IT2030 - Object Oriented Programming

Lecture 08

Generics

Contents

- **Wrapper classes**
- **Introduction to Generics**
  - Generic classes - Declaration & Instantiation
  - Generic Methods – Implementation & Invocation
  - Bounded Type Parameters
  - Wildcards
  - Limitations in Generics

Wrapper class / Covering class

- Wrapper class in java provides the mechanism *to convert primitive datatype into object* and *object into primitive type*.

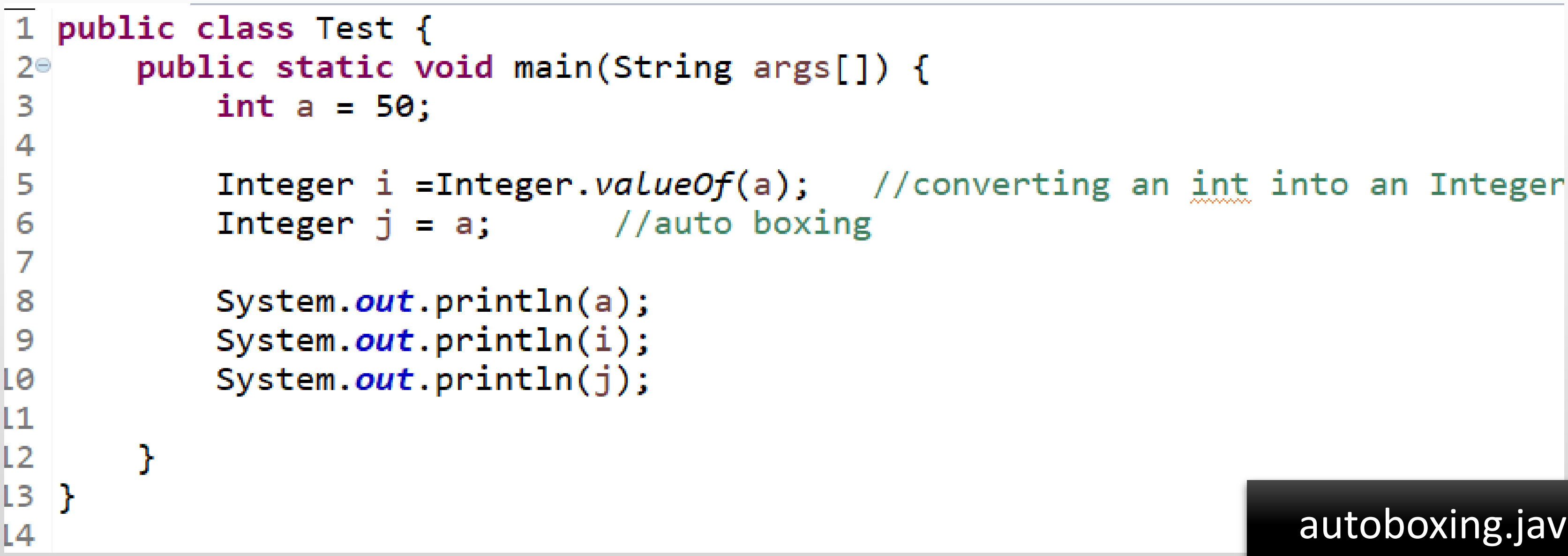
- For each primitive datatype, there exists a covering class in the *java.lang* package.

byte	→	Byte
short	→	Short
int	→	Integer
long	→	Long
float	→	Float
double	→	Double
char	→	Character
Boolean	→	Boolean

Wrapper class cont.

- The automatic conversion of primitive type into object is known as *autoboxing* and vice-versa *unboxing*.

- Since J2SE 5.0, autoboxing and unboxing feature converts primitive into object and object into primitive *automatically.*

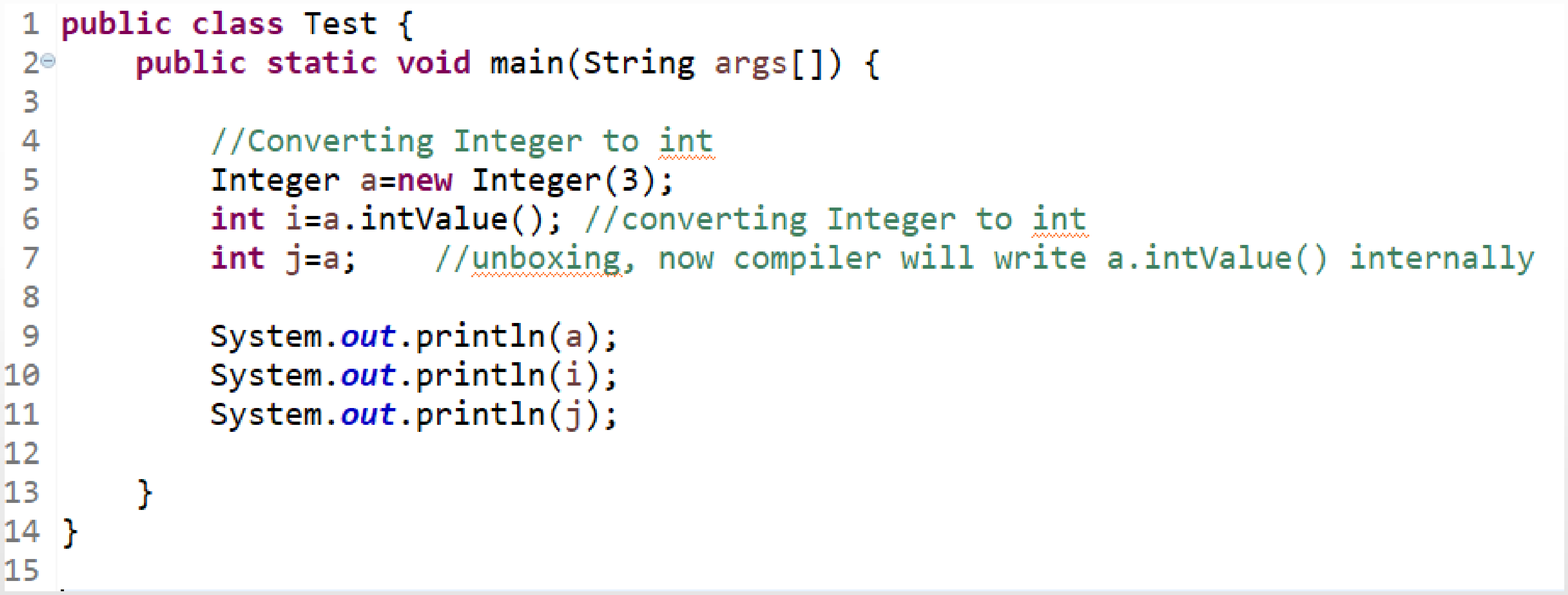


```java
public class Test {
    public static void main(String args[]) {
        int a = 50;

        Integer i =Integer.valueOf(a);   //converting an int into an Integer
        Integer j = a;      //auto boxing

        System.out.println(a);
        System.out.println(i);
        System.out.println(j);

    }
}

```

autoboxing.java

Wrapper class cont.

unboxing



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

        //Converting Integer to int
        Integer a=new Integer(3);
        int i=a.intValue(); //converting Integer to int
        int j=a;    //unboxing, now compiler will write a.intValue() internally

        System.out.println(a);
        System.out.println(i);
        System.out.println(j);

    }
}
```

unboxing.java

Generics in Java

- Concept of “generics” was introduced in JDK 5 to deal with type-safe objects.
- Introduction of generics has changed Java in 2 ways:
  - Added new syntax to Java language
  - Caused changes to many of the classes & methods in the core API
- With the use of generics, it is possible to create classes, interfaces & methods which works in type-safe manner.

Generics in Java cont.

- “Generics” means “parameterized types”.
- Many algorithms are same regardless of its data type that is applied. With generics, you can define the algorithm once independently of any specific type of data. Later, you can use the algorithm to a wide variety of data types without any additional effort.

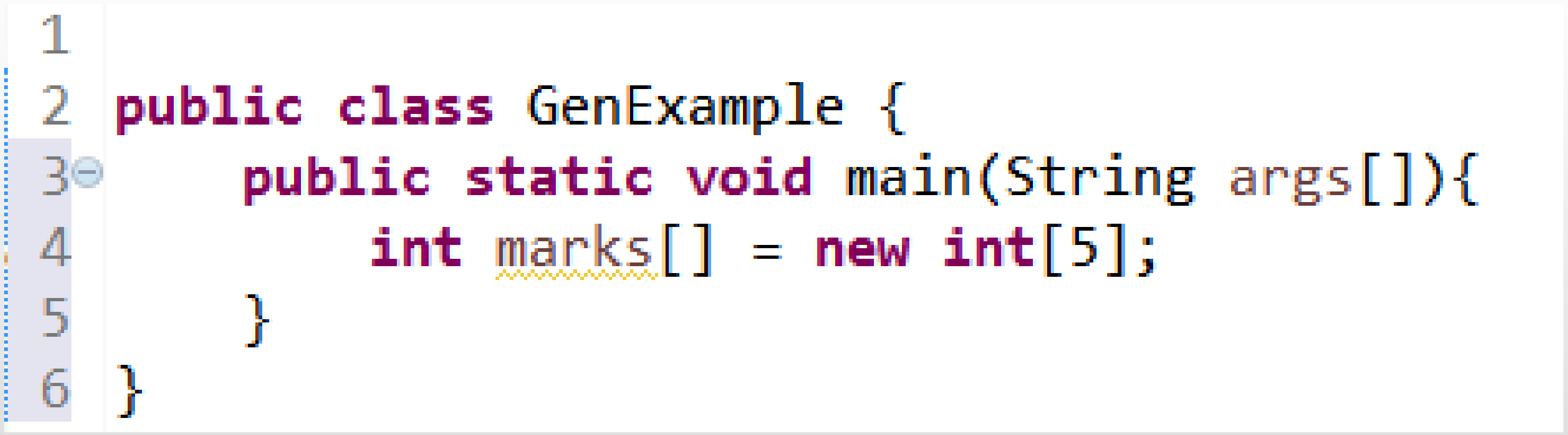
Advantages of Generics

1. **Type-safety :** We can hold only a single type of objects in generics. It doesn't allow to store other types of objects.
2. **Type casting is not required:** There is no need to typecast the object. All type conversions are implicit.
3. **Compile-Time Checking:** It is checked at compile time so problem will not occur at runtime. The good programming strategy says it is far better to handle the problem at compile time than at runtime.

Implementation

- When you are in need to store a list of values, you shall use an array.

e.g.: if you are storing marks of 5 students then you may create an array as shown in the code below



```java

public class GenExample {
    public static void main(String args[]){
        int marks[] = new int[5];
    }
}
```

Implementation cont.

Problem:

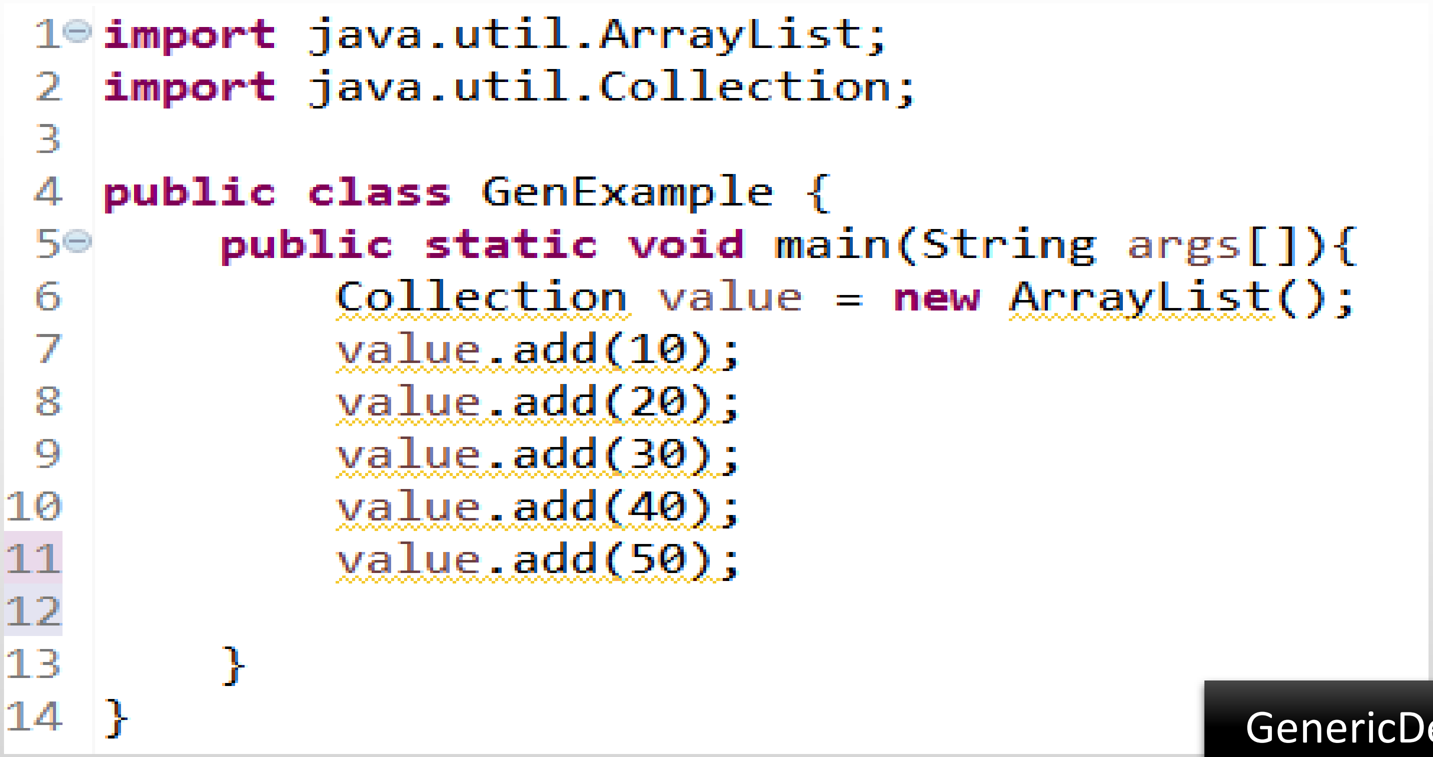
- *Size of the array is fixed.* If there is a change in the size of the array (in the number of elements that are stored), you have to modify the code manually.
- You may not be able to expand or shrink the array automatically as & when the element is being added.

Solution:

- Use Collection Interface

Implementation cont.

Shown below is an example of how Collection is used to store elements. Note that the list grows each & every time add() is called.



```java
import java.util.ArrayList;
import java.util.Collection;

public class GenExample {
    public static void main(String args[]){
        Collection value = new ArrayList();
        value.add(10);
        value.add(20);
        value.add(30);
        value.add(40);
        value.add(50);

    }
}
```

GenericDemo2.java

Implementation cont.

```
Collection value = new ArrayList();
```

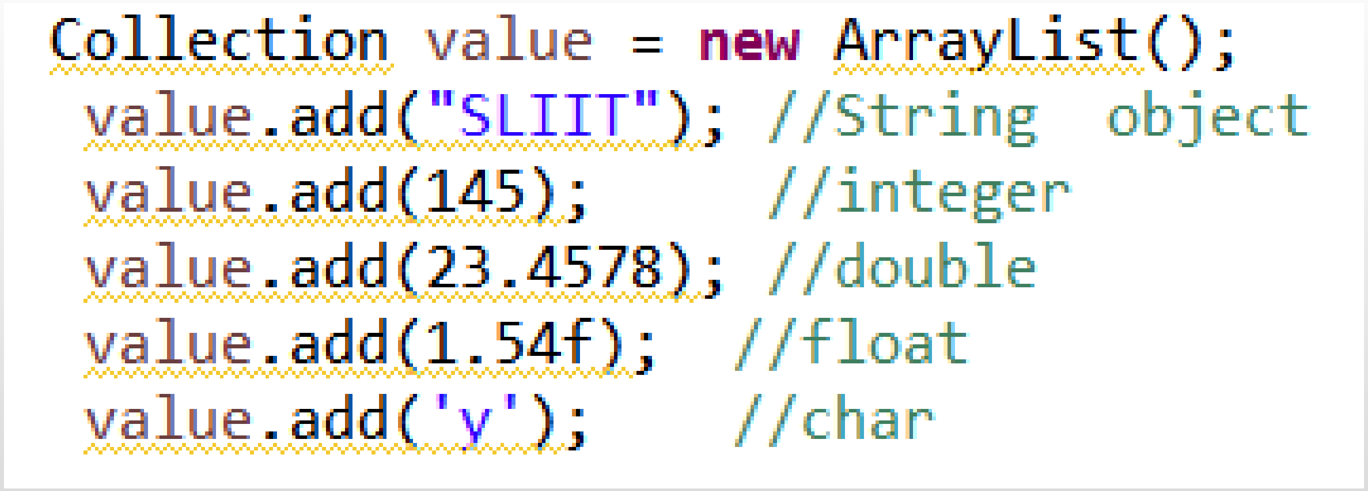
- This statement allows you to create a list of elements (list of marks).
- Note that Collection is an interface and cannot be instantiated directly.
- ArrayList is a class which implements the Interface List which extends the Interface Collection.

Implementation cont.

Collection value = new ArrayList();

Problem:

This list may contain any object as we have <u>not</u> specified the data type of the element to be added.



```
Collection value = new ArrayList();
 value.add("SLIIT"); //String  object
 value.add(145);     //integer
 value.add(23.4578); //double
 value.add(1.54f);  //float
 value.add('y');    //char
```

Implementation cont.

```
Collection value = new ArrayList();
```

Problem:

What if the requirement is to store only ***integers***?

Solution:

Make use of Generics!

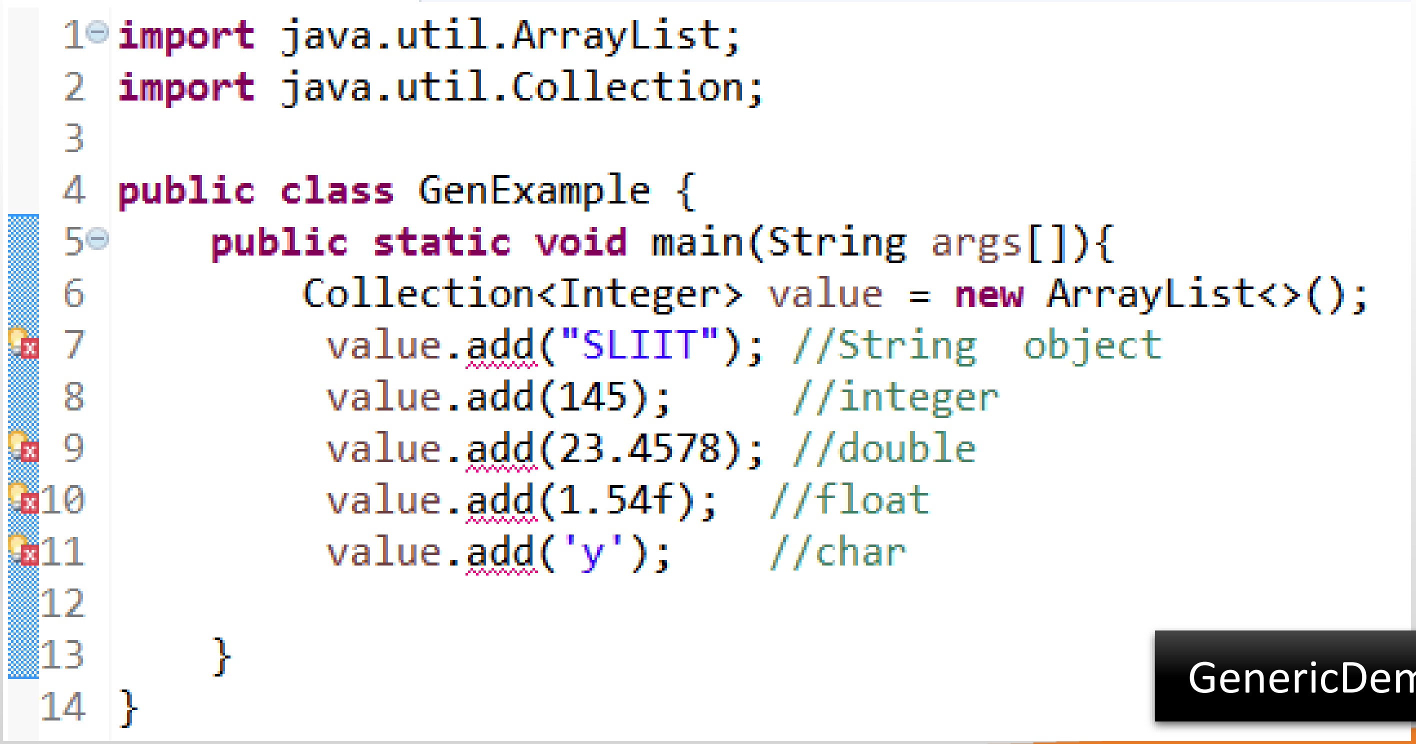
```
Collection <Integer> value = new ArrayList<>();
```

*Make use of Wrappers as primitive types do not support Generics.

Implementation cont.

Collection <Integer> value = new ArrayList<>();

- Now you may not be able to add elements other than Integer type. (Note the Compilation Error)



```
import java.util.ArrayList;
import java.util.Collection;

public class GenExample {
    public static void main(String args[]){
        Collection<Integer> value = new ArrayList<>();
         value.add("SLIIT"); //String  object
         value.add(145);     //integer
         value.add(23.4578); //double
         value.add(1.54f);  //float
         value.add('y');    //char

    }
}
```

GenericDemo3.java

Question:

Write a program to store a list of names, retrieve the names & display on the screen.

Hint: Use an ArrayList class

Type Parameters

The type parameter naming conventions are as follows:

T - Type

E - Element

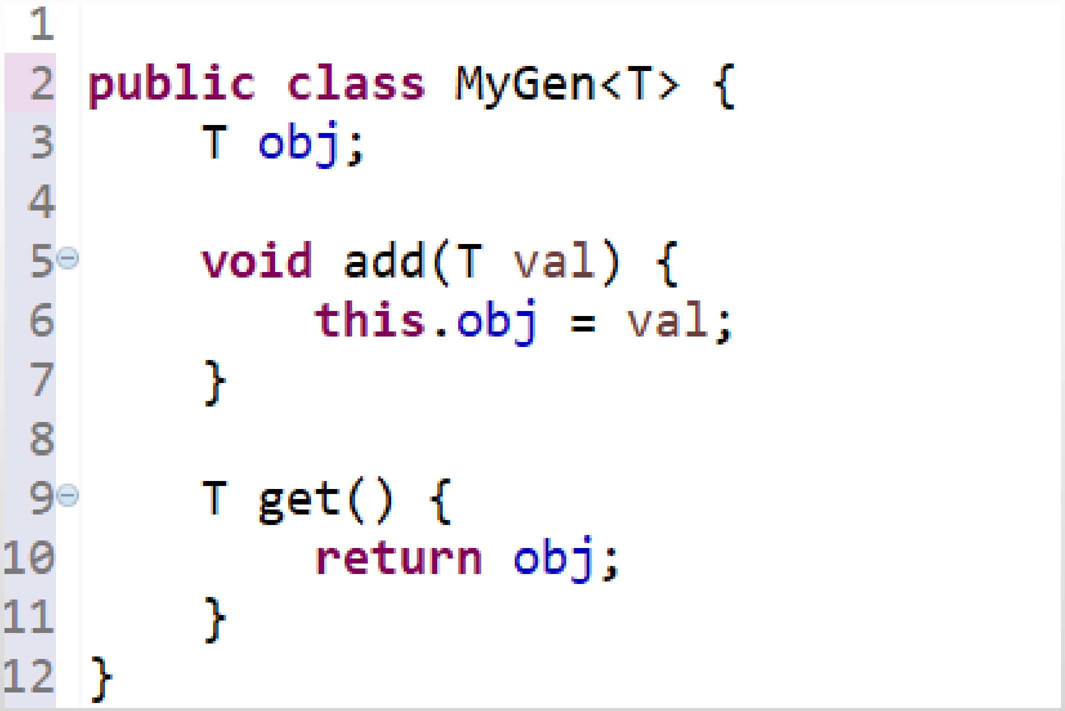
K - Key

N - Number

V - Value

Generic class

- A class that can refer to any type is known as generic class.
- Type T indicates that it can refer to any type (Integer, Double, Employee etc.)

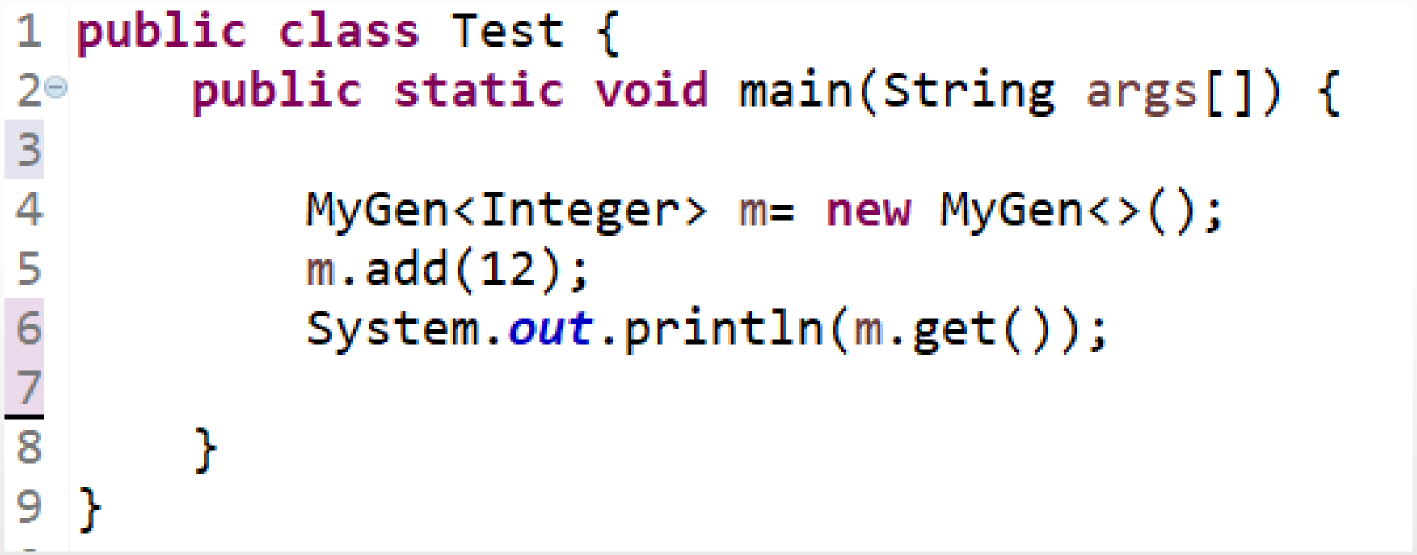


```java

public class MyGen<T> {
	T obj;

	void add(T val) {
		this.obj = val;
	}

	T get() {
		return obj;
	}
}
```

test.java

Generic class cont.

- You may make use of the generic class as shown below:



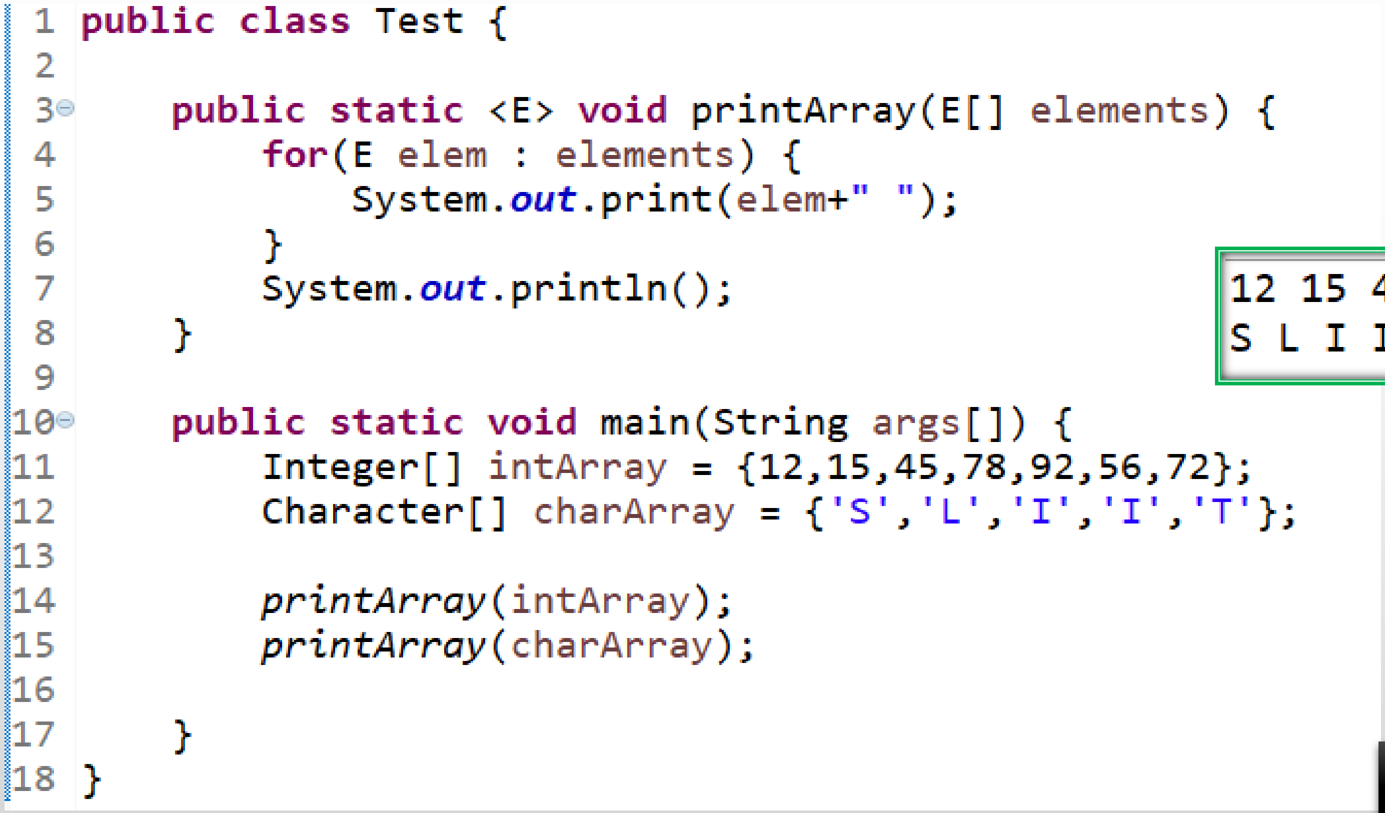
```java
public class Test {
    public static void main(String args[]) {

        MyGen<Integer> m= new MyGen<>();
        m.add(12);
        System.out.println(m.get());

    }
}
```

- Note that there will be a compilation error if you attempt to add any other type of data other than an integer.

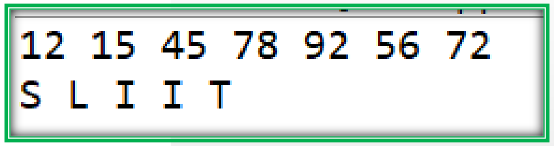
test.java

Generic method

- Like generic class, we can create generic method that can accept any type of argument.



```java
public class Test {

    public static <E> void printArray(E[] elements) {
        for(E elem : elements) {
            System.out.print(elem+" ");
        }
        System.out.println();
    }

    public static void main(String args[]) {
        Integer[] intArray = {12,15,45,78,92,56,72};
        Character[] charArray = {'S','L','I','I','T'};

        printArray(intArray);
        printArray(charArray);

    }
}
```

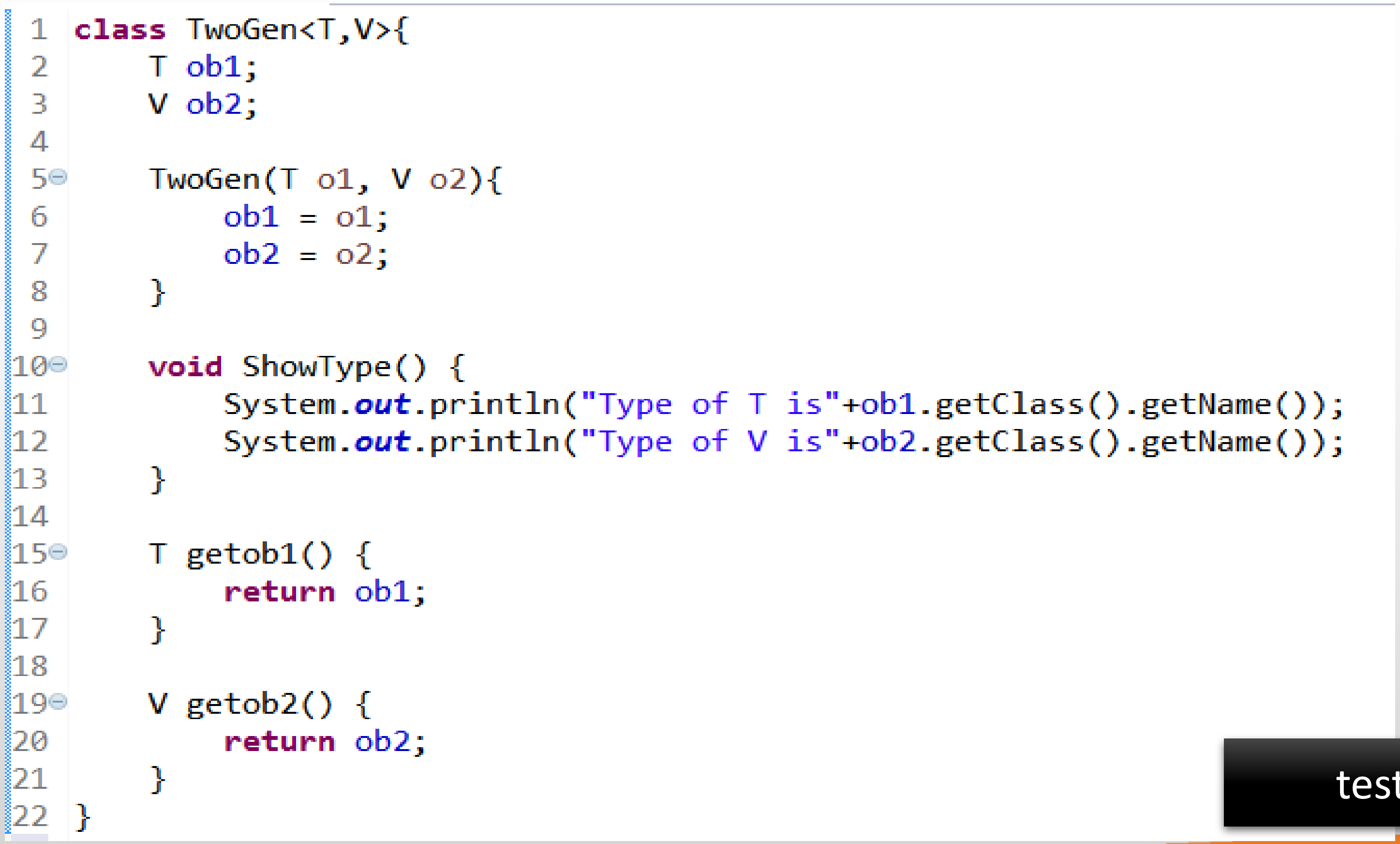


```
12 15 45 78 92 56 72
S L I I T
```

test1.java

Multiple Parameters

- A generic class or method can have multiple type parameters.

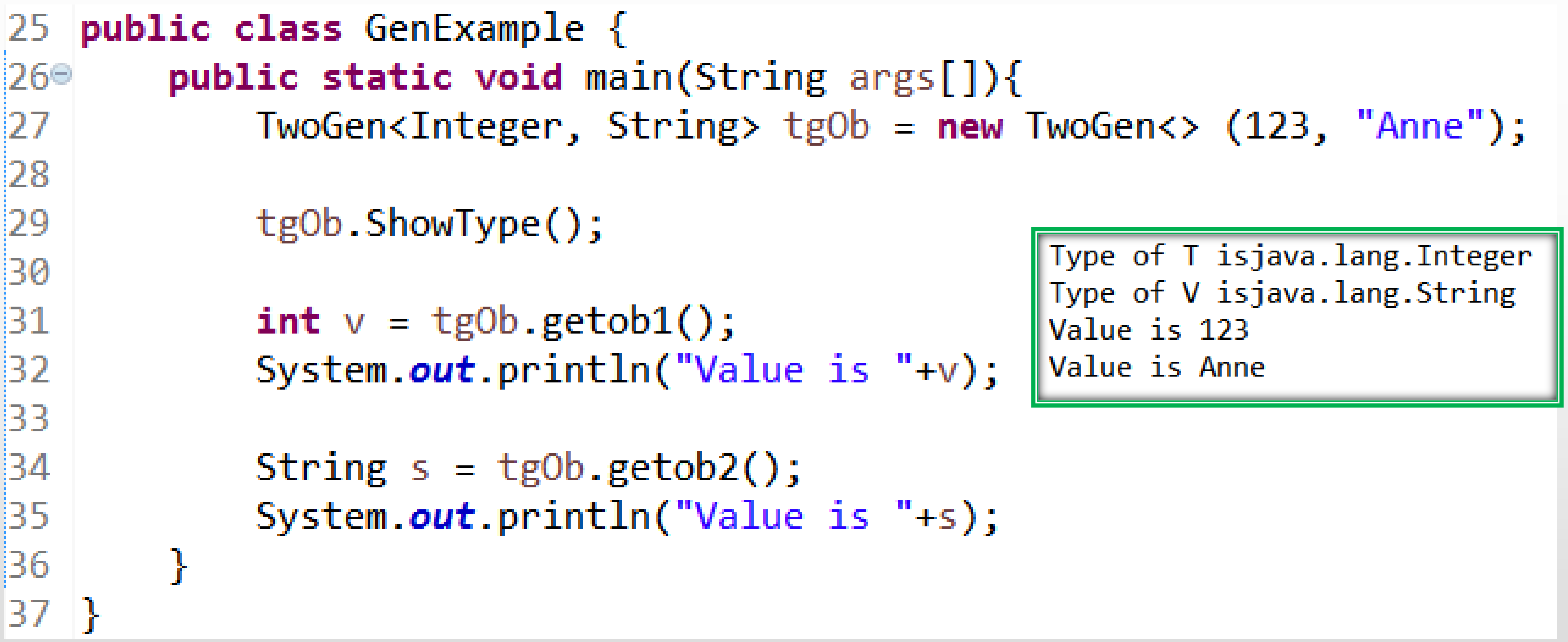


```java
class TwoGen<T,V>{
    T ob1;
    V ob2;

    TwoGen(T o1, V o2){
        ob1 = o1;
        ob2 = o2;
    }

    void ShowType() {
        System.out.println("Type of T is"+ob1.getClass().getName());
        System.out.println("Type of V is"+ob2.getClass().getName());
    }

    T getob1() {
        return ob1;
    }

    V getob2() {
        return ob2;
    }
}
```

test2.java

Multiple Parameters cont.

- This is how you will make use of the generic class you created:



```java
public class GenExample {
    public static void main(String args[]){
        TwoGen<Integer, String> tgOb = new TwoGen<> (123, "Anne");

        tgOb.ShowType();

        int v = tgOb.getob1();
        System.out.println("Value is "+v);

        String s = tgOb.getob2();
        System.out.println("Value is "+s);
    }
}
```

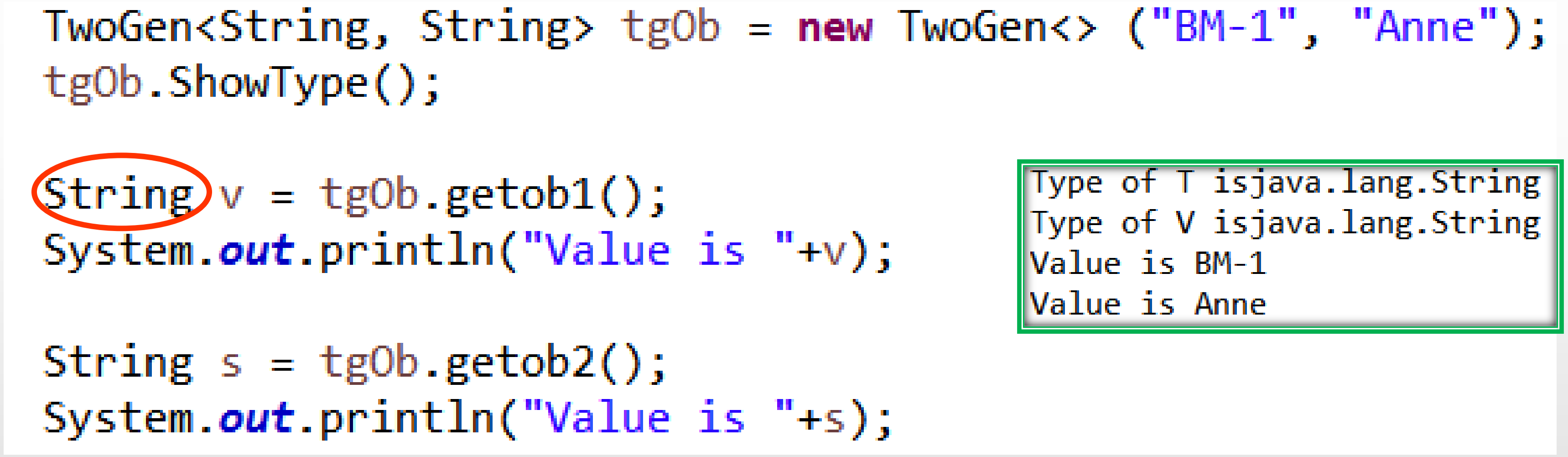
```
Type of T isjava.lang.Integer
Type of V isjava.lang.String
Value is 123
Value is Anne
```

test2.java

Multiple Parameters cont.

- Although the two type argument differ in the example, it is possible for both types to be same.

e.g.:



```
TwoGen<String, String> tgOb = new TwoGen<> ("BM-1", "Anne");
tgOb.ShowType();

String v = tgOb.getob1();
System.out.println("Value is "+v);

String s = tgOb.getob2();
System.out.println("Value is "+s);
```

```
Type of T isjava.lang.String
Type of V isjava.lang.String
Value is BM-1
Value is Anne
```

- Also note, if both type arguments were always the same, then two type parameters would be unnecessary.

Bounded Type Parameters

- There may be times when you will want to *restrict* the kind of data types that are allowed to be passed to a type parameter.

  e.g.: A method that operates on numbers might only want to accept instances of Number or its subclasses.

- To declare a bounded type parameter, list the type parameter's name, followed by the extends keyword, followed by its upper bound in the following format.

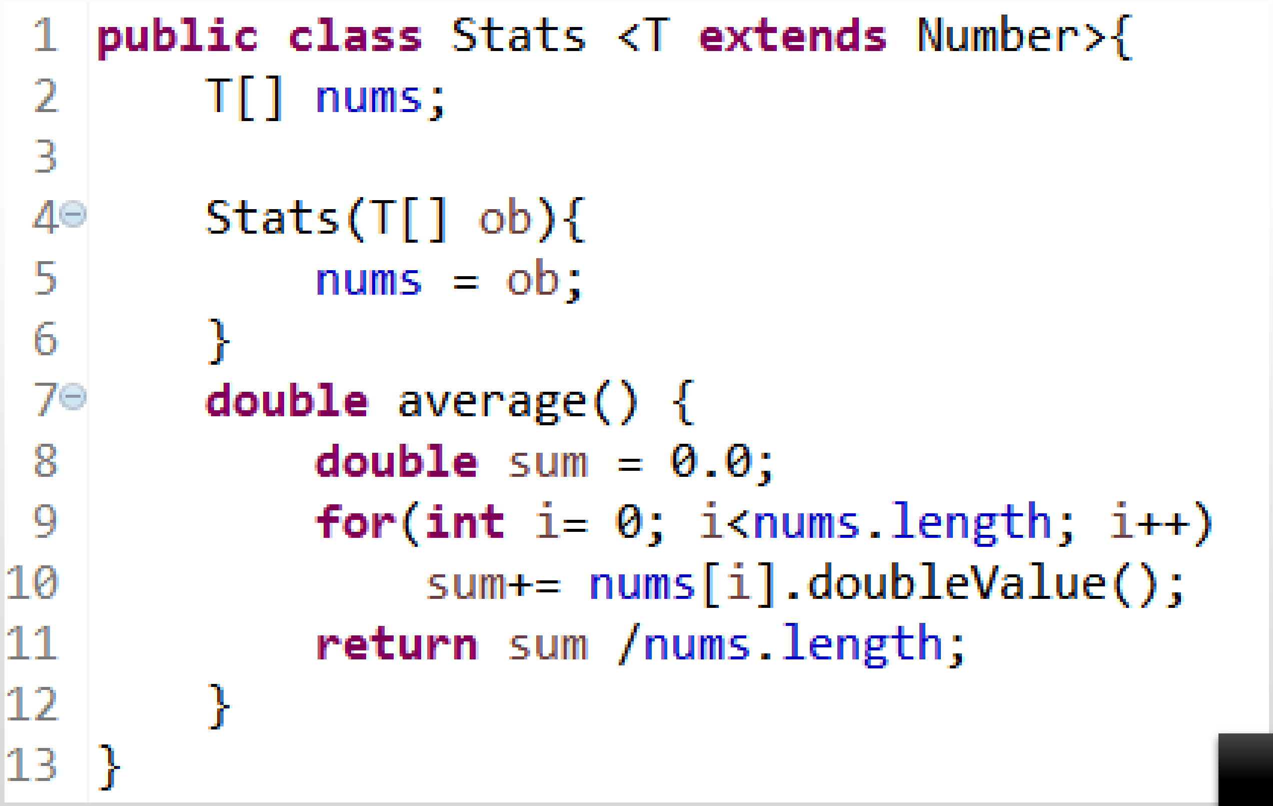
  <T extends superclass>

  e.g.: <T extends Number>

- Note: T, can only be replaced by the superclass or its subclasses.

Bounded Type Parameters cont.

- This is an example of the class with a bounded type parameter:

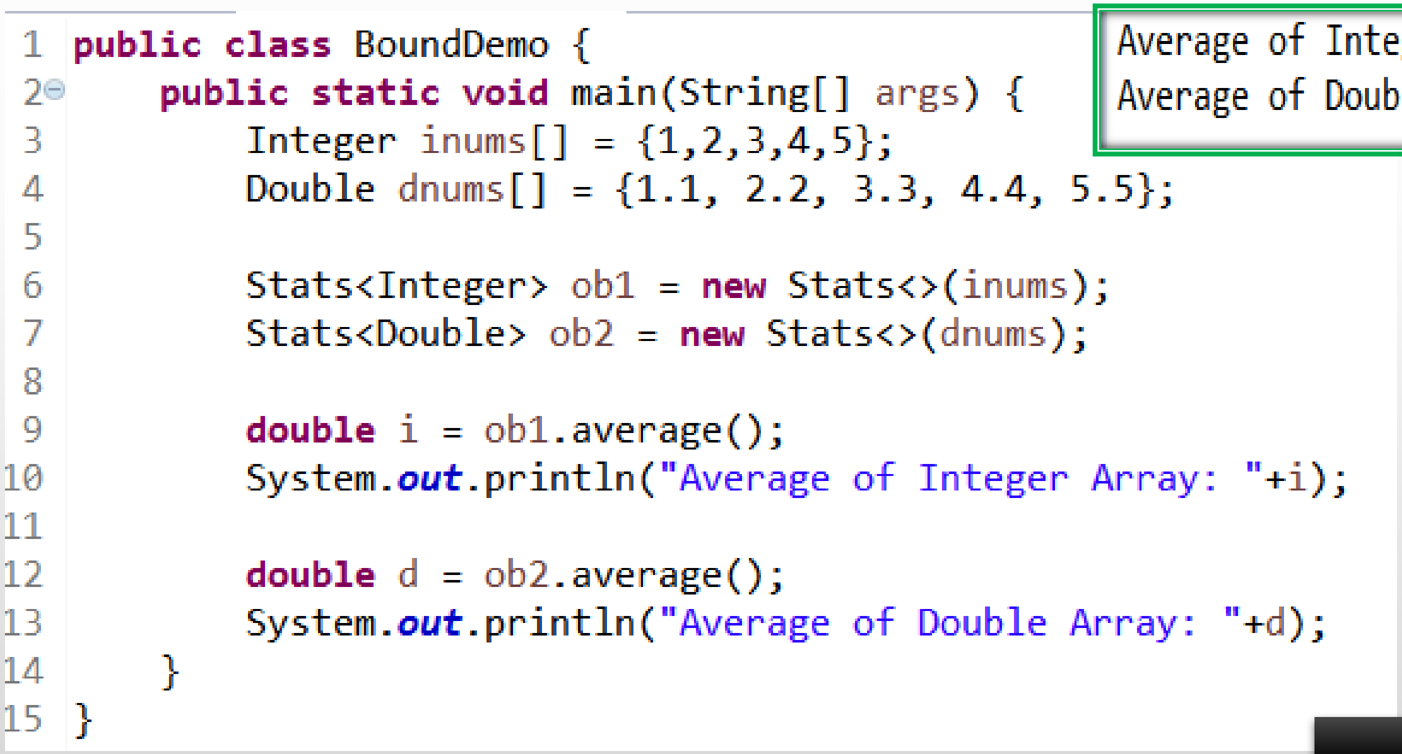


```java
public class Stats <T extends Number>{
    T[] nums;

    Stats(T[] ob){
        nums = ob;
    }
    double average() {
        double sum = 0.0;
        for(int i= 0; i<nums.length; i++)
            sum+= nums[i].doubleValue();
        return sum /nums.length;
    }
}
```

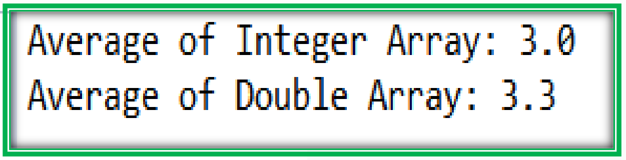
test3.java

Bounded Type Parameters cont.

- This is how you will make use of the class with a bounded type parameter.



```java
public class BoundDemo {
    public static void main(String[] args) {
        Integer inums[] = {1,2,3,4,5};
        Double dnums[] = {1.1, 2.2, 3.3, 4.4, 5.5};

        Stats<Integer> ob1 = new Stats<>(inums);
        Stats<Double> ob2 = new Stats<>(dnums);

        double i = ob1.average();
        System.out.println("Average of Integer Array: "+i);

        double d = ob2.average();
        System.out.println("Average of Double Array: "+d);
    }
}
```

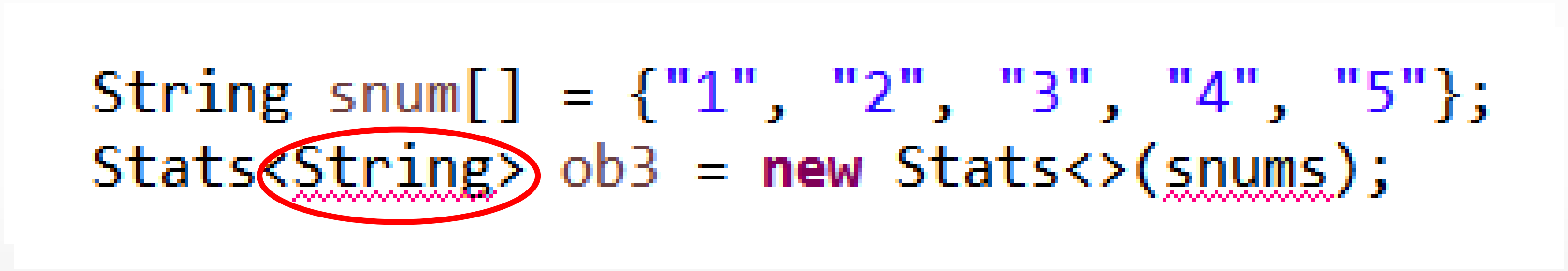


```
Average of Integer Array: 3.0
Average of Double Array: 3.3
```

test3.java

Bounded Type Parameters cont.

- Note that the following code segment will give compilation error as String is not a subclass of Number

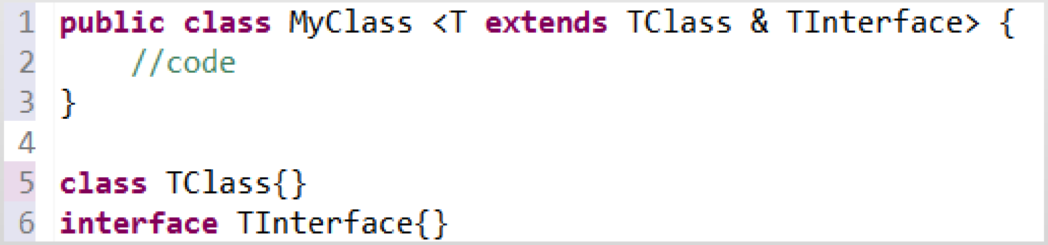


```
String snum[] = {"1", "2", "3", "4", "5"};
Stats<String> ob3 = new Stats<>(snums);
```

Bounded Type Parameters cont.

- In addition to using a class type as a bound, you can also use an interface type too! In fact, you can specify multiple interfaces as bounds.
- A bound can include a class type & one or more interfaces. In this case, the class type needs to be specified first.
- When a bound include an interface type, only type arguments that implement that interface are legal.
- When specifying a bound that has a class & an interface or multiple interfaces, use the & operator to connect them

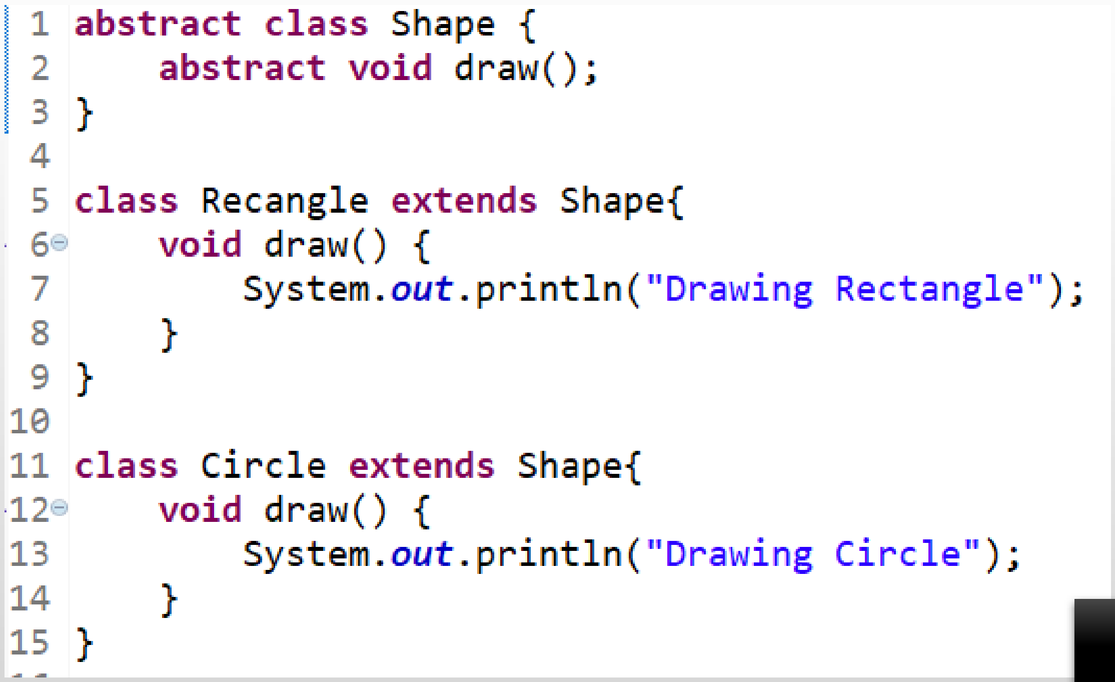
e.g:



```java
public class MyClass <T extends TClass & TInterface> {
    //code
}

class TClass{}
interface TInterface{}
```

Wildcard in Java Generics

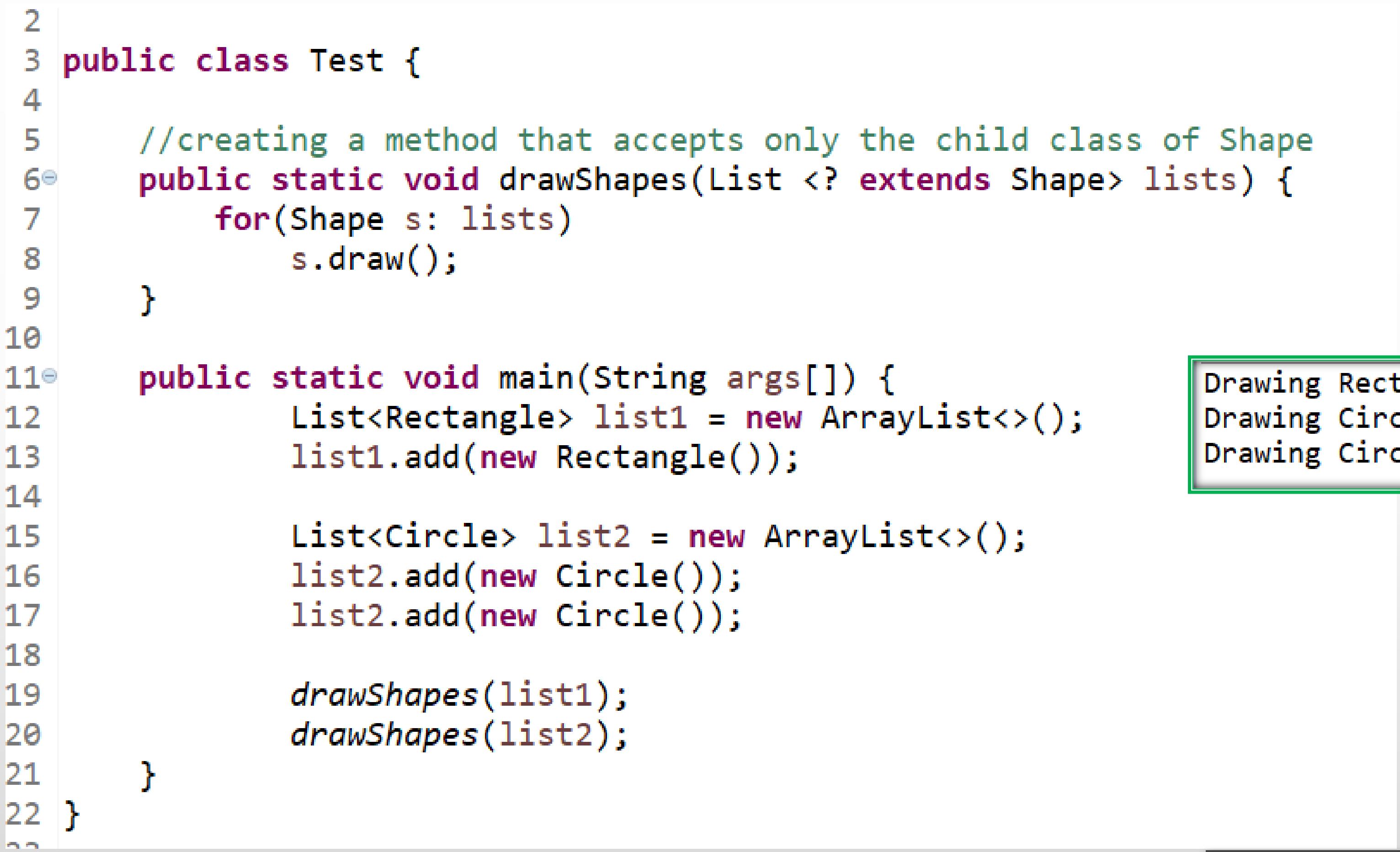
- The ? Symbol represents the wildcard element. It means any type can be matched with the ? symbol.
- If we write <? extends Number>, it means any child class of Number (e.g. Integer, Double, Float etc.) can be matched with the ?



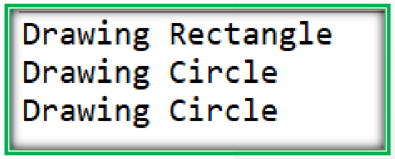
```java
abstract class Shape {
    abstract void draw();
}

class Recangle extends Shape{
    void draw() {
        System.out.println("Drawing Rectangle");
    }
}

class Circle extends Shape{
    void draw() {
        System.out.println("Drawing Circle");
    }
}
```

test4.java

Wildcard in Java Generics cont.



```java

public class Test {

    //creating a method that accepts only the child class of Shape
    public static void drawShapes(List <? extends Shape> lists) {
        for(Shape s: lists)
            s.draw();
    }

    public static void main(String args[]) {
            List<Rectangle> list1 = new ArrayList<>();
            list1.add(new Rectangle());

            List<Circle> list2 = new ArrayList<>();
            list2.add(new Circle());
            list2.add(new Circle());

            drawShapes(list1);
            drawShapes(list2);
    }
}
```



```
Drawing Rectangle
Drawing Circle
Drawing Circle
```

test4.java

Limitations in Generics

- **Type parameters cannot be instantiated -** It is not possible to create instances of a type parameter
- **Restriction on static members -** No static member can use a type parameter declared by the enclosing class. Note that you can declare static generic methods.
- **Generic array restriction -**
  - Cannot instantiate an array whose element type is a type parameter.
  - Cannot create an array of type specific references
- **Generic exception restriction -** A generic class cannot extend Throwable. This means we cannot create generic exception classes.

Thank you!