JAVA constructor

জাভা কন্সট্রাকটর

By

Turzo Ahsan Sami

133014007

Dept. of Computer Science & Engineering

University of Liberal Arts Bangladesh

Submitted in partial fulfillment of the module

CSE 404 – Software Engineering

To

Nabeel Mohammed, PhD

জাভা কন্সট্রাকটর

জাভা প্রোগ্রামিং এ কন্সট্রাকটর এমন একটা মেথড যা দিয়ে অবজেক্ট ইনিশিয়ালাইজ করা যায়। অবজেক্ট তৈরীর সময় জাভা কন্সট্রাকটর তৈরী হয় এবং অবজেক্টকে ডাটা প্রোভাইড করে। জাভা কন্সট্রাকটর একটা বিশেষ ধরণের মেথড। জাভা কন্সট্রাকটরের নিয়ম হল-

১। জাভা কন্সট্রাকটরের নাম ক্লাসের নামে হতে হবে। জাভা কেস সেন্সিটিভ, তাই ক্লাসের নাম যেখাভে যে কেসে লেখা, জাভা কন্সট্রাকটরের নামও সেই একই কেসে একই ভাবে লিখতে হবে।

২। জাভা কন্সট্রাকটরের কোনও রিটার্ন টাইপ নাই। রিটার্ন টাইপ লিখলে কম্পাইলার এরর দেখাবে।

জাভা কন্সট্রাকটর ২ ধরণের হতে পারে-

১। ডিফল্ট জাভা কন্সট্রাকটর- এই ধরণের জাভা কন্সট্রাকটরে প্যারামিটার পাস করা হয় না।

উদাহরণ ১-

```
class Bike{
    Bike(){
        System.out.println("Bike is created");
    }
    public static void main(String args[]){
    Bike b = new Bike();
    }
}
```

Output:

Bike is created

জাভা প্রোগ্রামে কোনও কন্সট্রাকটর না লেখা হলে কম্পাইলার একটা ডিফল্ট কন্সট্রাকটর ক্রিয়েট করে নেয়। ডিফল্ট কন্সট্রাকটর অবজেক্টকে একটা ডিফল্ট ভ্যালু দিয়ে ইনিশিয়ালাইজ করে- যেমন 0, null।

উদাহরণ ২-

```
class Student3{
int id;
String name;
void display(){
        System.out.println(id+" "+name);
}
public static void main(String args[])
{
        Student3 s1=new Student3();
        Student3 s2=new Student3();
        s1.display();
        s2.display();
}
}
```

Output:

```
0 null
0 null
```

এই ক্লাসে কোনও কন্সট্রাকটর লেখা হয় নি। কিন্তু কম্পাইলার নিজে একটা কন্সট্রাকটর তৈরী করে নিয়েছে এবং 0, null। দিয়ে অবজেক্ট ইনিশিয়ালাইজ করেছে।

২। প্যারামিটারাইজড জাভা কন্সট্রাকটর- এই ধরণের জাভা কন্সট্রাকটরে এক বা একাধিক প্যারামিটার পাস করানো হয়। এইসব প্যারামিটার দিয়ে বিভিন্ন অবজেক্টকে বিভিন্ন ভ্যালু দিয়ে ইনিশিয়ালাইজ করা হয়।

উদাহরণ ৩-

```
class Student{
        int id;
        String name;
        Student(int i, String n){
                id = i;
                name = n;
        }
        void display(){
                System.out.println(id+" "+name);
        }
        public static void main(String args[]){
                Student s1 = new Student4(1,"A");
                Student s2 = new Student4(8,"B");
                s1.display();
                s2.display();
          }
}
```

Output:

```
1 A
8 Q
```

এখানে স্টুডেন্ট ক্লাসে ২ ধরণের প্যারামিটার পাঠানো হল- ইন্টিজার ও স্ট্রিং। প্যারামিটারাইজড কন্সট্রাকটরে একাধিক প্যারামিটার পাঠানো যায়।

কন্সট্রাকটরের সাথে মেথডের পার্থক্যঃ

১। কন্সট্রাক্টর কিছু রিটার্ন করে না, কিন্তু মেথড রিটার্ন করতেও পারে।
২। কন্সট্রাকটর ইমপ্লিসিটলি ইনভোক করা হয়, মেথড এক্সপ্লিসিটলি ইনভোক করা হয়।
৩। মেথড অবজেক্তের বিহেভিয়ার এক্সপোজ করে, কন্সট্রাকটর অবজেক্টের স্টেট ইনিশিয়ালাইজ করে।
৪। কন্সট্রাক্টর লেখা না হলে জাভা কমপাইলার ডিফল্ট কন্সট্রাক্টিকর ধরে নেয়, কিন্তু মেথড লেখা না হলে কমপাইলার মেথড ধরে নেয় না।
৫। কন্সট্রাকটরের নাম ক্লাসের নামে হতে হবে, মেথডের নামের ক্ষেত্রে এমন বাধ্যবাধকতা নাই।

কন্সট্রাকটর ওভারলোডিংঃ

একই ক্লাসে একাধিক কন্সট্রাকটর রাখাকে কন্সট্রাকটর ওভারলোডিং বলা হয়। কিন্তু একই ক্লাসে একাধিক কন্সট্রাকটর রাখতে হলে কন্সট্রাকটরগুলোর প্যারামিটারের সঙ্খ্যা ও টাইপ অবশ্যই আলাদা হতে হবে।

উদাহরণ ৪-

```
class Student{

        int id;

        String name;

        double cgpa;

        Student(int i,String n){

                id = i;

                name = n;

        }

         Student(int i,String n,int a){
```

```
        id = i;
        name = n;
        cgpa = c;
    }
    void display(){
        System.out.println(id+" "+name+" "+cgpa);
    }
      public static void main(String args[]){
        Student s1 = new Student(1,"A");
        Student s2 = new Student(8,"Qq",3.4);
        s1.display();
        s2.display();
      }
}
```

Output:

```
1 A 0
8 Qq 3.4
```

এখানে প্রথম কন্সট্রাকটর শুধুমাত্র একটা ইন্টিজার ও একটা স্ট্রিং প্যারামিটার হিসেবে নেয় আর স্বিতীয় কন্সট্রাকটর একটা ইন্টিজার, একটা স্ট্রিং ও একটা ডাবল টাইপ প্যারামিটার হিসেবে নেয়। তাই যখন কন্সট্রক্টর শুধু ইন্টিজার ও স্ট্রিং দিয়ে কল করা হল তখন প্রথম কন্সট্রাক্টর রান করলো। কিন্তু যখন ইন্টিজার, স্ট্রিং ও ডাবল দিয়ে কল করা হল তখন দ্বিতীয় কন্সট্রাকটর রান করে।

জাভা কপি কন্সট্রাকটরঃ

সি++ এর মত জাভায় কপি কন্সট্রাকটর না থাকলেও জাভায় ৩ ভাবে এক অবজেক্টের ভ্যালু অন্য অবজেক্টে কপি করা যায়-

১। কন্সট্রাকটর দিয়ে

২। এক অবজেক্টের ভ্যালু অন্য অবজেক্তে এসাইন করে

৩। অবজেক্টে ক্লাসের clone() মেথড কল করে

উদাহরণ ৫-

```
class Student{
  int id;
  String name;
  Student(int i,String n){
  id = i;
  name = n;
  }
  Student(Student s){
  id = s.id;
  name =s.name;
  }
  void display(){System.out.println(id+" "+name);}

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

```
        Student s1 = new Student(11,"Aq");
        Student s2 = new Student(s1);
        s1.display();
        s2.display();
        }
    }
```

Output:

```
11 Aq
11 Aq
```

এখানে এক অবজেক্টের ভ্যালু দ্বিতীয় অবজেক্টে কপি করা হল কন্সট্রাকটরের মাধ্যমে।

উদাহরণ ৬-

```
class Student{
        int id;
        String name;
        Student7(int i,String n){
        id = i;
        name = n;
        }
        Student(){ }
        void display(){
            System.out.println(id+" "+name);
```

```
    }

    public static void main(String args[]){
    Student s1 = new Student(11,"Qq");
    Student s2 = new Student();
    s2.id=s1.id;
    s2.name=s1.name;
    s1.display();
    s2.display();
    }
}
```

Output:

```
11 Qq
11 Qq
```

এখানে এক অবজেক্টের ভ্যালু দ্বিতীয় অবজেক্টে কপি করা হল কন্সট্রাক্টর ছাড়া। s2 অবজেক্টে সরাসরি s2 অবজেক্টের ভ্যালু কপি করা হল।

উদাহরণ ৭-

```
class Cricketer {
    String name;
    String team;
    int age;
```

```
        Cricketer () {                          // ডিফল্ট কন্সট্রাকটর
                name ="";
                team ="";
                age = 0;
        }
        Cricketer(String n, String t, int a){    // কন্সট্রাক্টর ওভারলোড
                name = n;
                team = t;
                age = a;
        }
         Cricketer (Cricketer ckt) {            // কপি কন্সট্রাক্টর
                name = ckt.name;
                team = ckt.team;
                age = ckt.age;
        }
        public String toString(){
                return "this is " + this.name + " of "+this.team;
        }

        public static void main(String[] args){                 // মেইন মেথড
                Cricketer c1 = new Cricketer();
                Cricketer c2 = new Cricketer("Amla", "South Africa", 30);
                Cricketer c3 = new Cricketer(c2 );
                System.out.println(c2);
                System.out.println(c3);
                c1.name = "Warner";
                c1.team= "Australia";
                c1.age = 32;
                System .out. println(c1);
        }
}
```

output:

this is Amla of South Africa
this is Amla of South Africa
this is Warner of Australia

কন্সট্রাকটর চেইনঃ

উদাহরণ ৮-

```
class GrandParent {
  int a;

  GrandParent(int a) {
    this.a = a;
  }
}

class Parent extends GrandParent {
  int b;

  Parent(int a, int b) {
    super(a);
    this.b = b;
  }

  void show() {
    System.out.println("a = " + a);
    System.out.println("b = " + b);
  }
}

class Child {
  public static void main(String[] args) {
    Parent object = new Parent(8, 9);
    object.show();
  }
}
```

output:

```
a = 8

b = 9
```

কন্সট্রাক্টর চেইন ঘটে যখন একটা ক্লাস আরেকটা ক্লাসকে ইনহেরিট করে। ইনহেরিটেন্সের ক্ষেত্রে যেমন সাবক্লাস সুপারক্লাসের প্রোপার্টি গ্রহণ করতে পারে। সুপারক্লাস এবং সাবক্লাসে কনস্ট্রাকটর মেথড থাকতে পারে। যখন সাবক্লাসের অবজেক্ট ক্রিয়েট করা হয়, তখন এর কন্সট্রাক্টর কল করা হয়, এবং এট্রিবিউট ইনিশিয়ালাইজ করা হয়। এরপর সুপারক্লাসের কনস্ট্রাকটরকে কল করতে হলে super কিওয়ার্ড এর মাধ্যমে সুপার ক্লাসের কন্সট্রাক্টরের কাছে আর্গুমেন্ট পাস করা হয়। এই উদাহরণে parent ক্লাস grandparent ক্লাসকে এক্সটেন্ড করেছে। এরপর child ক্লাস থেকে parent ক্লাসের অবজেক্ট তৈরী করে কন্সট্রাকটরের কাছে ২টা প্যারামিটার পাটানো হয়েছে। এখানে parent ক্লাস super(a) কিওয়ার্ডের মাধ্যমে grandparent ক্লাসের অবজেক্ট তৈরী করেছে। এরপর show মেথডের মাধ্যমে ভ্যালু প্রিন্ট করেছে।

উদাহরণ ৯-

```
class Human
{
	String s1, s2;
	public Human()
	{
		s1 ="Super class";
		s2 ="Parent class";
	}
	public Human(String str)
	{
		 s1= str;
		 s2= str;
	}
}
class Boy extends Human
{
	public Boy()
	{
		s2 ="Child class";
	}
	public void disp()
	{
		 System.out.println("String 1 is: "+s1);
		 System.out.println("String 2 is: "+s2);
	}
	public static void main(String args[])
```

```
    {
        Boy obj = new Boy();
        obj.disp();
    }
}
```

Output:

```
String 1 is: Super class
String 2 is: Child class
```

এইখানে, প্রথমে সুপারক্লাসের কনস্ট্রাকটর কল হয়েছে। এরপর সাবক্লাসের কনস্ট্রাকটর কল হয়েছে। সাবক্লাসের কনস্ট্রাকটর কল হওয়ার পর সুপারক্লাসের কনস্ট্রাকটরের s2 এর ভ্যালু ওভাররিডেন হয়ে গেছে।

জাভা জেনেরিকস

আমরা জাভা-এর টাইপ সিস্টেম সর্ম্পকে জানি। আমরা জানি জাভাতে কোন প্রোগ্রাম লিখতে হলে আমাদের কে টাইপ বলে দিতে হয় । যেমন আমরা যদি একটি মেথড লিখি তাহলে মেথডটি কি টাইপ প্যারামিটার এক্সেপ্ট করবে তা বলে দিতে হয়। তবে জাভাতে একটি ফিচার আছে যাতে করে আমরা অনেক সময় টাইপ না বলে দিয়েই কোড লিখতে পারি। আমরা জেনেরিকস শুরু করার আগে একটি গুরুত্বপূর্ণ তথ্য জেনে নিই- জাভা প্রোগ্রামিং ল্যাংগুয়েজ এ সব ক্লাস java.lang.Object ক্লাসটিকে ইনহেরিট করে। জেনেরিকস দিয়ে আমরা যখন অবজেক্ট তৈরি করবো তখন টাইপ প্যারিমিটারাইজ করতে পারি। অর্থাৎ আমরা যখন new অপারেটর দিয়ে অবজেক্ট তৈরি করবো তখন আসলে ডিসিশন নেবো এটির টাইপ কি হবে। এর আগে আমরা এমন ভাবে একটা ক্লাস বা মেথড লিখে ফেলতে পারি যাতে করে এটি যে কোন টাইপ এর জন্যে কাজ করে।

উদাহরণ ৯-

```
public class Generic<T> {
	T obj;
	public Generic(T obj) {
		this.obj = obj;
	}
	public T getObj() {
		return obj;
	}
	public void showType() {
		System.out.println("Type of T is: " + obj.getClass().getName());
	}
	public static void main(String[] args) {
		Generic<Integer> iObj;
		iObj = new Generic<Integer>(8);
```

```
        iObj.showType();
        int v = iObj.getObj();
        System.out.println("value: " + v);
        Generic<String> strOb = new Generic<String>("This is a Generics Test");
        strOb.showType();
        String str = strOb.getObj();
        System.out.println("value: " + str);
    }
}
```

Output:

Type of T is: java.lang.Integer value: 88 Type of T is: java.lang.String value: This is a Generics Test

একটি সিম্পল ক্লাস , এখানে T হচ্ছে টাইপ প্যারামিটার যা অবজেক্ট তৈরি করার সময় রিয়েল টাইপ দিয়ে রিপ্লেস হবে। একটা টাইপ ভ্যারিয়বল ডিক্লেয়ার করা হলো। কনস্ট্রাকটর নেয়া হল। কন্সট্রাকটর এর কাজ হল একটি রিয়েল অবজেক্ট আর্গুমেন্ট হিসেবে নেয়া। অবজেক্টটি একসেস করার জন্যে একটি পাবলিক মেথড রাখা হল এবং রানটাইমে অবজেক্ট-এর টাইপ আসলে কি , তা প্রিণ্ট করা হল। অবজেক্ট তৈরি করার পর iObj রেফারেন্স এ এসাইন করে কনস্ট্রাকটর আর্গুমেন্ট হিসেবে 8 পাস করা হল। রানটাইম-এ তাহলে জেনেরিক ক্লাসটিতে T obj একটি ইন্টিজার হবে। ইন্টিজার ভ্যালুটি এর ভ্যালু v তে রাখা হল। স্ট্রিং টাইপ দিয়ে পরীক্ষা করা হল, স্ট্রিং এর ভ্যালু str এ রাখা হল। আউটপুট গুলো থেকে বুঝা যাচ্ছে প্রোগ্রামটি সঠিক ভাবে কাজ করছে এবং একটি জেনেরিক ক্লাসে একটি ইন্টিজার এবং একটি স্ট্রিং প্যারামিটারাইজড করতে পেরেছি। এভাবে আমরা আরও অন্যান্য টাইপ-ও প্যারামিটারাইজড করে পারি।

এবার আরও ভালভাবে এই প্রোগ্রামটি খেয়াল করা যাক-

```
public class Generic<T> {
}
```

এখানে T হচ্ছে টাইপ প্যারামিটার। এটি মূলত একটি প্লেস হোল্ডার। T কে <> এর মধ্যে রাখা হয়।

আমরা সাধারণত যেভাবে ভ্যারিয়েবল ডিক্লেয়ার করি, সেভাবেই আমরা জেনেরিক্স-এ ভ্যারিয়েবল ডিক্লেয়ার করতে পারি। উদাহরণ-T obj;

এখানে T অবজেক্ট তৈরি করার সময় একটি রিয়েল অবজেক্ট অর্থাৎ আমরা যে অবজেক্ট প্যারিমিটারাইড করবো তা দ্বারা প্রতিস্থাপিত(replaced) হবে ।

আমরা জানি যে জাভা একটি স্ট্যাটিক টাইপ অর্থাৎ টাইপ সেইফ ল্যাংগুয়েজ। অর্থাৎ জাভা কোড কম্পাইল করার সময় এর টাইপ ইনফরমেশন ঠিক ঠাক আছে কিনা তা চেক করে নেয়।

অর্থাৎ -

```
Generic<Integer> iObj;
```

এখানে iObj একটি ইন্টিজার প্যারমিটাইরজড অবজেক্ট রেফারেন্স ।

```
iObj = new Generic<Double>(88.0); // Error!
```

এখন অবজেক্ট তৈরি করার সময় যদি ডাবল প্যারমিটাইরজড করি এবং iObj তে এসাইন করি, তাহলে

Error:(24, 16) java: incompatible types: Generic<java.lang.Double> cannot be converted to Generic<java.lang.Integer> কম্পাইল করার সময় এইর এরর দেখাবে।

জেনেরিকস শুধুমাত্র অবজেক্ট নিয়ে কাজ করে-

আমারা জানি যে, জাভা দুই ধরণের টাইপ সাপোর্ট করে- PrimitiveType এবং ReferenceType। জেনেরিকস শুধুমাত্র ReferenceType অর্থাৎ শুধু মাত্র অবজেক্ট নিয়ে কাজ রে।

তাই-

```
Generic<int> intObj = new Generic<int>(50);
```

এই স্ট্যাটমেন্ট টি ভ্যালিড নয়। অর্থাৎ প্রিমিটিভ টাইপ এর ক্ষেত্রে জেনেরিকস কাজ করবে না।

জেনেরিক ক্লাস এর সিনট্যাক্স-

```
class class-name<type-param-list > { }
```

জেনেরিক ক্লাস ইনসটেনসিয়েট করার সিনটেক্স-

```
class-name<type-arg-list > var-name = new class-name<type-arg-list >(cons-arg-list);
```

আমরা চাইলে একাধিক জেনেরিক টাইপ প্যারমিটাইরজড করতে পারি।

উদাহরণ ১০-

```
public class Tuple<X, Y> {
    private X x;
    private Y y;
    public Tuple(X x, Y y) {
            this.x = x;
            this.y = y;
    }
    public X getX() {
        return x;
    }

    public Y getY() {
        return y;
    }

    public void showTypes() {
        System.out.println("Type of T is " +
                x.getClass().getName() + " and Value: " + x);
        System.out.println("Type of V is " +
                y.getClass().getName() + " and Value: " + y);
    }

    public static void main(String[] args) {
        Tuple<String, String> tuple = new Tuple<String, String>("Hello", "world");
```

```
        tuple.showTypes();


        Tuple<String, Integer> person = new Tuple<>("Rahim", 45);
        person.showTypes();
    }
}
```

Output:

Type of T is java.lang.String and Value: Hello Type of V is java.lang.String and Value: world Type of T is java.lang.String and Value: Rahim Type of V is java.lang.Integer and Value: 45

একটি টাপলের মধ্যে আমরা চাইলে আরেকটি টাপল রাখে পারি -

```
Tuple<String, Tuple<Integer, Integer>> tupleInsideTuple = new Tuple<String, Tuple<Integer, Integer>>("Tuple", new Tuple<Integer, Integer>(45, 89));
```

তবে আমরা যদি জাভা ৭ অথবা ৮ ব্যবহার করি তাহলে উপরের লাইনটি সংক্ষিপ্তভাবে লিখতে পারি -

```
Tuple<String, Tuple<Integer, Integer>> tupleInsideTuple = new Tuple<>("Tuple", new Tuple<>(45, 89));
```

`

জাভা ৭ এ একটি নতুন অপারেটর সংযুক্ত হয়েছে যাকে বলা হয় ডায়মন্ড অপারেটর। এটি ব্যবহার করে আমরা জেনেরকস এ verbosity কিছুটা কমানো যায়। অর্থাৎ

```
Map<String, List<String>> anagrams = new HashMap<String, List<String>>();
```

এই স্ট্যাটমেন্ট-টি অনেকটাই বড়। এটি আমরা এভাবে লিখতে পারি -

```
Map<String, List<String>> anagrams = new HashMap<>();
```

অর্থাৎ জেনেরিকস লেখার সময় বাম পাশে টাইপ প্যারামিটার ইনফরমেশন গুলো লিখলে ডান পাশে লিখতে হয় না । এটি অটোম্যাটিক্যালী ইনফার করতে পারে।

Bounded Types

আমরা উপরে দুটি উদাহরণ দেখেছি যেগুলোতে আমরা যে কোন ধরণের টাইপ প্যারামিটারাইউজড করতে পারি। কিন্তু কখনো কখনো আমাদের টাইপ restrict করতে হয়। যেমন- আমরা একটি জেনেরিক ক্লাস

লিখতে চাই যা কিনা একটি এরে-তে রাখা কতগুলো নাম্বার-এর গড়(average) রিটার্ন করবে এবং আমরা চাই, এই এরে তে যে কোন ধরণের নাম্বার থাকতে পারে, যেমন- ইন্টিজার, ফ্লোটিং পয়েন্ট, ডাবল ইত্যাদি। আমরা টাইপ প্যারামিটার দিয়ে বলে দিতে চাই কখন কোনটা থাকবে। উদারহরণ দেখা যাক-

```
public class Stats<T> {
    T[] nums;

    public Stats(T[] nums) {
        this.nums = nums;
    }

    // Return type double in all cases.
    double average() {
        double sum = 0.0;
        for (T num : nums) {
            sum += num.doubleValue(); // Error!!!
        }

        return sum / nums.length;
    }
}
```

`

এভারেজ ক্যালকুলেট করার জন্য আমাদের এভারেজ মেথড সবসময় এরে থেকে ডাবল ভ্যালু এক্সেপেক্ট করে। কিন্তু আমাদের এরে-এর টাইপ যেহেতু যে কোন রকম হতে পারে, সুতরাং সব অবজেক্ট থেকে ডাবল ভ্যালু পাওয়ার উপায় নেই।

ইনফ্যাক্ট এই ক্লাসটি কিন্তু কম্পাইল হবে না।

এই ক্লাসটিতে আমরা একটি restriction এড করতে পারি যাতে করে এই টাইপ প্যারামিটার শুধুমাত্র নাম্বার(ইন্টিজার, ফ্লোটিং পয়েন্ট,ডাবল) হবে, নতুবা এটি কাজ করবে না।

আমরা জানি যে সব নিউমেরিক অবজেক্ট গুলোর সুপার ক্লাস হচ্ছে Number. এবং Number এ doubleValue() মেথড ডিফাইন করা আছে। সুতরাং আমাদের ক্লাসটিকে একটু পরিবর্তন করি।

```
public class Stats<T extends Number> {
    T[] nums;

    public Stats(T[] nums) {
        this.nums = nums;
    }

    // Return type double in all cases.
    double average() {
        double sum = 0.0;
        for (T num : nums) {
            sum += num.doubleValue(); // Error!!!
        }

        return sum / nums.length;
    }
}
```

একটু লক্ষ্য করুন-

```
public class Stats<T extends Number>{
}
```

আমরা ক্লাস ডেফিনেশনে আমাদের টাইপ প্লেসহোল্ডার T নাম্বারকে extend করে। এটি আমাদের টাইপ প্যারামিটার পাস করতে restrict করে । অর্থাৎ আমরা শুধু মাত্র সেসব টাইপ পাস করতে পারবো যারা Number এর সাব টাইপ।

সুতরাং আমাদের এই Stats ক্লাস এখন Integer, Double, Float, Long, Short, BigInteger, BigDecimal, Byte ইত্যাদি অবজেক্ট এর জন্যে কাজ করবে।

সুতরাং দেখা যাচ্ছে যে, জেনেরিকস এর সুবিধা ব্যবহার করে আমরা এই স্ট্যাট ক্লাসটি আলাদা আলাদা করে অনেকগুলো না লিখে একটি দিয়েই কাজ করে ফেলা সম্ভব হল।

Wildcard Arguments

নিচের উদাহরণটি লক্ষ্য করি-

```
ArrayList<Object> lst = new ArrayList<String>();
```

এটি যদি কম্পাইল করতে চেষ্টা করি, তাহলে কম্পাইলার incompatible types ইরর দেবে। কিন্তু আমরা জানি যে, সকল অবজেক্ট এর সুপার ক্লাস Object। তাছাড়া আমরা polymorphism থেকে জানি যে আমরা সাব ক্লাসের রেফারেন্স কে সুপার ক্লাসের রেফারেন্স এ এসাইন করতে পারি। সুতরাং উপরের স্ট্যাটমেন্ট-টি কাজ করার কথা।

নিচের উদাহরণ দুটি লক্ষ্য করি -

```
List<String> strLst = new ArrayList<String>();   // 1
List<Object> objLst = strList;                    // 2 - Compilation Error
```

`

২ নাম্বার লাইনটি কাজ করছে না । যদিও বা এটি কাজ করে এবং আর্বিট্রারি কোন একটি অবজেক্ট যদি objLst এড করা হয় তাহলে কিন্তু strList করাপ্টেড হয়ে যাবে এবং সেটি আর স্ট্রিং থাকবে না।

ধরা যাক, আমরা একটা print মেথড লিখতে চাই যা কিনা একটি লিস্ট এর ইলিমেন্ট গুলো প্রিন্ট করে।

```
public static void print(List<Object> lst) {  // accept List of Objects only,
        // not List of subclasses of object
        for (Object o : lst) {
            System.out.println(o);
        }
```

```
    }
```

এটি কিন্তু শুধুমাত্র List<Object> একসেপ্ট করবে , List<String> অথবা List<Integer> করবে না।

উদাহরণ ১১-

```
    public static void main(String[] args) {
        List<Object> objLst = new ArrayList<Object>();
        objLst.add(new Integer(55));
        printList(objLst);   // matches

        List<String> strLst = new ArrayList<String>();
        strLst.add("one");
        printList(strLst);  // compilation error
      }
```

এই সমস্যা দূর করার জন্যে জাভাতে একটি একটি অপারেটর ব্যবহার করা হয় - যার নাম wildcard (?)।

আমরা যদি আমাদের print() মেথডটি নিচের মতো করে লিখি, তাহলে কিন্তু আমাদের সমস্যা দূর হয়ে যাবে।

```
    public static void print(List<?> lst) {  // accept List of Objects only,
          // not List of subclasses of object
          for (Object o : lst) {
              System.out.println(o);
          }
    }
```

List<?> lst এর মানে হচ্ছে আমরা এর টাইপ আমাদের জানা নেই, এটি যে কোন টাইপ হতে পারে। যেহেতু সব টাইপ এর সুপার ক্লাস Object সুতরাং এটি যেকোন টাইপ এর জন্যে কাজ করবে।

Bounded Types এর মতো আমরা Wildcard Arguments কেও Bounded করে ফেলতে পারি ।

উদাহরণ -

```
public static void process(List<? extends Foo> list) { /* ... */ }
```

এটি শুধু মাত্রে Foo এর সাব ক্লাস গুলো কে প্রসেস করতে পারবে। একে Upper Bounded Wildcards বলে ।

আমরা যদি এমন কোন মেথড লিখতে চাই যা শুধু মাত্র Integer, Number, and Object প্রসেস করবে অর্থাৎ Integerএবং এর সুপার ক্লাস প্রসেস করবে তাহলে -

```
public static void addNumbers(List<? super Integer> list) {
}
```

একে Lower Bounded Wildcards বলে।

Generic Methods

আমরা মূলত এতোক্ষণ জেনেরিক ক্লাস নিয়ে কথা বলেছি। আমরা একটি ক্লাসকে জেনেরিক না করে শুধুমাত্রে এর একটি বা একাধিক মেথড কে জেনেরিক করে লিখতে পারি।

উদহরণ-

```
public class Util {
    // Generic static method
    public static <K, V> boolean compare(Pair<K, V> p1, Pair<K, V> p2) {
        return p1.getKey().equals(p2.getKey()) &&
               p1.getValue().equals(p2.getValue());
    }
}
```

এটি একটি জেনেরিকম মেথড।

জেনেরিক মেথড-এ রিটার্নটাইপ এর আগে টাইপ-প্লেস হোল্ডার <> লিখতে হয়।

জেনেরিক লিংকড লিস্ট-

```
public class SinglyLinkedList<Type> {
    private long size;
```

```
private Node<Type> head;
private Node<Type> tail;

public void addFirst(Type value) {
    addFirst(new Node<>(value));
}

public void addLast(Type value) {
    addLast(new Node<>(value));
}

private void addLast(Node<Type> node) {
    if (size == 0) {
        head = node;
    } else {
        tail.setNext(node);
    }
    tail = node;
    size++;
}

public void addFirst(Node<Type> node) {
    Node<Type> temp = head;
```

```
        head = node;
        head.setNext(temp);

        size++;

        if (size == 1) {
            tail = head;
        }
    }

    public Node<Type> getHead() {
        return head;
    }

    public Node<Type> getTail() {
        return tail;
    }

    public void removeFirst() {
        if (size != 0) {
            head = head.getNext();
            size--;
        }
```

```
        if (size == 0) {
            tail = null;
        }
    }

    public void removeLast() {
        if (size != 0) {
            if (size == 1) {
                head = null;
                tail = null;
            } else {
                Node<Type> current = head;

                while (current.getNext() != tail) {
                    current = current.getNext();
                }
                current.setNext(null);
                tail = current;

            }
            size--;
        }
    }
```

```
public Type getFirst() {

    return getHead().getValue();
}

// four scenario
// 1. empty list-  do nothing
// 2. single node : ( previous is null)
// 3. Many nodes
//       a. node to remove is first node
//       b. node to remove is the middle or last

public boolean remove(Type type) {
    Node<Type> prev = null;
    Node<Type> current = head;

    while (current != null) {
        if (current.getValue().equals(type)) {
            if (prev != null) {

                // just skip the current node. it works fine
                prev.setNext(current.getNext());

                if (current.getNext() == null) {
```

```
                tail = prev;
            }

            size--;
        } else {
            removeFirst();
        }

        return true;
    }

    prev = current;
    current = current.getNext();
  }

  return false;
}

public long getSize() {

  return size;
}
```

```
public void print() {
    System.out.print("Total elements : " + size + " -> ");
    Node node = head;
    while (node != null) {
        System.out.print(node.getValue().toString() + " ,");
        node = node.getNext();
    }
    System.out.println();
}

public void clear() {
    for (Node<Type> x = head; x != null; ) {
        Node<Type> next = x.next;
        x.next = null;
        x.value = null;
        x = next;
    }

    head = tail = null;
    size = 0;
}


private class Node<Type> {
```

```
        private Type value;
        private Node<Type> next;

        public Node(Type value) {
            this.value = value;
        }

        public Type getValue() {
            return value;
        }

        public void setValue(Type value) {
            this.value = value;
        }

        public Node<Type> getNext() {
            return next;
        }

        public void setNext(Node<Type> next) {
            this.next = next;
        }
    }
}
```

```
public class LinkedListDemo {
    public static void main(String[] args) {
        SinglyLinkedList<Integer> integers = new SinglyLinkedList<>();
        integers.addFirst(4);
        integers.addFirst(3);
        integers.addFirst(2);
        integers.addFirst(1);

        integers.print();

        System.out.println("Remove first and last elements..");
        integers.removeFirst();
        integers.removeLast();
        integers.print();

        System.out.println("add elements at last ");
        integers.addLast(5);
        integers.addLast(6);
        integers.addLast(7);
        integers.print();

        SinglyLinkedList<String> stringLinkedList = new SinglyLinkedList<>();
        stringLinkedList.addFirst("abcd");
```

```
            stringLinkedList.addFirst("efgh");
            stringLinkedList.addFirst("ijkl");
            stringLinkedList.addFirst("mnop");
            stringLinkedList.addFirst("qrst");
            stringLinkedList.print();
        }
    }
```

Output:

Total elements : 4 - 1 ,2 ,3 ,4 , Remove first and last elements.. Total elements : 2 - 2 ,3 , add elements at last Total elements : 5 - 2 ,3 ,5 ,6 ,7 , Total elements : 5 - qrst ,mnop ,ijkl ,efgh ,abcd ,