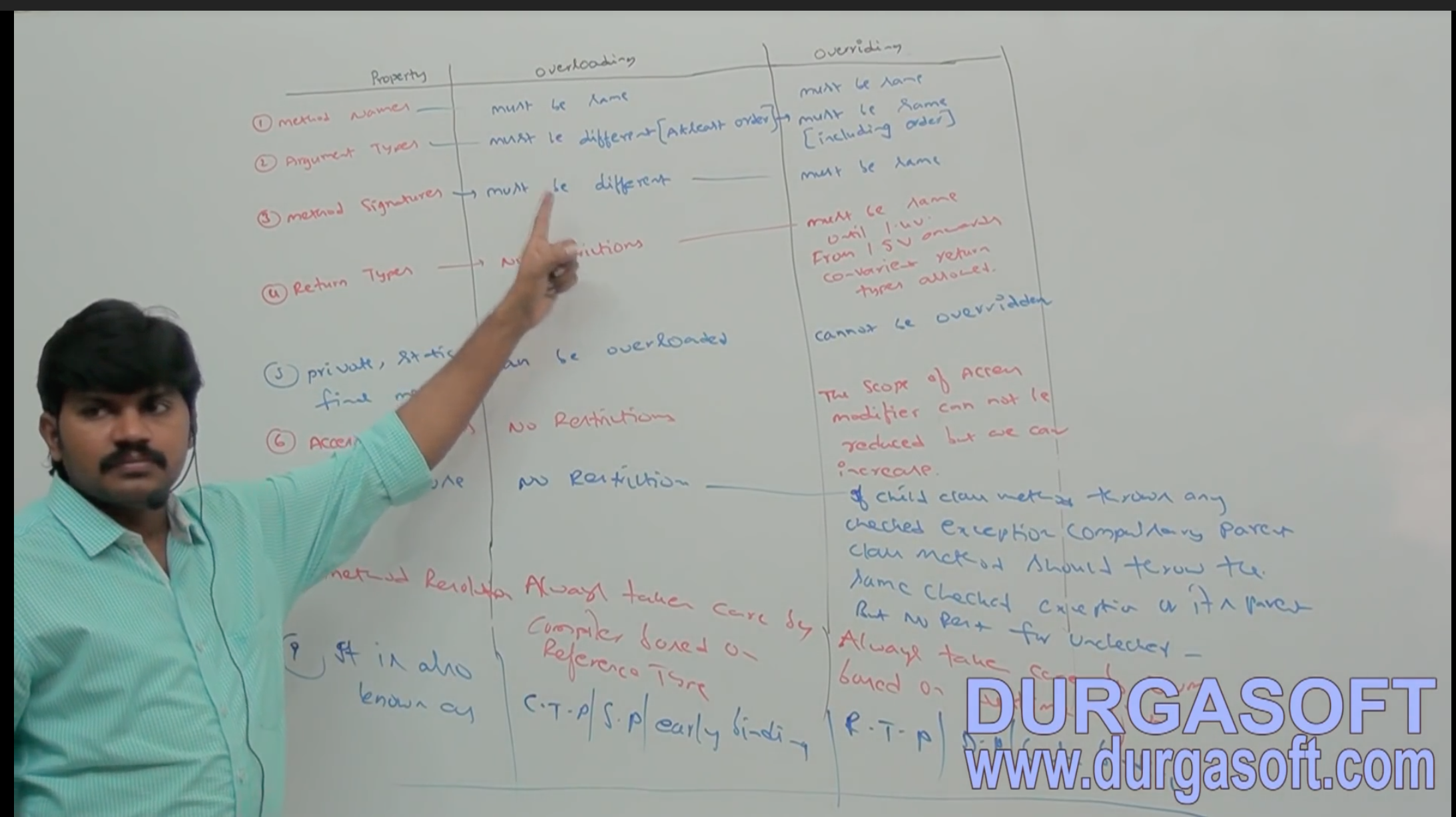
**Overriding w.r.t var-arg methods:**

We can Override var-arg method with another var-arg method only. If we are trying to Override with normal method then it will become Overloading but not Overriding.

**Overriding w.r.t Variables:**

Variable resolution always takes care by Compiler based on reference type, irrespective of whether the variable is static or non-static (Overriding concept applicable only for methods but not for variable).

**Differences between Overloading and Overriding:**

****

**Method with abstract should not be static. It is illegal.**

**Best example of polymorphism:**

A Boy starts LOVE with the word FRIENDSHIP, but GIRL ends with LOVE with the same word FRIENDSHIP. Word is the same but attitude is different. This beautiful concept OOPS is nothing but polymorphism.

1. Static Control Flow
2. Instance Control Flow
3. Constructors
4. **Coupling**

The degree of dependency between the components is called **Coupling.** If dependency is more then it is considered as **Tightly Coupling** and if dependency is less, then it is considered as **Loosely Coupling**.

Class A {

Static int I = B.j;

}

Class B {

Static int J = C.k;

}

Class C {

Static int K = D.m1();

}

Class D {

Public static int m1() {

Return 10;

}

}

The above components are said to be tightly coupled with each other because dependency between the components is more.

Tightly coupling is a not a good programming practice because it has several serious disadvantages.

1. Without effecting remaining components, we can’t modify any component and hence enhancement will become difficult
2. It suppresses reusability
3. It reduces maintainability of the application

Hence we have to maintain dependency between the components as less as possible i.e., **loosely coupling** is a good programming practice.

1. **Cohesion**

For every component a clear well defined functionality is defined then that component is said to be follow **High Cohesion.**

High Cohesion is always a good practice. Because it has several advantages.

1. Without effecting remaining components, we can modify any component hence, enhancement will become easy
2. It promotes reusability of the code (wherever validation is required we can reuse the same Validate Servlet without re-writing)
3. It improves maintainability of the application

**Note**: Loosely Coupling and High Cohesion are good programming practices.

1. **Object Type Casting:**

We can use Parent reference to hold Child Object.

Ex: Object obj = new String(“Phani”);

We can use Interface reference to hold implemented Class Object.

Ex: Runnable r = new Thread();

A b = ©d;

A – Class/Interface Name

b – reference variable name

C – Class/Interface Name

d – reference variable name

**Mantra 1(Compile time checking-1)**: The type of ‘d’ and ‘C’ must have some relation (either child to parent or parent to child or same type) otherwise we will get compile time error **saying inconvertible types found d type required C.**

**Ex:** Object obj = new String(“Phani”);

StringBuffer sb = (StringBuffer)obj; -- Valid

Ex: String s = new String(“Phani”);

StringBuffer sb = (StringBuffer)s; -- will through compile time error because there is no relation between String and StringBuffer

**Mantra 2(Compile time checking-2):** ‘C’ must be either same or derived type of ‘A’ otherwise we will get compile time error **saying incompatible types found ‘C’ required ‘A’**

**Ex:** Object obj = new String(“Phani”);

StringBuffer sb = (StringBuffer)obj; -- Valid

Ex: Object obj = new String(“Phani”);

StringBuffer sb = (String)obj; -- Invalid because, Incompatible types found java.lang.String required java.lang.StringBuffer.

**Mantra 3(Runtime checking):** Runtime object type of ‘D’ must be either same or derived type of ‘C’ otherwise we will get runtime exception saying **“ClassCastException”.**

**Ex:** Object obj = new String(“Phani”);

StringBuffer sb = (StringBuffer)obj; -- Invalid: because ‘obj’ is underline it is ‘String’ but trying to cast to ‘StringBuffer’ because StringBuffer is not child of String.

RE: ClassCastException: java.lang.String cannot be cast to java.lang.StringBuffer

Ex: Object o = new String(“Phani”);

Object o1 = (String)o; -- Valid. It passes 3 mantras mentioned above.

**Ex: Object**

Base1 Base2

Derive1 Derive2 Derive3 Derive4

Base2 b = new Derive4();

1. Object o = (Base2)b; -- Valid
2. Object o = (Base1)b; -- Compile time error
3. Object o = (Derive3)b; -- runtime exception
4. Base2 b1 = (Base1)b; -- Compile time error
5. Base1 b1 = (Derive4)b; -- Compile time error
6. Base1 b1 = (Derive1)b; -- Compile time

Strictly speaking through Type Casting, we are not creating any new object. For the existing Object we are providing another type of reference variable i.e., we are performing **Type Casting** but not **Object Casting.**

Ex 1:

String s = new String(“Phani”);

Object o = (Object)s;

In short we can write as:

Object o = new String(“Phani”);

Ex 2:

Integer I = new Integer(10);

Number n = (Number)I;

Object o = (Object)n;

In short we can write as:

Object o = new Integer(10);

**Note:** C c = new C();

B© --🡪 B b = new C();

A(B©) --🡪 A a = new C();

**Note**: Parent reference can be used to hold child object, but by using that reference we can’t call Child specific methods and we can call only the methods available in Parent class.

**Note:** In Overriding, method resolution is always based on Runtime Object type.

**Note:** When used Static, it is method hiding and method resolution is always based on **Reference Type.**

**Note:** Variable resolution is always based on **Reference Type** but **not based on** **Runtime Object**.

**Static Control flow:**

Whenever we are executing a Java class, the following sequence of steps will be executed as the part of Static Control Flow.

1. Identification of static members from top to bottom (1 to 6)
2. Execution of static variable assignments and static blocks from top to bottom (7 to 12)
3. Execution of main method (13 to 15)

Class Base {

(1) Static int I =10; (7)

(2) Static {

M1(); (8)

s.o.p(“first static block”); (10)

}

(3) p.s.v.m(String args[]) {

m1(); - (13)

s.o.p(“main method”); (15)

}

(4) p.s.v.m1(){

s.o.p(j); (9), (14)

}

(5) static {

s.o.p(“Second static block”); (11)

}

(6) static int j =20; (12)

}

I =0{RIWO(Read Indirectly Write Only)} This will be assigned to ‘0’ by JVM while identifying static members.

j=0{RIWO(Read Indirectly Write Only)} This will be assigned to ‘0’ by JVM while identifying static members.

i=10{R&W(Read And Write)} This is assigned to original value while execution of static variables

j=20{R&W(Read And Write)} This is assigned to original value while execution of static variables

**Read Indirectly Write Only:**

Inside a static block if we are trying to read a variable, that read operation, is called Direct Read.

If we are calling a method and within that method if we are trying to a variable that read operation is called Indirect Read.

If a variable is just identified by JVM and original value not yet assigned, then the variable is said to be in Read Indirectly Write Only state (RIWO).

If a variable is in RIWO state, then we cannot perform direct read but we can perform indirect read.

If we are trying to read directly then we will get compile time error saying “Illegal forward reference”.

**Static Block:**

Static blocks will be executed at the time of Class loading, hence at the time of class loading if we want to perform any activity we have to define that inside static block.

**Ex 1:** At the time of Java class loading the corresponding native libraries should be loaded hence we have to define this activity inside static block.

Static{

System.loadLibrary(“native library path”);

}

Ex: After loading every DB Driver class we have to register driver class with driver manager but inside database driver class there is a static block to perform this activity and we are not responsible to register explicitly.

Note: Within a class we can declare any number of static blocks but all these static blocks will be executed from top to bottom.

Note: From 1.7 version onwards, main method is mandatory to start a program execution. Hence from 1.7 version onwards without writing main method it is impossible to print some statements to the console.

**Static Control Flow in Parent to Child relationship:**

1. Identification of static members from Parent to Child
2. Execution of static variable assignments & static blocks from Parent to Child
3. Execution of only main method of Derived class (If derived class is executed)

**Instance Control flow:**

Whenever we are executing a Java class first static control flow will be executed. In the static control flow if we are creating an Object the following sequence of events will be executed as a part of instance control flow:

1. Identification of Instance members from top to bottom
2. Execution of Instance variable assignments and instance blocks from top to bottom
3. Execution of Constructor

**Note:** Static control flow is one time activity which will be performed at the time of class loading.

But instance control flow is not one time activity and it will be performed for every object creation.

Object creation is the most costly operation if there is no specific requirement then it is not recommended to create object.

**Instance Control Flow in Parent to Child relationship:**

1. Identification of instance variables from Parent to child
2. Execution of instance variable assignments and instance blocks only in Parent class
3. Execution of Parent Constructor
4. Execution of instance variable assignments and instance blocks in child class
5. Execution of Child Constructor

**Note:** From static area we can’t access instance members directly because while executing static area JVM may not identify instance members.

**In how many ways we can create an Object in Java? Or In how many ways we can get Object in Java?**

1. **By using new Operator:**

Test t = new Test();

**2) By using newInstance() method (By Reflection):**

Test t = (Test)Class.forName(“Test”).newInstance();

**3) By Using Factory method:**

Runtime r = Runtime.getRuntime();

DateFormat df = DateFormat.getInstance();

**4) By Using clone() method:**

Test t1 = new Test();

Test t2 = (Test)t1.clone();

**5) By Using Deserialization:**

FileInputStream fis = new FileInputStream(“textFile.txt”);

ObjectInputStream ois = new ObjectInputStream(fis);

Dog d1 = (Dog)ois.readObject();

**Constructors:**

Once we create an Object compulsory we should perform initialization then only the object is in a position to respond properly.

Whenever we are creating an Object some piece of the code will be executed automatically to perform initialization of the object. This piece of the code is nothing but Constructor. Hence, the main purpose of Constructor is initialization of an Object.

**Difference between Constructor and Instance Block:**

The main purpose of Constructor is to perform Initialization of an object.

But other than initialization if we want to perform any activity for every object creation then we should go for instance block (Like updating one entry in the Database for every object creation or incrementing count value for every object creation etc.,).

Both Constructor and Instance block have their own different purposes and replacing one concept with another concept may not work always.

Both Constructor and Instance blocks will be executed for every object creation but Instance block first followed by Constructor next.

**Modifiers allowed for Constructor:**

The only applicable modifiers for constructors are Public, Private, Protected, default. If we are trying to use any other modifier we will get compile time error.

**Default Constructor:**

Compiler is responsible to generate default Constructor (but not JVM).

If we are not writing any constructor then only compiler will generate default constructor i.e., atleast one constructor then compiler won’t generate default constructor. Hence every class in Java can contain constructor it may be default constructor generated by compiler or customized constructor explicitly provided by programmer but not both simultaneously.

**Singleton Class:**

For any java class if we are allowed to create only one object, such type of class is called Singleton Class.

Ex: Runtime, BusinessDelegate, ServiceLocator etc.

**Advantage of Singleton Class:**

If several people have same requirement, then it is not recommended to create a separate object for every requirement.

We have to create only 1 object and we can reuse same object for every similar requirement so that performance and memory utilizations will be improved.

This is the central idea of singleton classes.

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

Object means real world entity such as pen, chair, book etc. Object-Oriented Programming is a methodology to design a program using classes and objects.

Simplifies the software development and maintenance by providing some concepts:

* Object
* Class
* Inheritance
* Polymorphism
* Abstraction
* Encapsulation

**Object:**

Any entity that has state and behavior is known as object. Ex: Chair, pen etc. It can be physical and logical.

**Class:**

Collection of object is called class. It is a logical entity.

**Abstraction:**

The purpose of abstraction is to hide the information that is not relevant or show only relevant data.

Abstraction means “The process of forming of general and relevant concept from more complex scenarios”.

**Encapsulation:**

Encapsulation means the localization of the information or knowledge within an object.

Encapsulation means “Information hiding”.

How to bring “Encapsulation”

1. Make the instance variable protected.
2. Create a public accessory method and use these methods from within the calling code.
3. Use the JavaBeans naming convention of getter or setter

Eg: getPropertyName, setPropertyName



Take away the above example:

The application using an Object of this class EmployeeCount will not able to get the NoOfEmployees directly.

Setting and getting the value of the field NoOfEmployees is done with the help of Getter and Settter method shown as above.

**Inheritance:**

When one object acquires all the properties and behavior from parent object i.e., known as inheritance. It provides code reusability. It is used to achieve runtime polymorphism.

* Inheritance is a mechanism of defining a new class based on an existing class.
* Inheritance enables reuse of the code. Inheritance also provides the scope for refinement of the existing class. Inhertiance helps in specialization.
* The existing (or original) class is called base class or super class or parent class. The new class which inherits from the base class is called the derived class or child class or sub class.
* Inheritance implements the “Is-A” or “Kind Of/Has-A” relationship.

Note: The biggest advantage of the inheritance is that, code in base class need not to be rewritten in the derived class.

The member variables and methods of the base class can be used in the derived class as well.

1. A refernece variable of super class can be assigned to a refernece to any sub class derived from that super class.
2. Multi-level inheritance is allowed in Java but not multiple inheritance
3. Although a sub class includes all of the members of its super class it can not access those members of the super class that have been declared as private.

**Polymorphism:**

When one task is performed by different ways i.e., known as polymorphism.

In Java, we use method overloading and method overriding to achieve polymorphism.

Ex: Can be speak to something e.g cat speaks meaw, dog barks woof etc.

Polymorphism is a feature that allows one interface to be used for a general class of actions. Its an operation exhibits different behaviour in different instances. The behaviour depends on the types of data used in the operation.

It plays an important role in allowing objects having different internal structures to share the same exeternal interface.

Polymorphism is extensively used in implementing Inheritance.

Types of Polymorphism:

* Static Polymorphism
* Dynamic Polymorphism

Static Polymorphism:

Function Overloading:

* within same class more than one method having same name but different signature.
* Resolved during compilation time.
* Return type is not part of method signature.

Dynamic Polymorphism:

Function Overriding:

* Keeping the return type and method signature same, method in base class is redefined in derived class.
* Resolved during run time.
* Which method to be invoked is decided by the object that refernece points to and not by the type of reference.

Overriding:

* Redefining a super class method in a sub class is called method overriding.
* The method signature i.e., method name, parameter list and return type have to match exactly.
* The overriden method can be widely the accessibility but not narrow it i.e., if it is protected in the base class, the child class can make it public but nt vice versa.

Method/Function Overloading:

The concept of the same function name with different types of parameters being passed is called Function Overloading.

1. In Overloading we can reuse the same method name by changing the arguments.
2. Overload methods – Must and Must Not Facts

* The overload method must have different argument lists.
* Can have different return types but in that case it is manditory to have different argument list.
* Can have different access modifiers and
* Can throw different exceptions.

1. Methods can be overloaded in the same as well as the sub classes.

Q: What determines which overridden method is used at runtime?

A: Object type

Q: What determines which overloaded method will be used at compile time?

A: Reference type determines. \_Operator overloading refers to the operators like ‘+’ being used for different purposes based on the data type on either side of the operator.

**HAS-A** relationships are based on usage, rather than inheritance. In other words, class A **HAS-A** B if code in class A has a reference to an instance of class B.

A Car **IS-A** Vehicle. A Car **HAS-A** License. and the code looks like this:

public class Vehicle{ }

public class Car extends Vehicle{

private License myCarLicense;

}

**Interface:**

1. Java doesn’t support Multiple Inhertiance.
2. Interface is similar to an abstract class that contains only abstract methods.
3. Declared with keyword interface instead of keyword class.
4. Keyword implements is used to represent the interface implemented by the class.

* All methods in an interface are implicitly public and abstract.
* The keyword abstract before each method is optional.
* An interface may contain defination of final variables.
* A class may extend only one other class, but implements any number of interfaces.
* When a class implements an interface, it may implement (define) some or all of the inherited abstract methods.
* A class must itself declared as abstract, if it inherits abstract methods that if it does not define.
* An interface reference can point to objects of its implementing classes.

**Abstract method:**

* A method that is declared but not defined.
* Declared with a keyword abstract.
* Has a header like any other method, but ends with semicolumn instead of method body.
* Used to put some kind of compulsion on the class who inherits for this class i.e., the class who inherits MUST provide the implementation of the method else the subclass will also become abstract.
* The following cannot be marked as abstract modifier:
  + Constructors
  + Static methods
  + Private methods
  + Methods marked with “final” modifier.

**Abstract Classes:**

Outlines the behaviour but not necesarrily implements all of its behaviour.

Provides outline for behaviour by means of method(abstract methods) signatures without an implementation.

**Note 1:** There can be some scenarios where it is difficult to implement all the methods in the base class. In such scenarios one can define the base class as an abstract class which signifies that this base class is a special kind of class which is not complete on its own.

A class derived from the abstract base class must implements those member functions which are not implemented in the abstract class.

**Note 2:** Abstract class cannot be instantatied.

To use an abstract class one has to first derive another class from this abstarct class using inheritance and then provide the implementation for the abstract methods.

**Note 3:** If a derived class doesn’t implement all the abstarct methods, then the derived class is also abstarct in nature and cannot be instantiated.

**What is a Constructor?**

* A method with the same name as class name used for the purpose of creating an object in a valid state.
* It doesn’t return a value not even void.
* It may or may not have parameters(arguments).
* A class may contain one or more constructors for making a new object of the class.
* If and only if programmer doesn’t write a constructor, Java provides a default constructor with no arguments.

The default constructor sets instance variables as:

* Numeric types are set to zero.
* Boolean variables are set to false.
* Char variables are set to ‘’.
* Object variables are set to null.

**When a constructor executes, before executing its own code:**

It implicitly call the default constructor of its super class Or can make this constructor call explicitly, with super(…);

A constructor for a class can call another constructor for the same class by putting this(….); as the first thing in the constructor. This allows you to avoid repeating code.

**Exception handling in Method Overriding:**

[**http://beginnersbook.com/2014/01/exception-handling-in-method-overriding-with-example/**](http://beginnersbook.com/2014/01/exception-handling-in-method-overriding-with-example/)

**Rule:** An overriding method can throw any unchecked exceptions, regardless of whether the overriden method throws or not.

However overriding method should not throw checked exceptions that are new or broader than the ones declared by overriden method.

The overriding method can throw those checked exceptions, which have less scope than the exceptions declared in the overriden method.

**Java – Constructor in Interface?**

[**http://beginnersbook.com/2013/12/java-constructor-in-interface/**](http://beginnersbook.com/2013/12/java-constructor-in-interface/)

No, interface cannot have constructors.

**Java - Constructor Chaining with Example:**

[**http://beginnersbook.com/2013/12/java-constructor-chaining-with-example/**](http://beginnersbook.com/2013/12/java-constructor-chaining-with-example/)

Calling a constructor from another constructor of same class is known as Constructor chaining.

**Example Program:**

In the below example the class “ChainingDemo” has 4 constructors and we are calling one constructor from another using **this() statement.**For e.g. in order to call a constructor with single string argument we have supplied a string in this() statement like **this(“hello”).**

**Note**: this() should always be the first statement in **constructor** otherwise you will get the error message.

|  |
| --- |
| package beginnersbook.com;  public class ChainingDemo {  //default constructor of the class  public ChainingDemo(){  System.out.println("Default constructor");  }  public ChainingDemo(String str){  this();  System.out.println("Parametrized constructor with single param");  }  public ChainingDemo(String str, int num){  //It will call the constructor with String argument  this("Hello");  System.out.println("Parametrized constructor with double args");  }  public ChainingDemo(int num1, int num2, int num3){  // It will call the constructor with (String, integer) arguments  this("Hello", 2);  System.out.println("Parametrized constructor with three args");  }  public static void main(String args[]){  //Creating an object using Constructor with 3 int arguments  ChainingDemo obj = new ChainingDemo(5,5,15);  }  } |

Output:

|  |
| --- |
| Default constructor  Parametrized constructor with single param  Parametrized constructor with double args  Parametrized constructor with three args |

**Static and Dynamic binding in Java:**

Association of method definition to method call is known as binding.

**Static Binding or Early Binding:**

The binding which can be resolved at compile time by compiler is known as Static or Early Binding.

All static, private, final methods have always been bonded at compile-time.

Compiler knows that all such methods cannot be overridden and will always be accessed by object of local class.

Hence compiler doesn’t have any difficulty to determine object of class.

**Dynamic or Late Binding:**

When compiler is not able to resolve the call/binding at compile time, such binding is known as Dynamic or Late Binding.

Overriding is a perfect example of dynamic binding as in overriding both parent and child classes having same method. Thus while calling the overridden method, the compiler gets confused between parent class and child class method(since both the methods have same name).

|  |  |
| --- | --- |
| Static Binding | Dynamic Binding |
| It happens at compile time | It happens at run time |
| Binding of private, static and final methods always happen at compile time since these methods cannot be overridden | Binding of overridden methods happen at run time |
| Java uses static binding for overload methods | Dynamic binding for overridden methods |

**Static:**

[**http://beginnersbook.com/2013/04/java-static-class-block-methods-variables/**](http://beginnersbook.com/2013/04/java-static-class-block-methods-variables/)

1. Static class
2. Static block
3. Static methods
4. Static variables

**Static Class:**

A Class can be made static only if it is a nested Class. The nested static Class can be accessed without having object of outer class.

Compile Time error:

class Example2{

int num;

//Static class

static class X{

static String str="Inside Class X";

num=99;

}

public static void main(String args[])

{

Example2.X obj = new Example2.X();

System.out.println("Value of num="+obj.str);

}

}

Output:

Compile time error. Static Inner class cannot access instance data of outer class.

**Static Block:**

Static block is used for changing the default values of static variables. This block gets executed when the class is loaded in the memory.

A class can have multiple Static blocks, which will execute in the same sequence in which they have written into the program.

**Static Methods:**

Static methods can access static variables without using object of the class. It can access non-static methods and non-static variables by using objects. Static methods can be accessed directly by static and non-static methods.

**Static Variable:**

* Static variables are also known as class variables.
* Such variables get default values based on the data type.
* Data stored in static variables is common for all the objects of that Class.
* Memory allocation for such variables only happens once when class is loaded in the memory.
* These variables can be accessed in any other class using class name.
* Unlike non-static variables, such variables can be accessed directly in static and non-static methods.

**Inner Class:**

<http://beginnersbook.com/2013/05/inner-class/>

What is Inner class?

Inner classes are defined inside the body of another class (known as outer class). These classes can have access modifier or even can be marked as abstract or final. Inner class has special relationship with outer class instances. This relationship allows them to have access to outer class members including private members too.

Inner classes can be defined in four different following ways as mentioned below:

* Inner class
* Method – local inner class
* Anonymous inner class
* Static nested class

1. Inner class

An inner class is declared inside the curly braces of another enclosing class. Inner class is coded inside a top-level class.

Inner class acts as a member of the enclosing class and can have any access modifiers: abstract, final, public, protected, private, static.

Inner class can access all members of the outer class including those marked private.

**Initiating an inner class:**

To instantiate the inner class, there should be a live instance of outer class. An inner class instance can be created only from an outer class instance.

class MyOuterClassDemo {

private int x= 1;

public void innerInstance()

{

MyInnerClassDemo inner = new MyInnerClassDemo();

inner. seeOuter();

}

public static void main(String args[]){

MyOuterClassDemo obj = new MyOuterClassDemo();

obj.innerInstance();

}

// inner class definition

class MyInnerClassDemo {

public void seeOuter () {

System.out.println("Outer Value of x is :" + x);

}

} // close inner class definition

} // close Top level class definition

**Instantiating an inner class from outside the outer class Instance Code:**

The public static void main code in the above example can be replaced with this one. It will also give the same output.

public static void main(String args[]){

MyOuterClassDemo.MyInnerClassDemo inner = new MyOuterClassDemo().new MyInnerClassDemo();

inner. seeOuter();

}

1. Method-Local inner classes:

A method-local inner class is defined within a method of the enclosing class. If you want to use inner class, you must instantiate the inner class in the same method, but after the class definition code. Only 2 modifiers are allowed for method-local inner class, which are abstract and final. The inner class can use the local variables of the method, only if they are marked final.

1. Anonymous Inner Classes:

It is a type of inner class which:

* Has no name
* Can be instantiated only once
* Is usually declared inside a method or a code block, a curly braces ending with semi column.
* Is accessible only at the point where it is defined.
* Does not have constructor simply because it does not have a name
* Cannot be static.

class Pizza{

public void eat()

{

System.out.println("pizza");

}

}

class Food {

/\* There is no semicolon(;)

\* semicolon is present at the curly braces of the method end.

\*/

Pizza p = new Pizza(){

public void eat()

{

System.out.println("anonymous pizza");

}

};

}

1. Static Nested Classes:

Static nested classes are the inner classes marked with static modifier. Because this is static in nature so this type of inner class doesn’t share any kind of relationship with an instance of outer class. A static nested class cannot access non-static members of outer class.

**Abstract Class:**

A class that is declared using “abstract” keyword is known as abstract class. It may or may not include abstract methods which means in abstract class you can have concrete methods (methods with body) as well along with abstract methods (without an implementation, without braces, and followed by a semicolon). An abstract class can not be **instantiated** (you are not allowed to create **object** of Abstract class).

Remember 2 rules:

1. If the class is having few abstract methods and few concrete methods: declare it as abstract class.
2. If the class is having only abstract methods; declare it as interface.

Key Points:

* An abstract class has no use until unless some other class extends it.
* If you declare an abstract method in a class then you must declare the class abstract as well. You cannot have a abstract method in non-abstract class. Its vice versa is not always true: If a class is not having any abstract method then also it can be marked as abstract.
* Abstract class can have non-abstract method (concrete) as well.

|  |  |
| --- | --- |
| **Abstract Classes** | **Interfaces** |
| abstract class can extend only one class or one abstract class at a time | interface can extend any number of interfaces at a time |
| abstract  class  can extend from a class or from an abstract class | interface can extend only from an interface |
| Abstract class has both abstract methods and concrete methods | Interface has only abstract methods |
| A class can extend only one abstract class | A class can implement more than one interface |
| In abstract class keyword ‘abstract’ is mandatory to declare a method as an abstract | It is optional |
| It can have public, protected and public abstract methods | Interface can have only public abstract methods by default. |
| Abstract classes can have static, final or static final variable with any access specifier. | Interface can have only static final (constant) variable i.e., by default. |

**Final Keyword:**

1. **Final variable:**

Final variables are nothing but constants. We cannot change the value of a final variable once it is initialized.

**Blank final variable:**

A final variable that is not initialized at the time of declaration is known as Blank final variable. We must initialize the blank final variable in constructor of the class otherwise it will throw compilation error.

**Uninitialized static final variable:**

A static final variable that is not initialized during declaration can only be initialized in static block.

1. **Final method:**

A final method cannot be overridden. Which means even though a sub class can call the final method of parent class without any issues but it cannot override it.

1. **Final class:**

We cannot extend final class.

Points to remember:

* A constructor can be declared as final.
* Local final variable must be initializing during declaration.
* All variables declared in interface are final.
* We cannot change the value of final variable.
* A final method cannot be overridden.
* A final class cannot be inherited.
* If method parameters are declared as final then the values of these parameters cannot be changed.
* It is a good practice to name final variable in all CAPS.
* Final, finally and finalize are 3 different terms. Finally is used in exception handling and finalize is a method that is called by JVM during garbage collection.

**What is Association?**

Association is a relationship where all object has their own lifecycle and there is no owner. Let's take an example of Teacher and Student. Multiple students can associate with single teacher and single student can associate with multiple teachers but there is no ownership between the objects and both have their own lifecycle. Both can create and delete independently.

**What is Aggregation?**

Aggregation is a specialize form of Association where all object has their own lifecycle but there is ownership and child object can not belong to another parent object. Let's take an example of Department and teacher. A single teacher can not belong to multiple departments, but if we delete the department teacher object will not destroy. We can think about "has-a" relationship.

**What is Composition?**

Composition is again specializing form of Aggregation and we can call this as a "death" relationship. It is a strong type of Aggregation. Child object dose not have their lifecycle and if parent object deletes all child object will also be deleted. Let's take again an example of relationship between House and rooms. House can contain multiple rooms there is no independent life of room and any room can not belong to two different house if we delete the house room will automatically delete.

**Interview Questions:**