# **Coupling and Cohesion**

(A program should have – Loose coupling and High Cohesion)

**Coupling**

In order to create an effective application that is easy to develop, easy to maintain and easy to update, we need to know the concept of **coupling and cohesion**. **Coupling** refers to the extent to which a class knows about the other class. 

There are two types of coupling -

* **Tight Coupling**(*bad programming design*)
* **Loose Coupling**(*good programming design*)
* **Tight Coupling**

If a class A is able to access data members/instance variables of a class B **directly by dot operator**(because they were declared **public**), the two classes are said to be **tightly coupled** and it leads to *bad designing*, because all the checks made to ensure the valid access to data members of class B are bypassed by their direct access. Let's understand this by an example -

***//Tight coupling***

**class Names**

**{**

**public String name;**

**public String getName()**

**{**

***//some code to check valid access to name***

**return name;**

**}**

**public void setName(String s)**

**{**

***//some code to check valid setting to name***

**name=s;**

**}**

**}**

**class ModifyData**

**{**

**public void updateName()**

**{**

**Name ob= new Name();**

**ob.name="Hello"; *//Directly accessing name with dot operator shows tight coupling between two classes***

**System.out.println(ob.name); *//Tight coupling because of bad encapsulation***

**}**

**}**

* **Loose Coupling**

A good application designing is creating an application with loosely coupled classes by following proper **encapsulation**, i.e. by declaring data members of a class with **private** access and forcing other classes to access them only through **public getter, setter methods**. Let's understand this by an example -

**class Names**

**{**

**private String name;**

**public String getName()**

**{**

***//some code to check valid access to name***

**return name;**

**}**

**public void setName(String s)**

**{**

***//some code to check valid setting to name***

**name=s;**

**}**

**}**

**class ModifyData**

**{**

**public void updateName()**

**{**

**C ob= new C();**

**ob.setName("Howard");**

**ob.getName();**

**}**

**}**

**Cohesion**

**Cohesion** refers to the extent to which a class is defined to do a **specific specialized task**. A class created with high cohesion is targeted towards a single specific purpose, rather than performing many different specific purposes.

There are two types of cohesion -

* + **Low cohesion**(*bad programming design*)
  + **High Cohesion**(*good programming design*)

* **Low Cohesion**

When a class is designed to do many different tasks rather than focussing on a **single specialized task**, this class is said to be a "*low cohesive*" class. Low cohesive class are said to be *badly designed* leading to a hard time at creating, maintaining and updating them. Let's understand this by an example -

**class PlayerDatabase**

**{**

**public void connectDatabase();**

**public void printAllPlayersInfo();**

**public void printSinglePlayerInfo();**

**public void printRankings()**

**public void closeDatabase();**

**}**

* **High Cohesion**

A good application design is creating an application with *high cohesive* classes, which are targeted towards a *specific specialized task* and such class is easy not only easy to create, but also easy to maintain and update.

**class PlayerDatabase**

**{**

**ConnectDatabase connectD= new connectDatabase();**

**PrintAllPlayersInfo allPlayer= new PrintAllPlayersInfo();**

**PrintRankings rankings = new PrintRankings();**

**CloseDatabase closeD= new CloseDatabase();**

**PrintSinglePlayerInfo singlePlayer = PrintSinglePlayerInfo();**

**}**

**class ConnectDatabase**

**{**

***//connecting to database.***

**}**

**class CloseDatabase**

**{**

***//closing the database connection.***

**}**

**class PrintRankings**

**{**

***//printing the players current rankings.***

**}**

**class PrintAllPlayersInfo**

**{**

***//printing all the players information.***

**}**

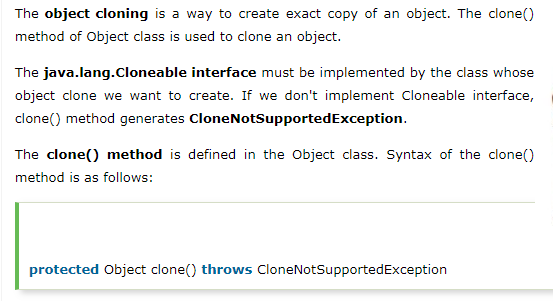
**class PrintSinglePlayerInfo**

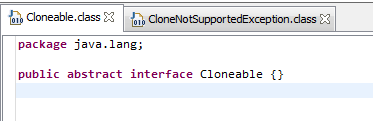
**{**

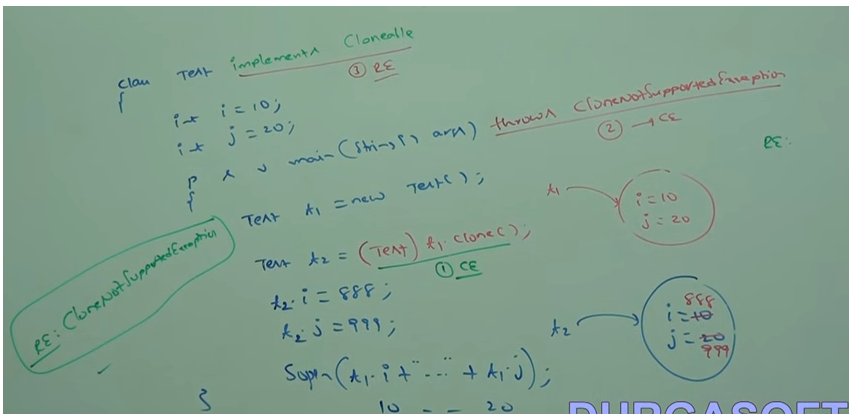
***//printing a single player information.***

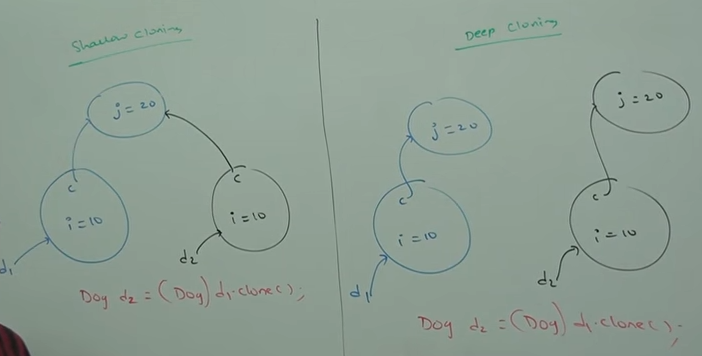
**}**

**Object Cloning:**

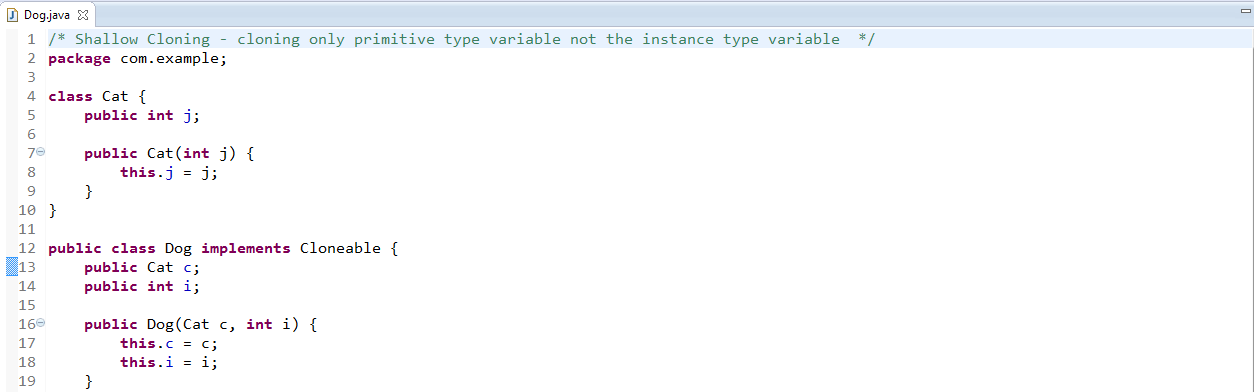


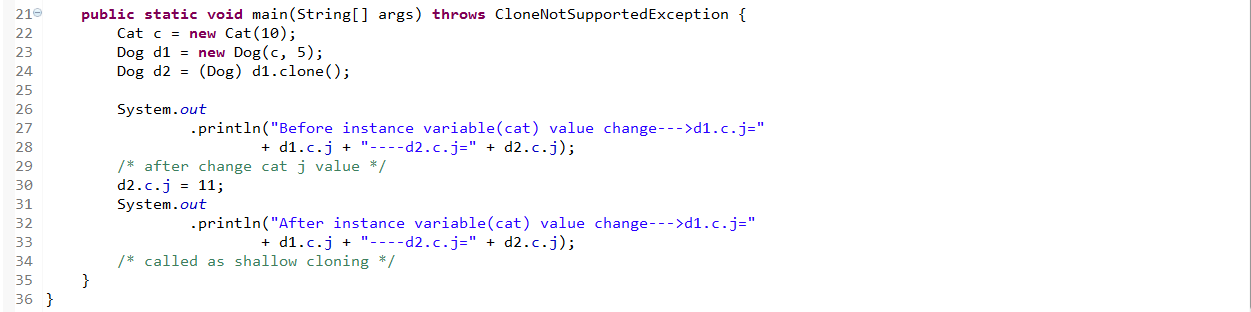


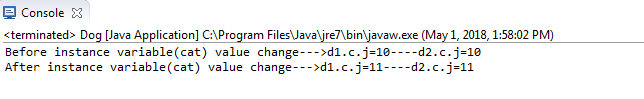


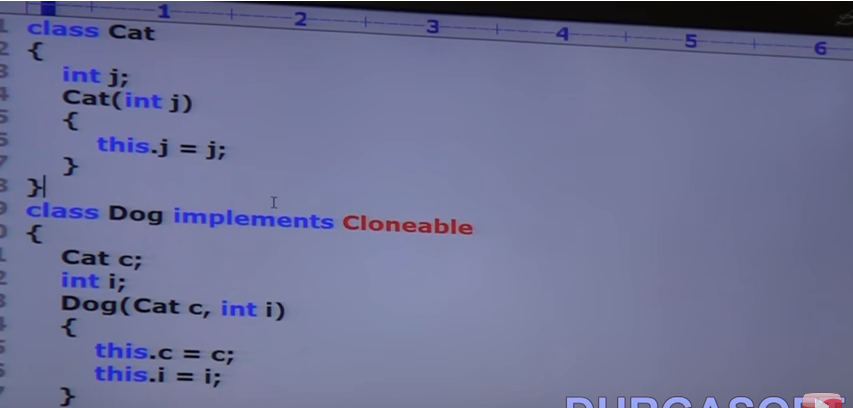


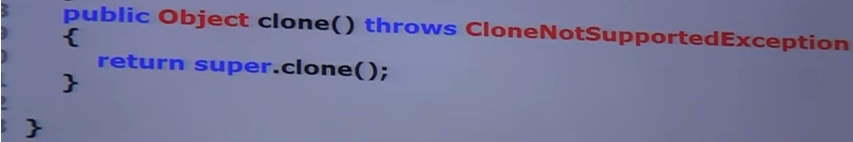
Note : Shallow cloning uses Object class clone() method.

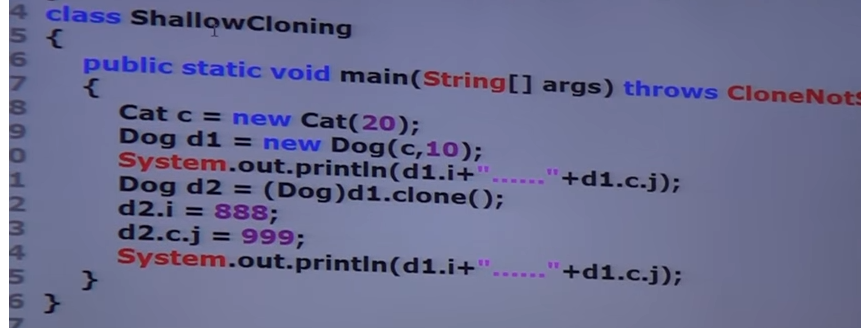


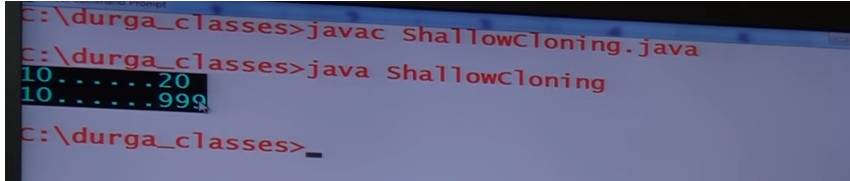


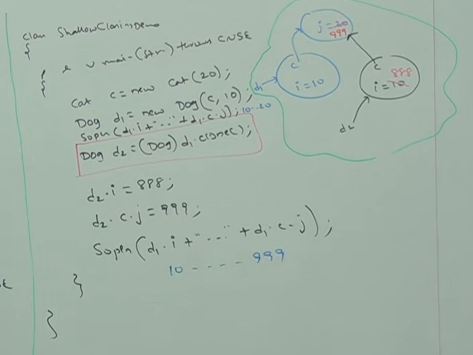












**In Shallow cloning** – By using cloned object reference if we perform any change to the content object then those changes will be reflected to the main object.

To overcome this problem we should go for “Deep Cloning”.

**Deep Cloning**: The process of creating exactly duplicate independent copy including “contend object” is called deep cloning.

In deep cloning if the main object contend any primitive variables then in the cloned object duplicate copy will be created.

If the main object contend any reference variable then corresponding contend object will be created in the cloned copy.

By default object class clone() method mint for shallow cloning, but we can implement deep cloning explicitly by overriding clone() method in over class.

