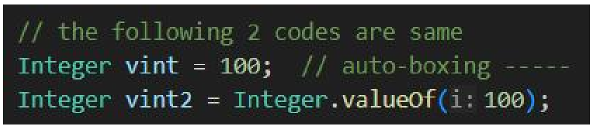
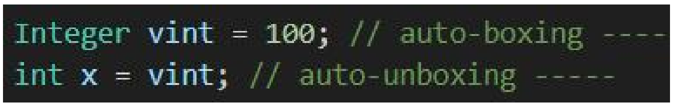
- **Auto-Boxing**

  - 

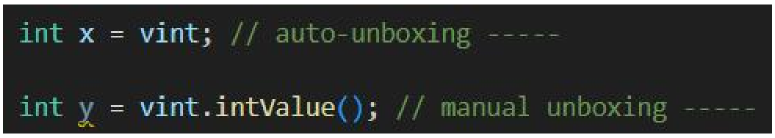
    ```
    // the following 2 codes are same
    Integer vint = 100;  // auto-boxing -----
    Integer vint2 = Integer.valueOf(i: 100);
    ```

  - It is auto-boxing,
  - When we assign any **primitive** value to **wrapper class** (just like here assigning int value (100) to wrapper (Integer)). under the hood it is getting converted using that **valueOf** method.
  - **Integer.valueOf()** method takes **int** value as an argument and return one **Integer** type of object in return.

- **Auto-unboxing**

  - 

    ```
    Integer vint = 100; // auto-boxing ----
    int x = vint; // auto-unboxing -----
    ```

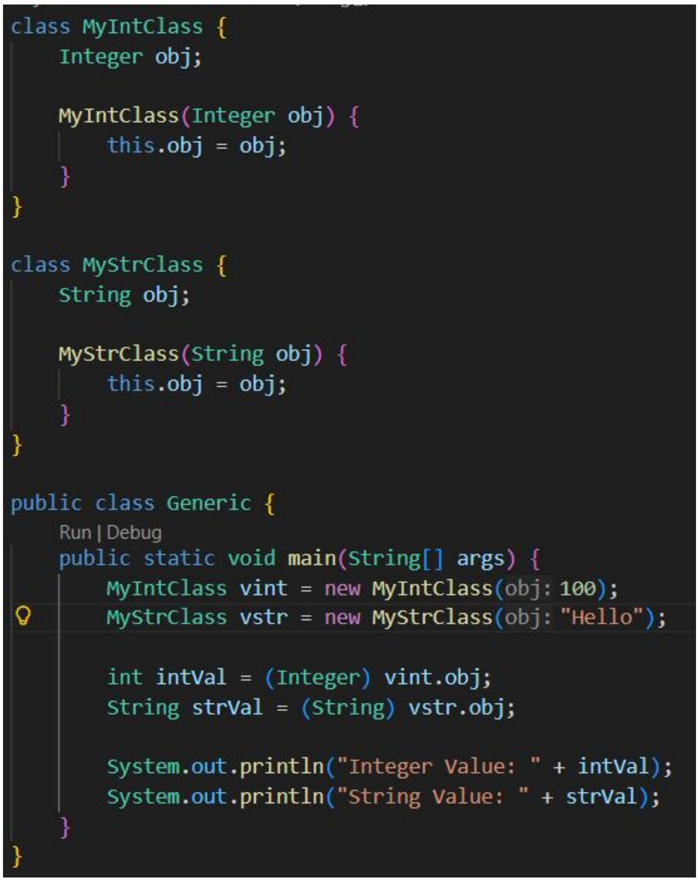
  - It is auto-unboxing.
  - Here, the wrapper type (**Integer**) is getting converted to primitive type (**int**).
  - 

    ```
    int x = vint; // auto-unboxing -----

    int y = vint.intValue(); // manual unboxing -----
    ```

  - This is what happens under the hood. **intValue** method returns the primitive i.e. **int** value of the wrapper i.e. **Integer**.

GENERIC

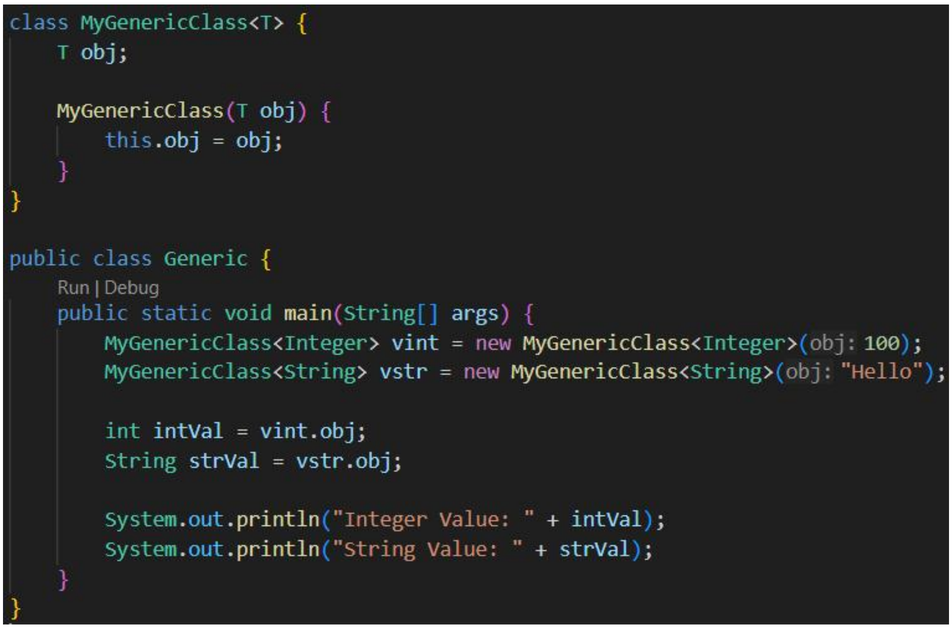
- The below is one non-generic class.



```
class MyIntClass {
    Integer obj;

    MyIntClass(Integer obj) {
        this.obj = obj;
    }
}

class MyStrClass {
    String obj;

    MyStrClass(String obj) {
        this.obj = obj;
    }
}

public class Generic {
    Run | Debug
    public static void main(String[] args) {
        MyIntClass vint = new MyIntClass(obj: 100);
        MyStrClass vstr = new MyStrClass(obj: "Hello");

        int intVal = (Integer) vint.obj;
        String strVal = (String) vstr.obj;

        System.out.println("Integer Value: " + intVal);
        System.out.println("String Value: " + strVal);
    }
}
```

In here, for the same operations, we need to create different different classes for different types of data types. To overcome this issue, we can use **Generic classes**.

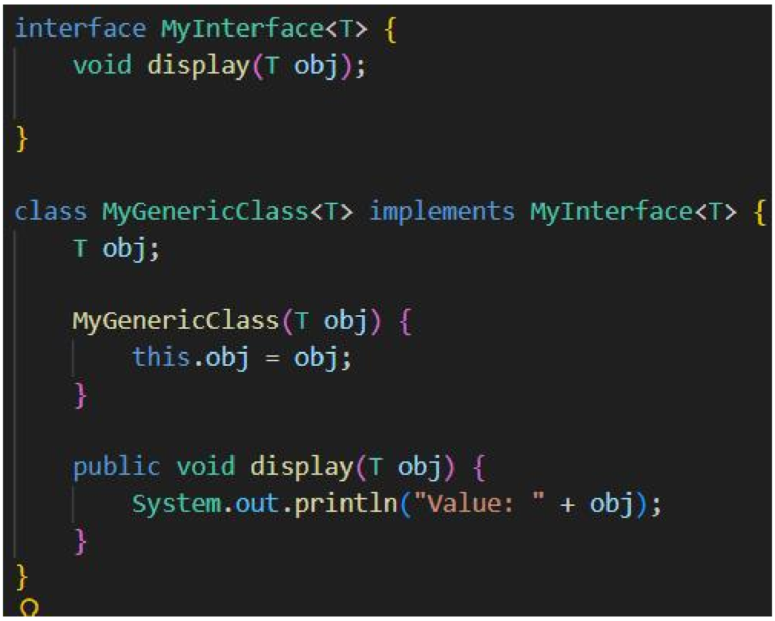
- The below is the example with **Generic** class.



```
class MyGenericClass<T> {
    T obj;

    MyGenericClass(T obj) {
        this.obj = obj;
    }
}

public class Generic {
    Run | Debug
    public static void main(String[] args) {
        MyGenericClass<Integer> vint = new MyGenericClass<Integer>(obj: 100);
        MyGenericClass<String> vstr = new MyGenericClass<String>(obj: "Hello");

        int intVal = vint.obj;
        String strVal = vstr.obj;

        System.out.println("Integer Value: " + intVal);
        System.out.println("String Value: " + strVal);
    }
}
```

  - **<T>** denotes the **Object** type.
  - **Generic** type cannot take **primitive** type. It only takes the **Objects** (i.e. **wrappers**).

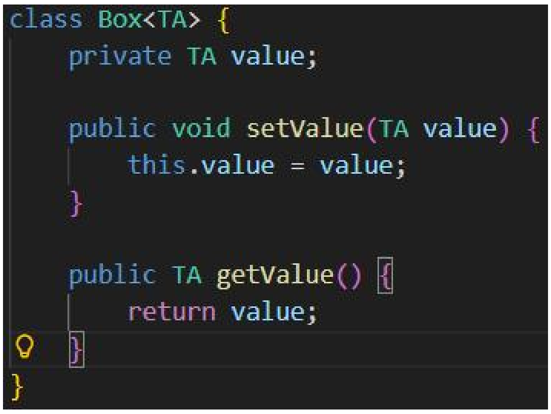
- In the below example you can see if one class/interface is implementing/inheriting another **interface** or **class**, then it should include the class for the generic class.

  - 

```
interface MyInterface<T> {
    void display(T obj);

}

class MyGenericClass<T> implements MyInterface<T> {
    T obj;

    MyGenericClass(T obj) {
        this.obj = obj;
    }

    public void display(T obj) {
        System.out.println("Value: " + obj);
    }
}
```

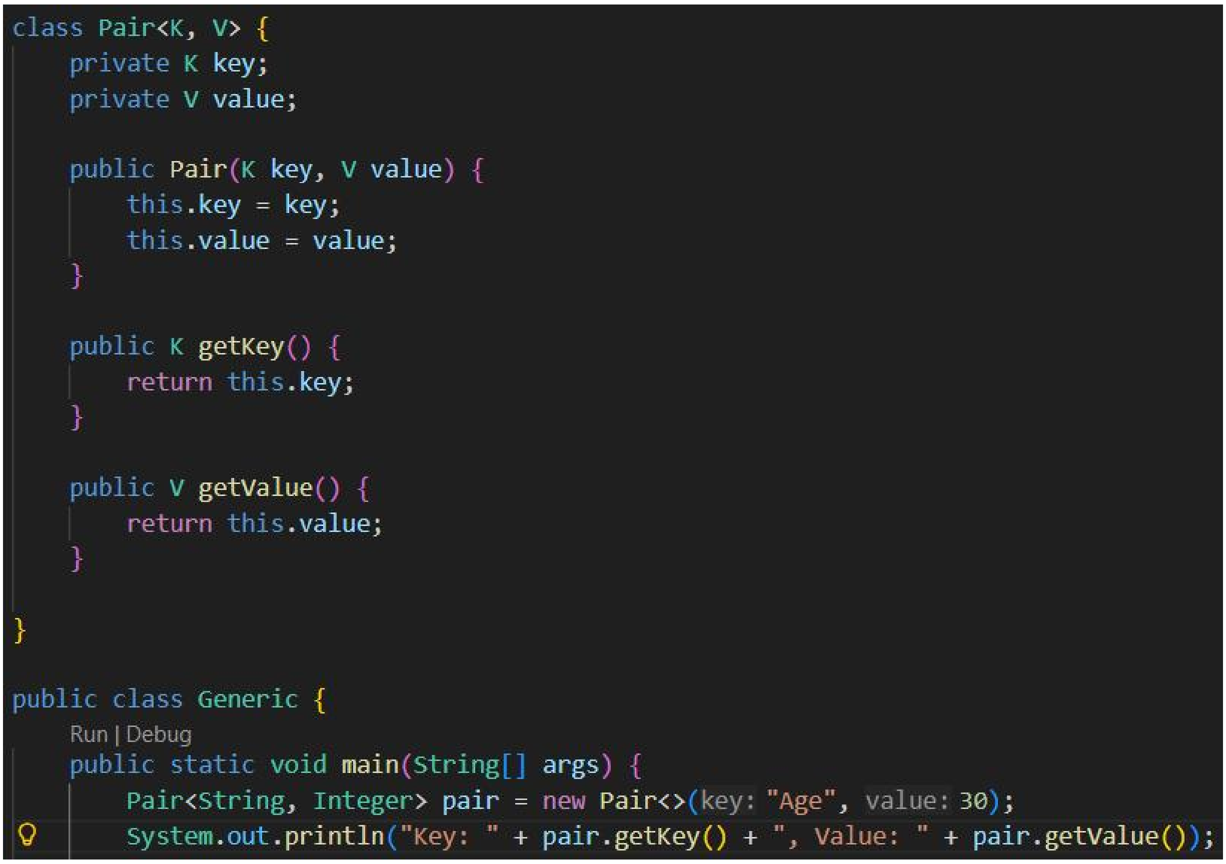
  - MyInterface<T> is mentioned with the MyGenericClass<T>

- You can give multi-letter generic type as well. But the convention is to use single letter.

  - 

```
class Box<TA> {
    private TA value;

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

    public TA getValue() {
        return value;
    }
}
```

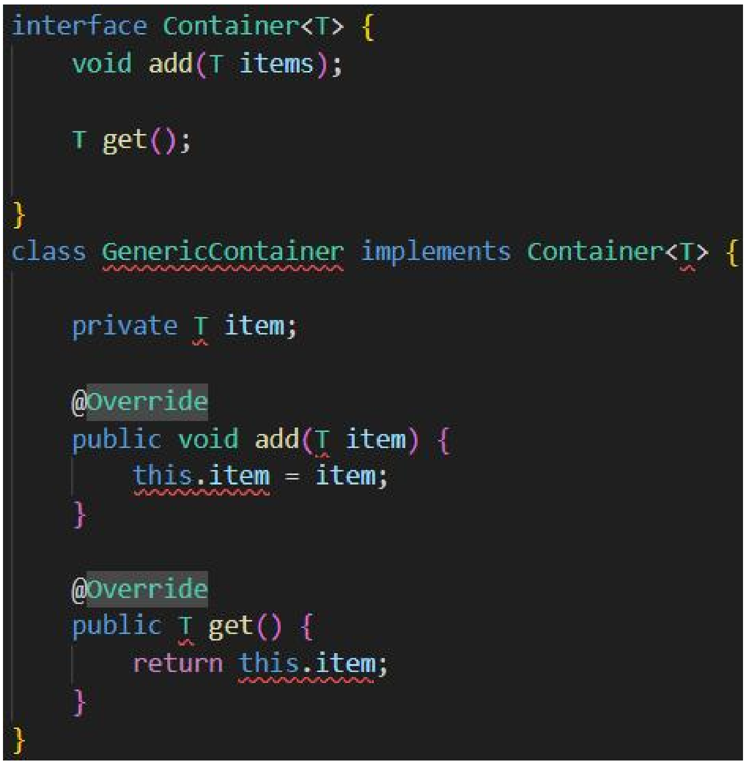
- Multiple types can also be used.



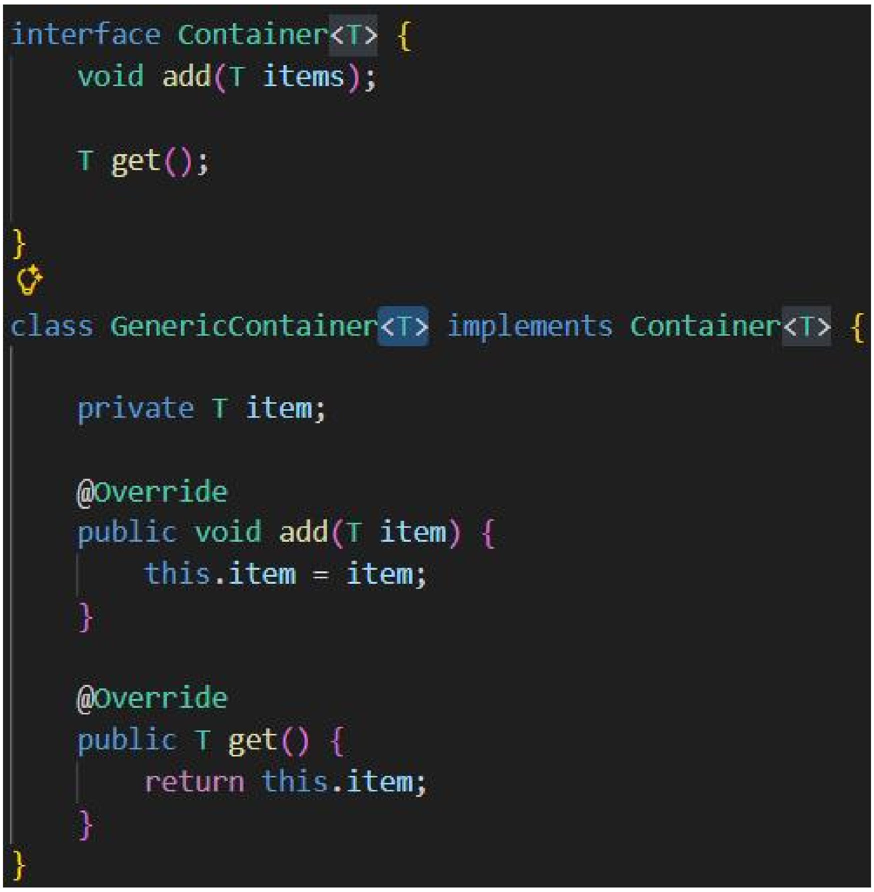
```java
class Pair<K, V> {
    private K key;
    private V value;

    public Pair(K key, V value) {
        this.key = key;
        this.value = value;
    }

    public K getKey() {
        return this.key;
    }

    public V getValue() {
        return this.value;
    }

}

public class Generic {
    Run | Debug
    public static void main(String[] args) {
        Pair<String, Integer> pair = new Pair<>(key: "Age", value: 30);
        System.out.println("Key: " + pair.getKey() + ", Value: " + pair.getValue());
```

- Conventions:
    - **T** : Type
    - **K, V** : Key and Value in case of maps
    - **E** : Element (Used in Collections)
    - **N** : Number

- In the below example, I used **T** in the implemented interface and in the methods as well. But didn't defined it; So, it'll give me error.



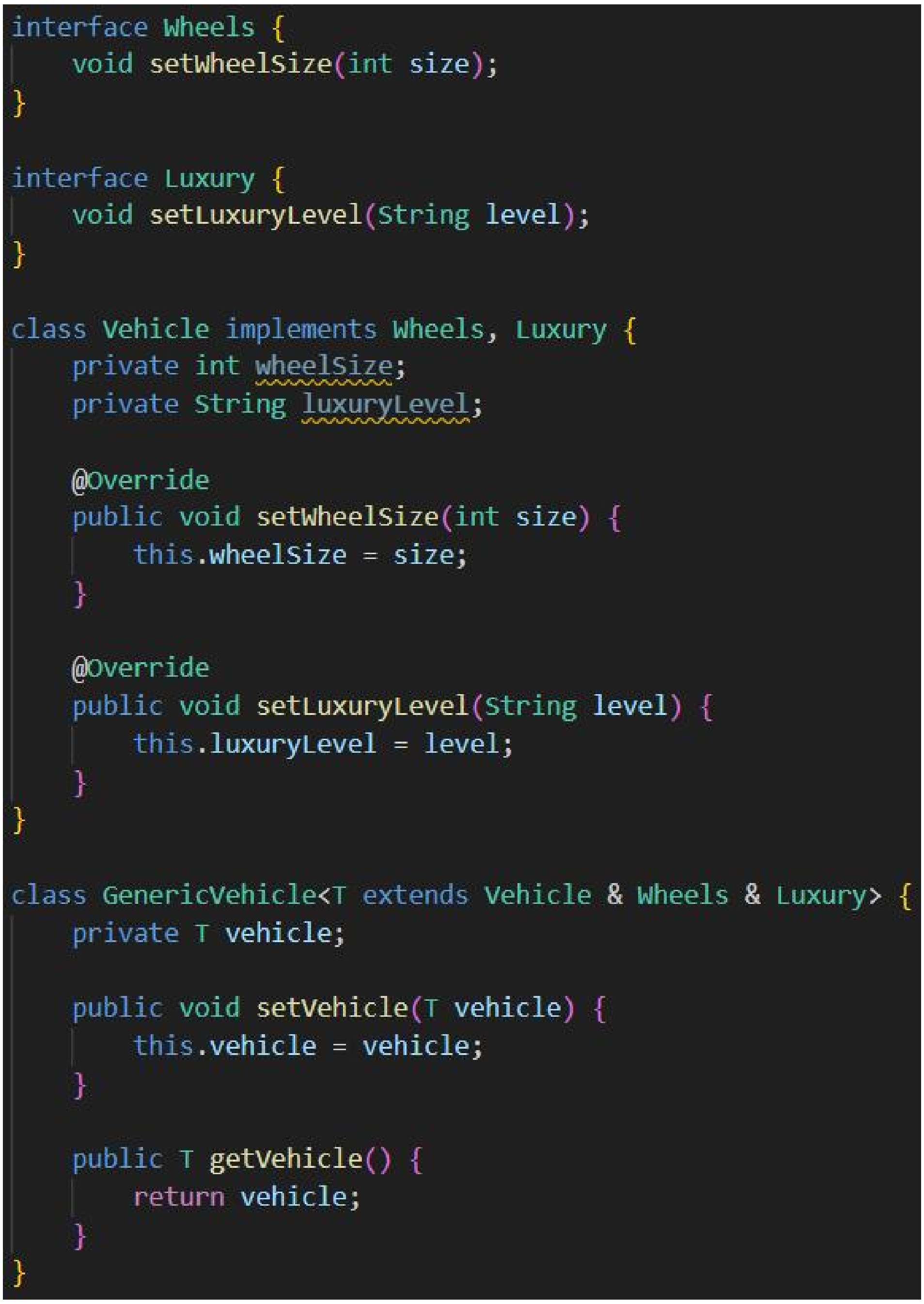
```
interface Container<T> {
    void add(T items);

    T get();

}
class GenericContainer implements Container<T> {

    private T item;

    @Override
    public void add(T item) {
        this.item = item;
    }

    @Override
    public T get() {
        return this.item;
    }
}
```



```
interface Container<T> {
    void add(T items);

    T get();

}

class GenericContainer<T> implements Container<T> {

    private T item;

    @Override
    public void add(T item) {
        this.item = item;
    }

    @Override
    public T get() {
        return this.item;
    }
}
```

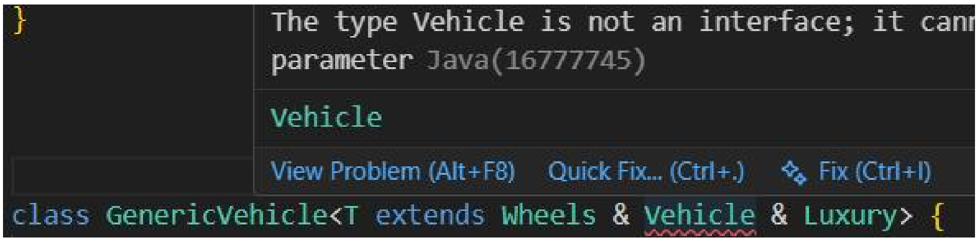
- Now it is fixed.

- **Bounded type**
  - It means the **object type** should be such that it will be inheriting one **Class or/and** implementing **one/more** interface(s).



```java
interface Wheels {
    void setWheelSize(int size);
}

interface Luxury {
    void setLuxuryLevel(String level);
}

class Vehicle implements Wheels, Luxury {
    private int wheelSize;
    private String luxuryLevel;

    @Override
    public void setWheelSize(int size) {
        this.wheelSize = size;
    }

    @Override
    public void setLuxuryLevel(String level) {
        this.luxuryLevel = level;
    }
}

class GenericVehicle<T extends Vehicle & Wheels & Luxury> {
    private T vehicle;

    public void setVehicle(T vehicle) {
        this.vehicle = vehicle;
    }

    public T getVehicle() {
        return vehicle;
    }
}
```

  - **T** extends **class & interface1 & interface2**
  - If you only want **T** to implement only interfaces then:
    - **T** extends **interface1 & interface2** (NOTE: **extends** keyword will be used)
  - The **class** should be written first (if you want **T** to extend some class); If you write **class** in 2nd or 3rd or… after places, it'll assume that that is **interface** (not class)



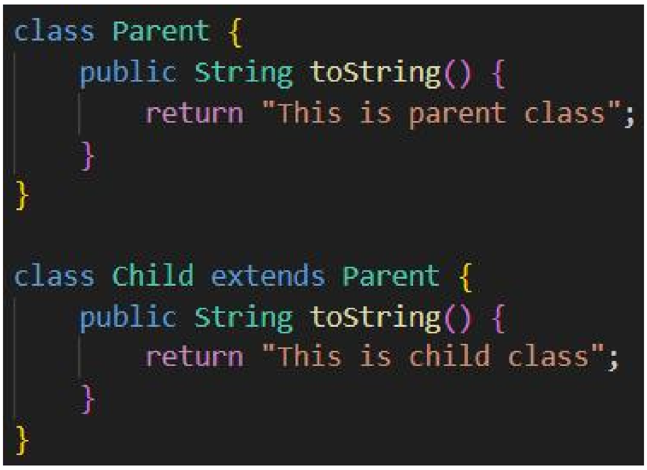
```
}
```

The type Vehicle is not an interface; it cann
parameter Java(16777745)

Vehicle

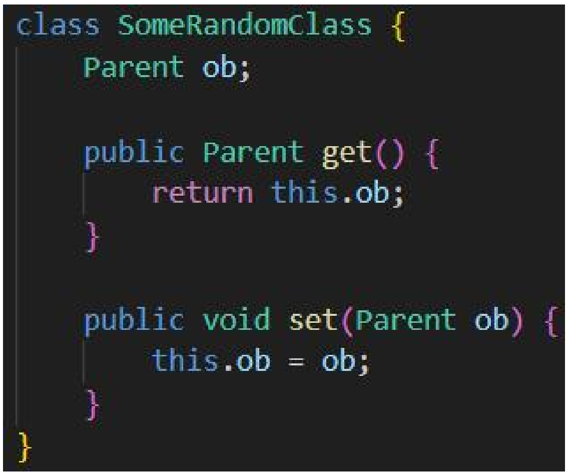
```
class GenericVehicle<T extends Wheels & Vehicle & Luxury> {
```

- First one can be **class** or **interface** doesn't matter; but after **first,** it must be interfaces.
  - The reason is, in java, multiple inheritance is not supported; extending multiple classes won't be allowed; so if you want, then write the class in the first place then go ahead with the interfaces.
- Here you can say why to write like this **T extends SomeClass**, we can directly write **SomeClass** and we can use **SomeClassB** instead of that **T;** but what is the problem?
  - Lets consider the below example:



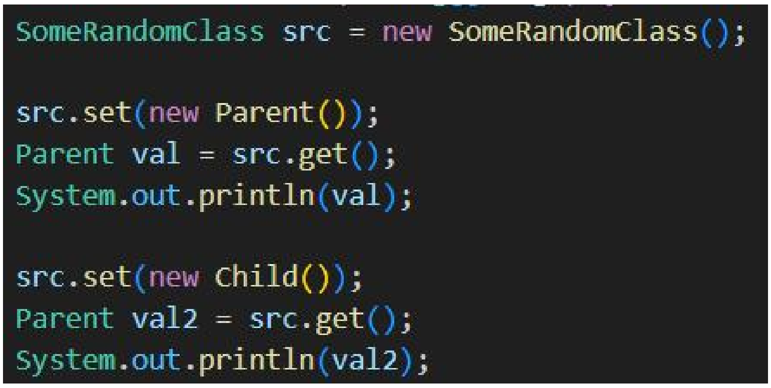
```
class Parent {
    public String toString() {
        return "This is parent class";
    }
}

class Child extends Parent {
    public String toString() {
        return "This is child class";
    }
}
```

- I have 2 classes; **Parent** and **Child**. **Child** extens **Parent** class.



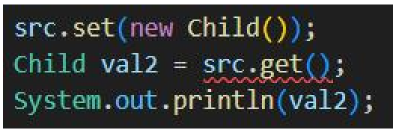
```
class SomeRandomClass {
    Parent ob;

    public Parent get() {
        return this.ob;
    }

    public void set(Parent ob) {
        this.ob = ob;
    }
}
```

- This class accepts **Parent** type values; so we can also use **Child** values here.



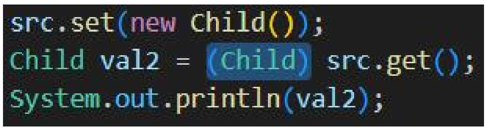
```
SomeRandomClass src = new SomeRandomClass();

src.set(new Parent());
Parent val = src.get();
System.out.println(val);

src.set(new Child());
Parent val2 = src.get();
System.out.println(val2);
```

- Here it is perfectly fine because we are keeping the return value of **get** method in a variable of type **Parent.**
- And also, in the class **SomeRandomClass**, the variable is of type **Parent** so it can store **Child** type objects without doing **upcasting** explicitely.
- NOTE: **upcasting** happens by default; but for **downcasting**, you need to explicitely mention.



```
src.set(new Child());
Child val2 = src.get();
System.out.println(val2);
```

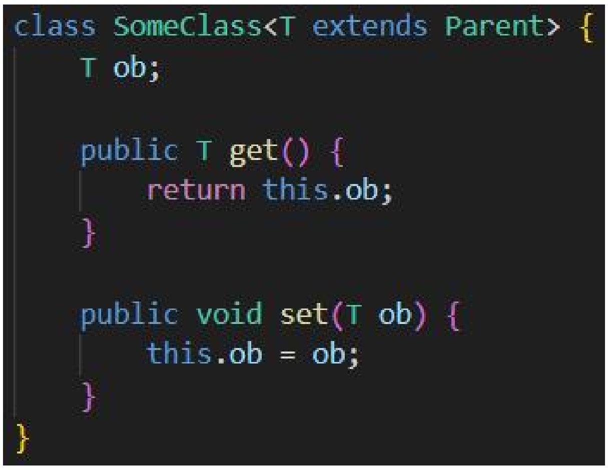
- Now it'll fail, because **get** method is returning one **Parent** type object and we are creating variable of **Child** type.



```
src.set(new Child());
Child val2 = (Child) src.get();
System.out.println(val2);
```

- Now it'll work properly because we did **downcasting** here.

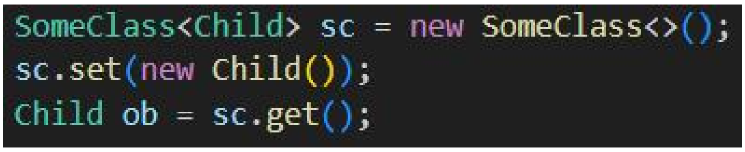
Now, the problem is, why to use this **downcasting** and **upcasting** ; its not generic anymore.



```
class SomeClass<T extends Parent> {
    T ob;

    public T get() {
        return this.ob;
    }

    public void set(T ob) {
        this.ob = ob;
    }
}
```

- I have one more class here which is using **generic** type **T** which needs to inherit the **Parent** class.

- One more thing, **T extends Parent** means, **T** can be **Parent** itself or any **subclass of Parent**.

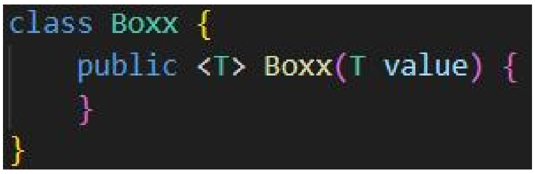
- 

```
SomeClass<Child> sc = new SomeClass<>();
sc.set(new Child());
Child ob = sc.get();
```

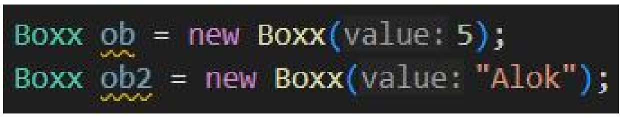
- Now it'll perfectly work.

Generic Constructor

- If we don't want to explicitely write any generic type **T** for the whole class, but just want to use the generic for the constructor then we can do this like below:

  - 

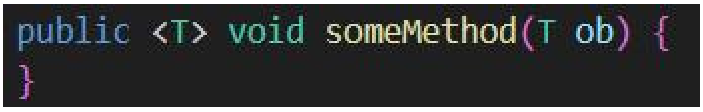
```java
class Boxx {
    public <T> Boxx(T value) {
    }
}
```

  - 

```java
Boxx ob = new Boxx(value: 5);
Boxx ob2 = new Boxx(value: "Alok");
```

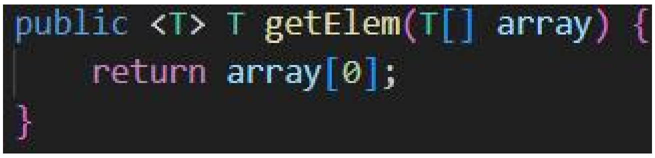
    - You can use like this.

- **Generic Method**

  - 

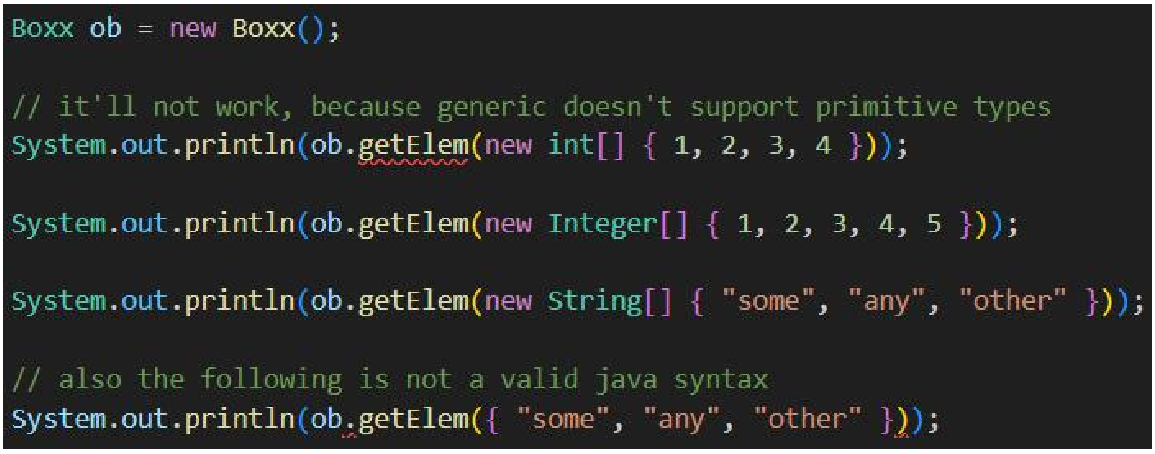
```java
public <T> void someMethod(T ob) {
}
```

    - Return type is: **void**

  - 

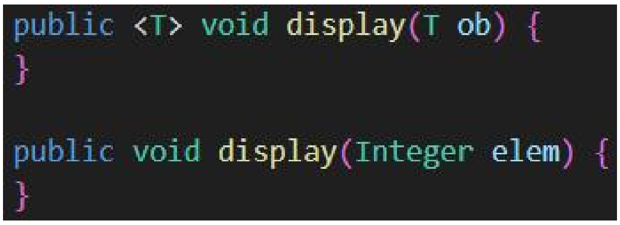
```java
public <T> T getElem(T[] array) {
    return array[0];
}
```

    - Return type: **T**
    - Here I am getting one array of type **T**, and returning the first value of the array which is ofcourse of type **T** only.

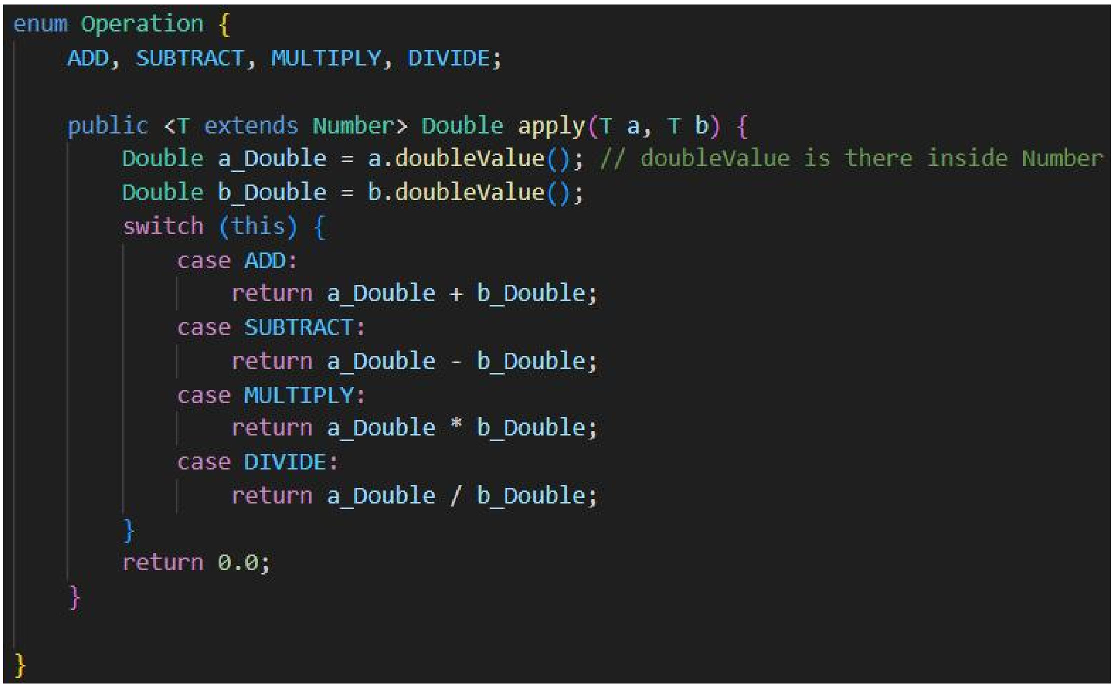
  - 

```java
Boxx ob = new Boxx();

// it'll not work, because generic doesn't support primitive types
System.out.println(ob.getElem(new int[] { 1, 2, 3, 4 }));

System.out.println(ob.getElem(new Integer[] { 1, 2, 3, 4, 5 }));

System.out.println(ob.getElem(new String[] { "some", "any", "other" }));

// also the following is not a valid java syntax
System.out.println(ob.getElem({ "some", "any", "other" }));
```

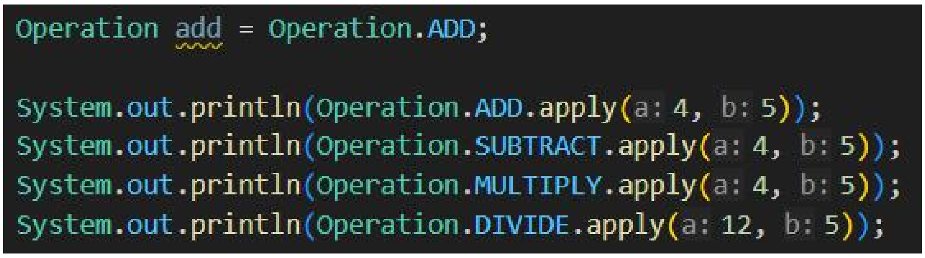
  - The first one is giving error because **int[]** is a primitive type array and generic doesn't support primitive.
  - The last one is not valid syntax because **Java never allows a raw array initializer as an expression**.

- **{1,2,3}** or **{“some”, “any”, “other”}** is **NOT an object, NOT an array, NOT an expression**. It is ONLY a **shorthand** allowed in one specific place: **variable initialization. {..}.**
- In simple terms, **{1,2,3}** or **{“some”, “other”}** ..etc are **shorthand** for **new int[] {1,2,3}** or **new String[] {“some”, “other”}** only when it is being initialized to a variable. Apart from this, it is not valid anywhere else.
- 

```java
public <T> void display(T ob) {
}

public void display(Integer elem) {
}
```

    - It is method overloading. If **Integer** is passed then the 2nd method will be called. Otherwise the first method will be called.
- ---
- 

```java
enum Operation {
    ADD, SUBTRACT, MULTIPLY, DIVIDE;

    public <T extends Number> Double apply(T a, T b) {
        Double a_Double = a.doubleValue(); // doubleValue is there inside Number
        Double b_Double = b.doubleValue();
        switch (this) {
            case ADD:
                return a_Double + b_Double;
            case SUBTRACT:
                return a_Double - b_Double;
            case MULTIPLY:
                return a_Double * b_Double;
            case DIVIDE:
                return a_Double / b_Double;
        }
        return 0.0;
    }

}
```

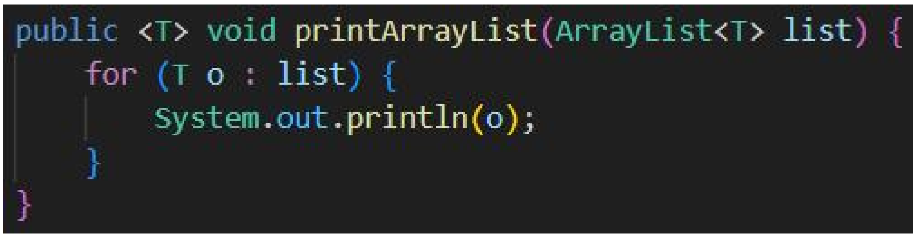
    - I wrote one **generic method** inside **enum** here. **[switch (this)]**
- 

```java
Operation add = Operation.ADD;

System.out.println(Operation.ADD.apply(a: 4, b: 5));
System.out.println(Operation.SUBTRACT.apply(a: 4, b: 5));
System.out.println(Operation.MULTIPLY.apply(a: 4, b: 5));
System.out.println(Operation.DIVIDE.apply(a: 12, b: 5));
```

    - Now we can use like this.

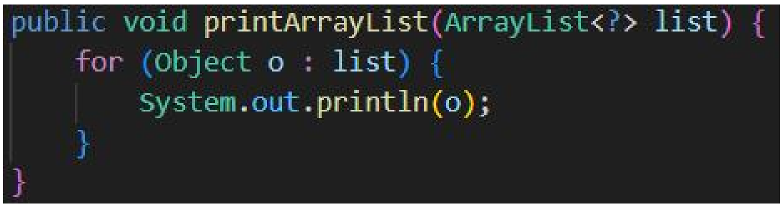
Wildcards in Generics

- Wildcards (**?**) is used in method arguments or class definitions to represent an **unknown** type.
  - The type can be specified later or be more loosely defined.
  - If you are just doing **read** operation and **not returning** anything then **wildcards** is useful.
- In the below example, you can see, we are only reading the values.



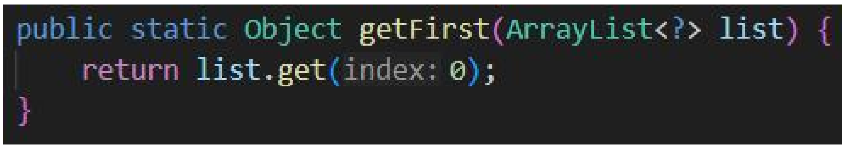
```
public <T> void printArrayList(ArrayList<T> list) {
    for (T o : list) {
        System.out.println(o);
    }
}
```

- We can write like the below:



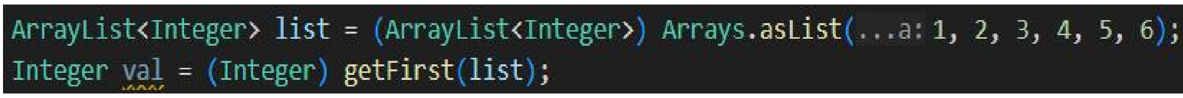
```
public void printArrayList(ArrayList<?> list) {
    for (Object o : list) {
        System.out.println(o);
    }
}
```

  - No need to explicitly mention **<?>** before **void** just like the generic type **T**.
- In the below example, we are returning something.



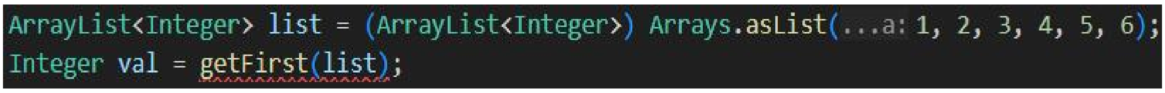
```
public static Object getFirst(ArrayList<?> list) {
    return list.get(index: 0);
}
```

  - Here, Object is the parent class of every class, so it'll not give any error in the method.
  - But, we need to explicitly **downcast** in the function call step.



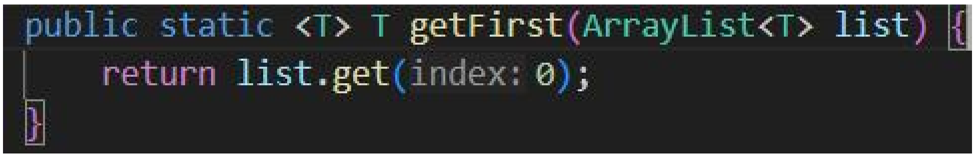
```
ArrayList<Integer> list = (ArrayList<Integer>) Arrays.asList(...a: 1, 2, 3, 4, 5, 6);
Integer val = (Integer) getFirst(list);
```

  - If we don't downcast that, it'll give error.

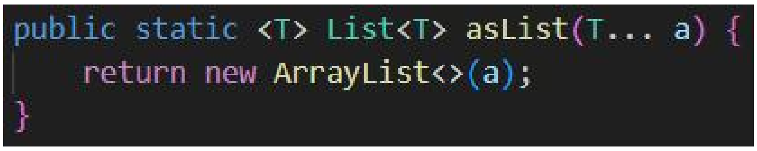


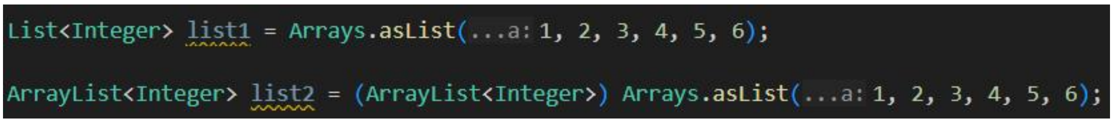
```
ArrayList<Integer> list = (ArrayList<Integer>) Arrays.asList(...a: 1, 2, 3, 4, 5, 6);
Integer val = getFirst(list);
```

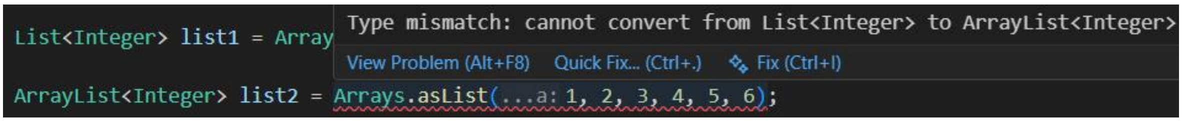
- So, the method should be using **generic** only.

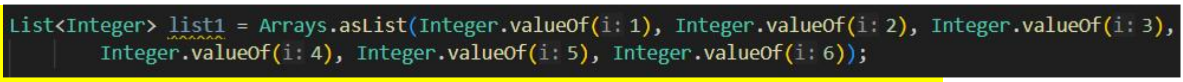


```
public static <T> T getFirst(ArrayList<T> list) {
    return list.get(index: 0);
}
```

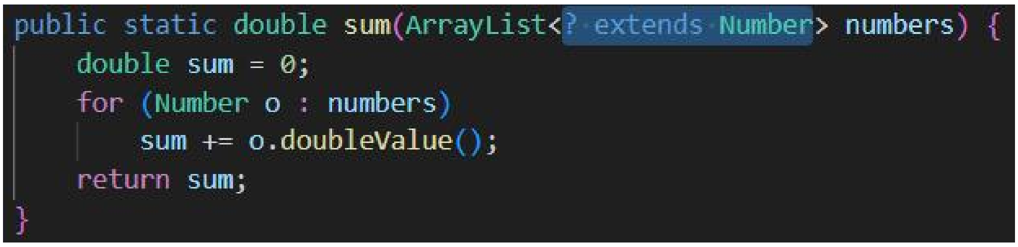
- **NOTE**
    - **Arrays.asList()** returns **List** type.
    - 

      ```
      public static <T> List<T> asList(T... a) {
          return new ArrayList<>(a);
      }
      ```
    - As the return type of **List<T>**, and its returning **ArrayList<T>** so its basically doing **upcasting** (which is not needed to be explicitly mentioned).
    - So, there can be 2 possibilities:
        - 

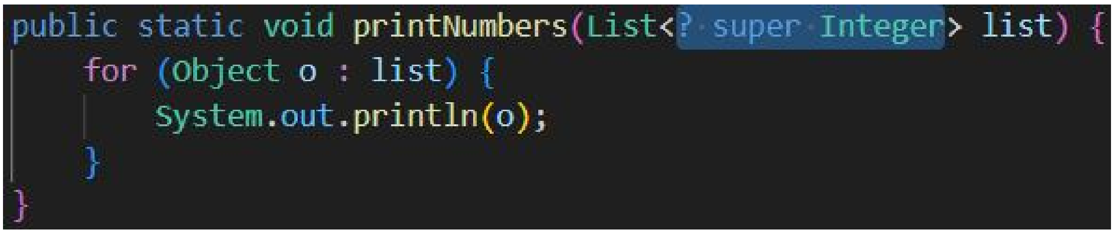
          ```
          List<Integer> list1 = Arrays.asList(...a: 1, 2, 3, 4, 5, 6);

          ArrayList<Integer> list2 = (ArrayList<Integer>) Arrays.asList(...a: 1, 2, 3, 4, 5, 6);
          ```
        - If you don't **downcast** it then it'll give error just like below.
        - 

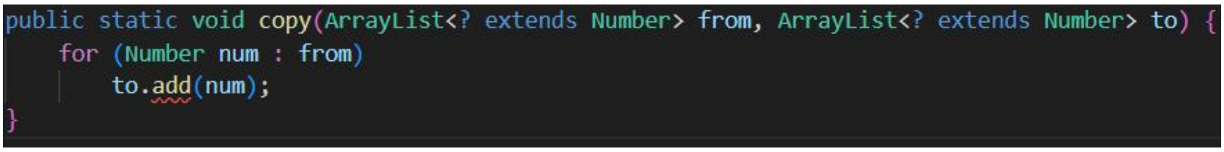
          ```
          List<Integer> list1 = Array    Type mismatch: cannot convert from List<Integer> to ArrayList<Integer>
                                         View Problem (Alt+F8)   Quick Fix... (Ctrl+.)   Fix (Ctrl+I)
          ArrayList<Integer> list2 = Arrays.asList(...a: 1, 2, 3, 4, 5, 6);
          ```
    - One more thing, as the **generic** doesn't support **primitive** type then how are we able to pass normal **1,2,3** .. inside the function as arguments.
        - You can see, in the definition of **asList** method, **(T… a)** is mentioned.
        - So, the **int** values are getting **auto-boxing** by default.
        - NOTE: We are not assigning **T** as **int**, but we are passing the **int** value; which are being converted to **Integer**.
        - The above is same as the below:
        - 

          ```
          List<Integer> list1 = Arrays.asList(Integer.valueOf(i: 1), Integer.valueOf(i: 2), Integer.valueOf(i: 3),
                  Integer.valueOf(i: 4), Integer.valueOf(i: 5), Integer.valueOf(i: 6));
          ```
        - **Integer.valueOf** is used to convert **int** to **Integer**.

- So, in shorts: when only **read-only** operation is being done and **not returning** anything; then **wildcards** can be used.
- It can also **extend** some class just like **generic T**
    - 

      ```
      public static double sum(ArrayList<? extends Number> numbers) {
          double sum = 0;
          for (Number o : numbers)
              sum += o.doubleValue();
          return sum;
      }
      ```
    - It is **upper-bound**.
    - It means max **Number** class can be given. Parent of Number cannot be given.

- Below is the example of **lower-bound**
  - 

    ```
    public static void printNumbers(List<? super Integer> list) {
        for (Object o : list) {
            System.out.println(o);
        }
    }
    ```
  - **super** keyword is used for this.
  - **NOTE:** You cannot write **T super Integer** for generics.
- In case of upper bound, we can't do **write** operation like below.
  - 

    ```
    public static void copy(ArrayList<? extends Number> from, ArrayList<? extends Number> to) {
        for (Number num : from)
            to.add(num);
    }
    ```
  - The **to** list can be of **Integer**, **Double** or anything.
  - Lets say the **from** list is of type **Integer**.
  - Here you need to know the type of **?** in **to** list to store the array i.e. the type of **to** is Double, and you are trying to store **Number** type (which is super class of Double and Integer), but to store **Parent type in Child type** you need to do downcast explicitly. Which is not possible in this case without knowing the exact type of **to** list.
  - So, if we know the **sub-class** then we can downcast and perform the **write** operation.
  - So, we can perform **write** operation in **lower-bound** type of wildcards.
-
- F
- F
- F
- F
- F
- F
-
-