OOPS II - Access Modifiers & Constructors

In this tutorial, we will learn about

- Access Modifiers
- Getters & Setters
- Constructors
- Shallow & Deep Copy
- Java Memory Model - Objects & References
- Life & Death on Objects on Heap
- Project : Guess Game using OOPS Concepts

Access Modifiers

The access modifiers in Java specifies the accessibility or scope of a field, method, constructor, or class. We can change the access level of fields, constructors, methods, and class by applying the access modifier on it. Let's understand it through an example.

```
public class Player {
    //Data Members
    String name;
    private int guess;
    public String handle;

    //Example of private method
    // can't be called from outside the class
    private void assignTeam(){
        ...
    }
    //example of a public method
    //can be called from anywhere
    public void setTeam(int teamId){
        ...
    }
}
```

In the above class, `guess` is private, `name` is default, `handle` is public. What does't it mean? Let understand the meanings of above acess modifiers.

There are four types of access modifiers in Java:

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

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

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

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

it will be the default.

Modifier	Class	Package	Subclass	Global
Public	Yes	Yes	Yes	Yes
Protected	Yes	Yes	Yes	No
Default	Yes	Yes	No	No
Private	Yes	No	No	No

Getters & Setters

If you try to read or write a private data member, outside the class, you will get a compile error. In order to work with private data members, you might need to create special public methods called `getters()` and `setters()` in the class for specific data members as shown below.

```
public class Player {
    //Data Members
    String name;
    private int guess;
    public String handle;

    //Setter Method
    public int setGuess(int guess){
        //Setters can have their validation logic before updating class member
        if(guess>=0){
            this.guess = guess;
        }
    }
    // Getter Method
    public int getGuess(){
        return this.guess;
    }
}
```

The advantage of this approach is you can set the value if it satisfies the class specific validation logic.

Constructors

A constructor is a special method that is called when an object is created. It is used to initialize the object. It is called automatically when the object is created. It can be used to set initial values for object attributes.

Constructors are the gatekeepers of object-oriented design. Let us create a class for students:

Student.java

```
public class Student {

    private String name;
    private String email;
    private Integer age;
    private String address;
    private String batchName;
    private Integer psp;

    public void changeBatch(String batchName) {
        ...
    }
}
```

The above class can be used to create objects of type Student. This is done by using the new keyword:

```
Student student = new Student();
student.name = "Eklavya";
```

You can notice that we did not define a constructor for the Student class. This brings us to our first type of constructor

Default constructor

A default constructor is a constructor created by the compiler if we do not define any constructor(s) for a class.

A default constructor is a constructor that either has no parameters, or if it has parameters, all the parameters have default values. If no user-defined constructor exists for a class and one is needed, the compiler implicitly declares a default parameterless constructor.

A default constructor is also known as a no-argument constructor or a nullary constructor. All fields are left at their initial value of 0 (integer types), 0.0 (floating-point types), false (boolean type), or null (reference types) An example of a no-argument constructor is:

```
public class Student {
    private String name;
    private String email;
    private Integer age;
    private String address;
    private String batchName;
    private Integer psp;

    public Student() {
        // no-argument constructor
    }
}
```

Notice a few things about the constructor which we just wrote. First, it's a method, but it has no return type. That's because a constructor implicitly returns the type of the object that it creates. Calling new Student() now will call the constructor above.

Secondly, it takes no arguments. This particular kind of constructor is called a no-argument constructor.

Syntax of a constructor In Java, every class must have a constructor. Its structure looks similar to a method, but it has different purposes. A constructor has the following format `<Constructor Modifiers> <Constructor Declarator> <Constructor Body>`

Constructor declarations begin with access modifiers: They can be public, private, protected, or package access, based on other access modifiers. Unlike methods, a constructor can't be abstract, static, final, native, or synchronized.

The declarator is the name of the class, followed by a parameter list. The parameter list is a comma-separated list of parameters enclosed in parentheses. The body is a block of code that defines the constructor's behavior.

`Constructor Name (Parameter List)`

Parameterised Constructor

Now, a real benefit of constructors is that they help us maintain encapsulation when injecting state into the object. The constructor above is a no-argument constructor and hence value have to be set after the instance is created.

```
Student student = new Student();
student.name = "prateek";
student.email = "prateek@gmail.in";
...
```

The above approach works but requires setting the values of all the fields after the instance is created. Also, we won't be able to validate or sanitize the values. We can add the validation and sanitization logic in the getters and setters but we wont be able to fail instance creation. Hence, we need to add a parameterised constructor. A parameterised constructor has the same syntax as the constructors before, the onl change is that it has a parameter list.

```
public class Student {
    private String name;
    private String email;

    public Student(String name, String email) {
        this.name = name;
        this.email = email;
    }
}
```

```
Student s1 = new Student("prateek", "prateek@gmail.in");
Student s2 = new Student("rahul", "Rahul@gmail.in");
```

In Java, constructors differ from other methods in that:

- Constructors never have an explicit return type.
- Constructors cannot be directly invoked (the keyword “new” invokes them).
- Constructors should not have non-access modifiers.

Copy constructor

A copy constructor is a member function that initializes an object using another object of the same class. A copy constructor has the following general function prototype:

```
class Student {
    private String name;
    private String email;

    public Student(String name, String email) {
        this.name = name;
        this.email = email;
    }

    //Copy Constructor
    public Student(Student student) {
        this.name = student.name;
        this.email = student.email;
    }
}
```

Shallow & Deep Copy

When we do a copy of some entity to create two or more than two entities such that changes in one entity are reflected in the other entities as well, then we can say we have done a shallow copy. In shallow copy, new memory allocation never happens for the other entities, and the only reference is copied to the other entities. The following example demonstrates the same.

```
public class Test {
    public int n;
    String name;
    public int arr[];

    Test(int n,String name){
        this.n = n;
        this.name = name;
        this.arr = new int[n];
        for(int i=0;i<n;i++){
            arr[i] = i+1;
        }
    }

    //Copy Constructor
    Test(Test X){
        //Copying the references (Shallow Copy)
        this.n = X.n;
        this.name = X.name;
        this.arr = X.arr;
    }
}
```

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

        //Parametrised Constructor
        Test t1 = new Test(6,"Test1");
        //Copy Constructor Call
        Test t2 = new Test(t1);

        t2.n = 7;
        t2.name = "Test2";
        t2.arr[0] = 56;

        //Changes in T2's array are also reflect in T1's Array
        System.out.println(t1.arr);  // 56, 2,3,4,5,6
        System.out.println(t2.arr); // 56,2,3,4,5,6
    }
}
```

In the above example, we see we are creating a shallow copy in Line Copy Constructor of Test Class. Both the array references point to the same memory, the ideal way to do it would be to create a new array inside the copy constructor.

```
class Test{
    ...
     Test(Test X){
        this.n = X.n;
        this.name = X.name;

        // allocate a new memory - Deep Copy
        this.arr = new int[this.n];
        for(int i=0;i<n;i++){
            this.arr[i] = X.arr[i];
        }
    }
```

You might be wondering why didn't we create a deep copy for `int` and `String` data-types. This is because of the way java memory model works, for primitive data types like int, float etc Java always create new memory for different objects, and for strings because of immutability whenever you try to update the value of a String object, a new String object is automatically created in the string pool and the reference starts pointing to it. Hence for `int' and 'string' even if they are changed in T2 object, the changes won't be reflected in T1.

Java Memory Model

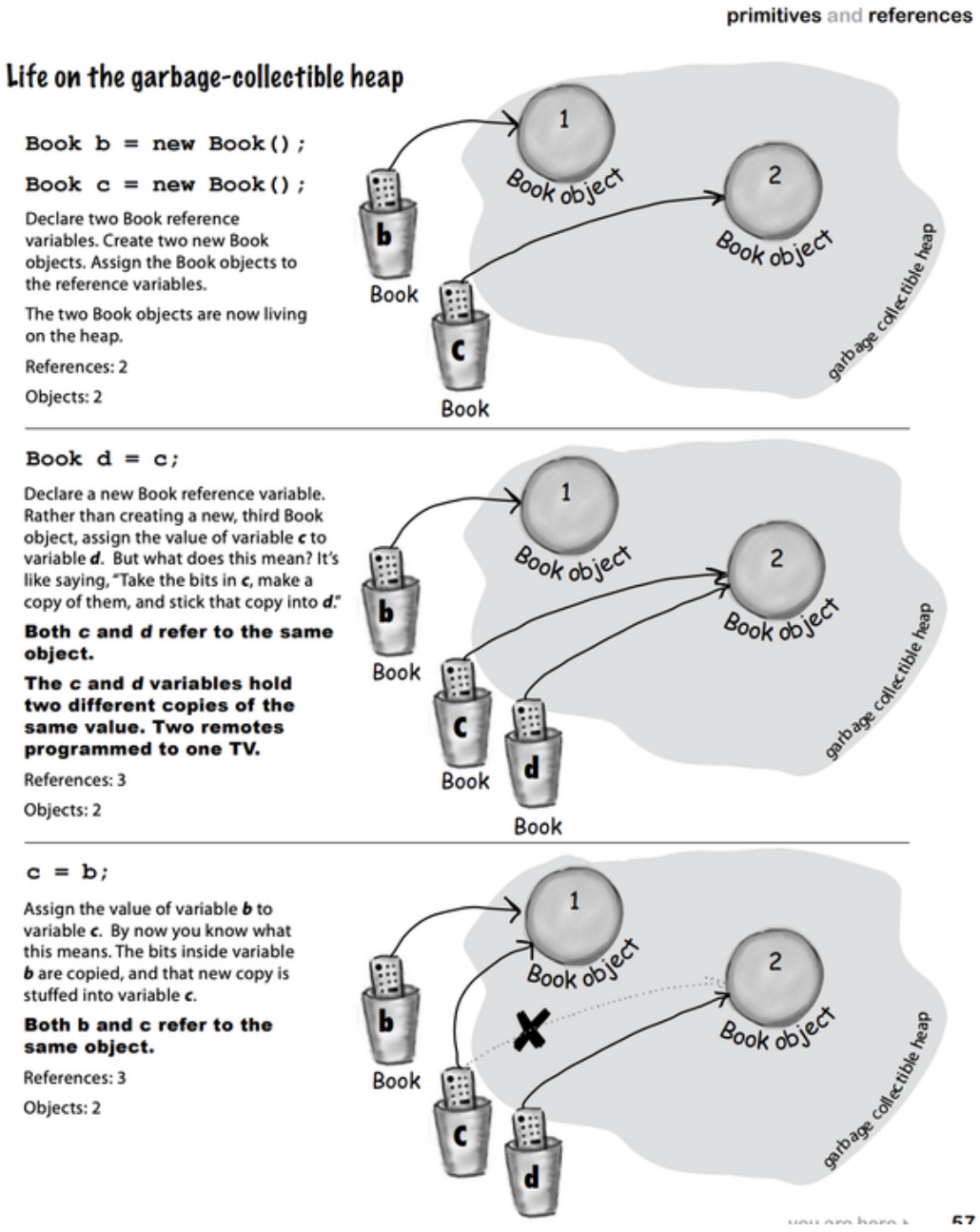
Let's try to understand how are objects stored in the memory. Calling a constructor with the command `new` causes several things to happen. First, space is reserved in the heap memory for storing object variables. Then default or initial values are set to object variables (e.g. an int type variable receives an initial value of 0). Lastly, the source code in the constructor is executed.

A constructor call returns a reference to an object. A reference is information about the location of object data.

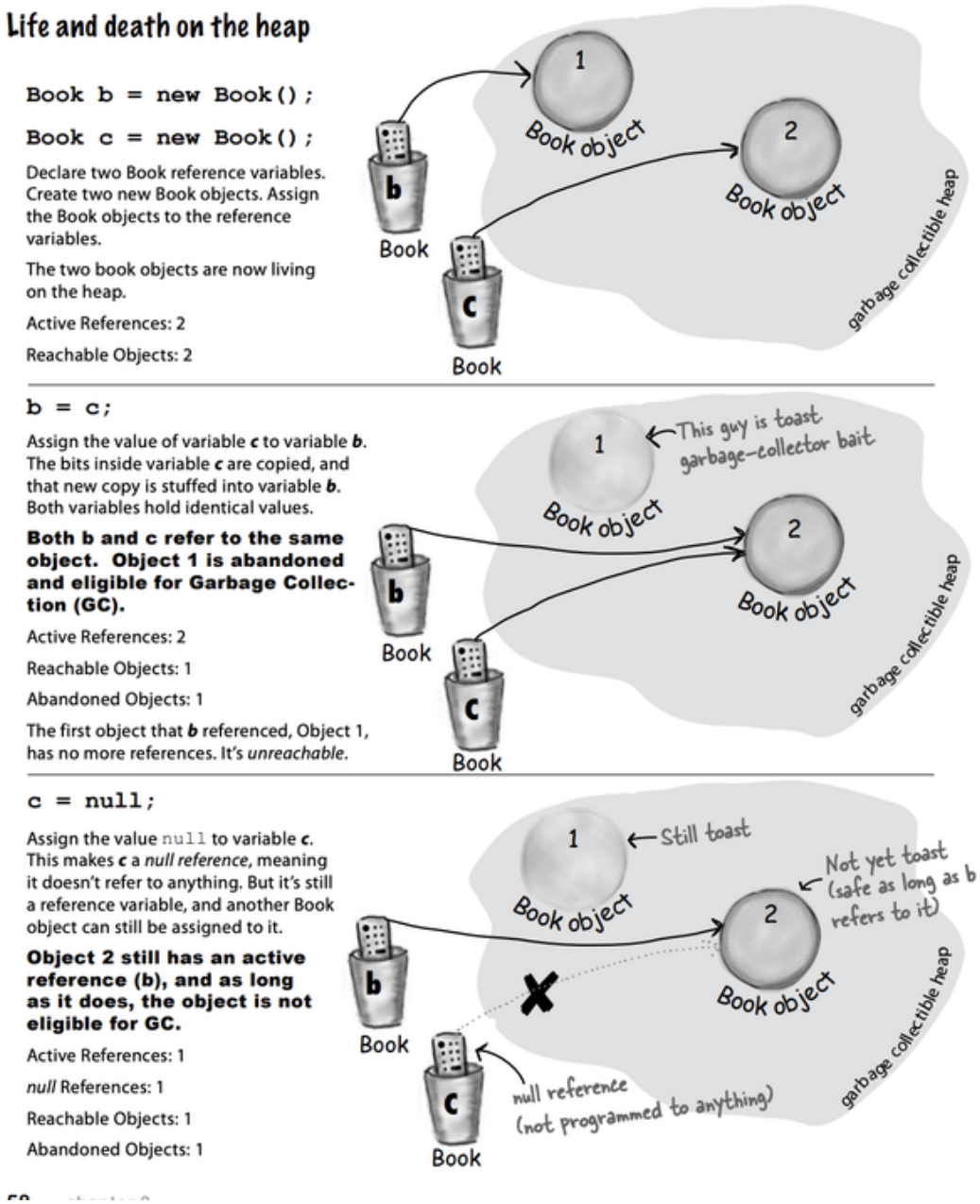
```
Player p1 = new Player("Prateek");
```

Here `p1` stores the location of object on the heap, and hence `p1` in a reference to the newly created player object.

Objects & Object References



Life and Death on Heap



Project - Guess Game using OOPS Concepts

Problem Statement - Create a 3 player game, in which a computer generates a random integer between 1-9. Each player has to make a random guess, guessing the number. The player takes turn in order to make a guess, the player who guesses the number correctly wins the game, if all three players make a wrong guess, the game starts again with computer making a new guess. Think about the entities, and their attributes, and the actions they can perform. Design & execute the game using OOPS principles.

Player.java

```java
package GuessGame;

public class Player {
    //Attributes
    String name;
    private int guess;

    //Methods
    Player(String name){
        this.name = name;
    }

    int getGuess(){
        return guess;
    }

    void makeGuess(){
        this.guess = (int)(Math.random()*9) + 1;
        System.out.println(this.name + " guessed "+this.guess);
    }
}
```

Game.java

```java
package GuessGame;

public class Game {
    int computerGuess;
    Player p1,p2,p3;

    Game(String n1,String n2,String n3){
        //init players
        p1 = new Player(n1);
        p2 = new Player(n2);
        p3 = new Player(n3);
    }

    private boolean checkWinner(){
        if(p1.getGuess()==computerGuess){
            System.out.println(p1.name + " wins");
            return true;
        }
        else if(p2.getGuess()==computerGuess){
            System.out.println(p2.name + " wins");
            return true;
        }
        else if(p3.getGuess()==computerGuess){
            System.out.println(p3.name + " wins");
            return true;
        }
        return false;
    }
```

```
    void launch(){
        //update the computer Guess
        System.out.println("Welcome to Game");
        this.computerGuess = (int)(Math.random()*9) + 1;

        while(true){
            System.out.println("Computer Guessed" + this.computerGuess);
           p1.makeGuess();
           p2.makeGuess();
           p3.makeGuess();

           if(checkWinner()){
               System.out.println("Game Over");
               break;
           }
           else{
               this.computerGuess = (int)(Math.random()*9) + 1;
           }
        }
    }
}
```

Launcher.java

```
package GuessGame;

public class Launcher {
    public static void main(String[] args) {
        Game firstGame = new Game("Sachin","Harsh","Suraj");
        firstGame.launch();
    }
}
```

Diagram References: [Head First Java Book]