**\*Assigning One Object to Another**

**Shallow copy:** You can create a new object and copy the reference of the original object to the new object. This is known as a shallow copy. However, this does not create a new copy of the object, it just creates a new reference to the same object. Any changes made to the object through the new reference will be reflected in the original object as well. Here's an example of how you can do a shallow copy:

class S

{

int i=10;

}

class Hi {

public static void main(String[] args) {

System.out.println("Hello, World!");

**S s1= new S();**

System.out.println("s1.i initial value is :" + s1.i);

**S s2;**

**s2=s1;**

System.out.println("s2.i initial value is :" + s2.i);

s1.i++;

System.out.println("s1.i updated value is :" + s1.i);

System.out.println("s2.i updated value is :" + s2.i);

}

}

**Deep copy**: To create a new copy of an object and all its fields, you can use deep copying. One way to do this is to use the clone method of the Object class. However, this method is protected, so you need to override it in your class and make it public.

1. The class whose object’s copy is to be made must have a public clone method in it or in one of its parent classes.
2. Every class that implements clone() should call super.clone() to obtain the cloned object reference.
3. The class must also implement java.lang.Cloneable interface whose object clone we want to create otherwise it will throw CloneNotSupportedException when the clone method is called on that class’s object.

Here's an example of how you can do a deep copy using the clone method:

class MyClass **implements Cloneable {**

int value;

public MyClass(int value) {

this.value = value;

}

public MyClass **clone() throws CloneNotSupportedException**

{

return **(MyClass) super.clone();**

}

}

public class Main {

public static void main(String[] args) {

MyClass original = new MyClass(1);

MyClass copy;

try {

copy = original.clone();

copy.value = 2;

System.out.println(original.value); // Outputs 1

System.out.println(copy.value); // Outputs 2

} catch (CloneNotSupportedException e) {

e.printStackTrace(); } } }

\* **Keyword this :**

 this keyword refers to the current object in a method or constructor.

The most common use of , this keyword is to eliminate the confusion between class attributes and parameters with the same name (because a class attribute is shadowed by a method or constructor parameter).

**With out using this keyword :**

//If you omit the keyword in the example above, the output would be "0" instead of "9".

public class Main {

int x;

// Constructor with a parameter

public Main(int x) {

**x = x;**

}

// Call the constructor

public static void main(String[] args) {

Main myObj = new Main(9);

System.out.println("Value of x = " + myObj.x); // output : **Value of x =0;**

}

}

**With using this keyword :**

public class Main {

int x;

// Constructor with a parameter

public Main(int x) {

**this. x = x;**

}

// Call the constructor

public static void main(String[] args) {

Main myObj = new Main(9);

System.out.println("Value of x = " + myObj.x); // output : **Value of x =9;**

}

}

**this can also be used to:**

**Invoke current class constructor**

**Calling default constructor from parameterized constructor:**

1. **class** A{
2. A(){System.out.println("hello a");}
3. A(**int** x){
4. **this**();
5. System.out.println(x);
6. }
7. }
8. **class** TestThis5{
9. **public** **static** **void** main(String args[]){
10. A a=**new** A(10);
11. }}

**Invoke current class method**

1. **class** A{
2. **void** m(){System.out.println("hello m");}
3. **void** n(){
4. System.out.println("hello n");
5. //m();//same as this.m()
6. **this**.m();
7. }
8. }
9. **class** TestThis{
10. **public** **static** **void** main(String args[]){
11. A a=**new** A();
12. a.n();
13. }}

**Return the current class object**

1. **class** A{
2. A getA(){
3. **return** **this**;
4. }
5. **void** msg(){System.out.println("Hello java");}
6. }
7. **class** Test1{
8. **public** **static** **void** main(String args[]){
9. **new** A().getA().msg();
10. }
11. }

**Pass an argument in the method call**

1. **class** S2{
2. **void** m(S2 obj){
3. System.out.println("method is invoked");
4. }
5. **void** p(){
6. m(**this**);
7. }
8. **public** **static** **void** main(String args[]){
9. S2 s1 = **new** S2();
10. s1.p();    } }

**Pass an argument in the constructor call**

1. **class** B{
2. A4 obj;
3. B(A4 obj){
4. **this**.obj=obj;
5. }
6. **void** display(){
7. System.out.println(obj.data);//using data member of A4 class
8. }
9. }
11. **class** A4{
12. **int** data=10;
13. A4(){
14. B b=**new** B(**this**);
15. b.display();
16. }
17. **public** **static** **void** main(String args[]){
18. A4 a=**new** A4();
19. }
20. }

**\*Class Objects as Parameters in Methods :**

class AsArgument

{

int age;

String name;

AsArgument(int a,String n)

{

age=a;

name=n;

}

void display(**AsArgument aa**)

{

System.out.println("age is :"+ aa.age);

System.out.println("name is :"+ aa.name);

}

}

class Main

{

public static void main(String ar[])

{

AsArgument obj = new AsArgument(16,"ABC");

obj.display**(obj);**

}

}

**A method can return an object too :**

class AsArgument

{

int age;

String name;

AsArgument(int a,String n)

{

age=a;

name=n;

}

AsArgument retObj(int a,String s)

{

return new AsArgument(a,s);

}

}

class Main

{

public static void main(String ar[])

{

AsArgument obj = new AsArgument(16,"ABC");

AsArgument obj2=obj.retObj(20,"DEF");

System.out.println("age is :"+ obj2.age);

System.out.println("name is :"+ obj2.name);

}

}

**Passing Arguments by Value and by Reference**

**Call by Value in Java**

In Java, call by value refers to calling a method by passing the value in the parameter. In Java, call by value passes a copy of the variable to the method, so all changes are reflected only in that method; thus, no changes are reflected in the main method. This is the main difference between call by value in java and call by reference in java. When we pass a variable with primitive data types, it is considered a call by values in Java, so any changes to the variable will not be reflected in the caller’s scope.

Example of Call by Value in Java

**class** CallByVal{

**public** **static** **void** increment(**int** number){

number = number+1; // increment variable by 1

System.out.println("value in method: "+number);

}

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

**int** number=10;

System.out.println("value before method call : "+number);

increment(number);

System.out.println("value after method call: "+number);

}

}

**Call by Reference in Java**

Though Java is strictly call by value when we pass the reference of the object it creates a copy of the reference and then passes it as value to the method. The copy reference also points to the same address so all the changes also reflect in the main method this is the main difference between call by value in java and call by reference in java. Let’s see examples of how we can achieve call by reference in java.

Example of Call by Reference in Java using Object

**class** CallByRef{

**int** number=10;

// pass object as parameter

**public** **static** **void** increment(CallByRef c){

c.number = c.number+1; // increment variable by 1

System.out.println("value in method: "+c.number);

}

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

CallByRef c=new CallByRef(); // c is an object of class CallByRef

System.out.println("value before method call: "+c.number);

increment(c); // pass object of the class CallByRef

System.out.println("value after method call: "+c.number);

}

}