

CS102

Programming II

GENERICS

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

TOPICS

- Java wrapper classes
 - Boxing / unboxing
- Generics
 - Methods
 - Classes
 - Extends type
 - Extends Comparable
 - Generics with Interface

CS102

Programming II

WRAPPER CLASSES

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

WRAPPER CLASSES

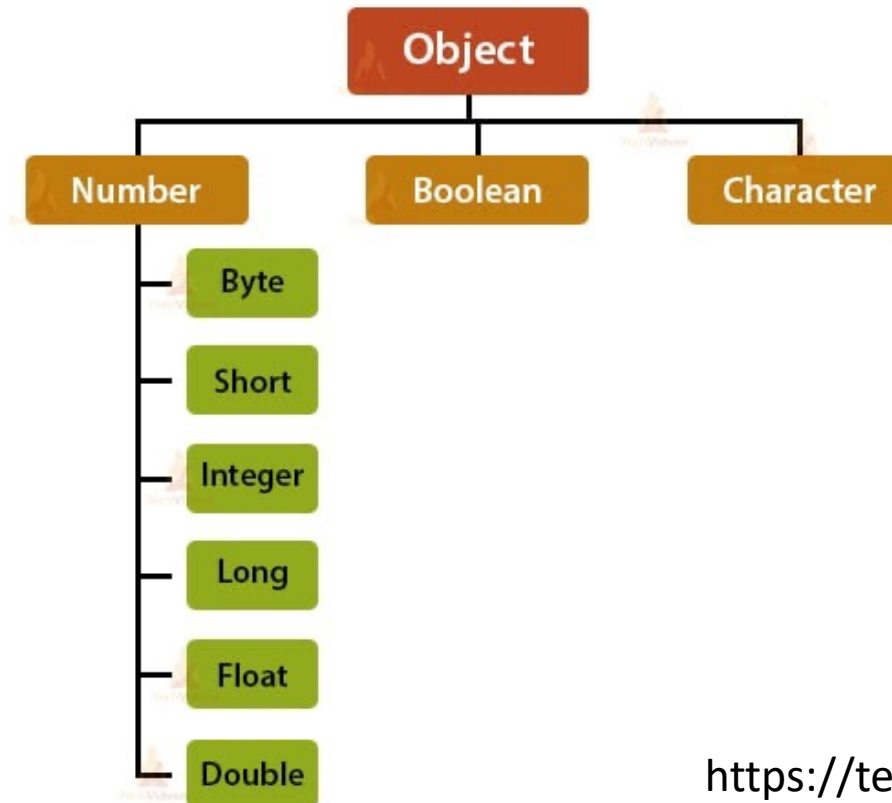
- Wrapper classes provide a way to use primitive data types (**int**, **boolean**, etc..) as objects.

Primitive Data Type	Wrapper Class
byte	Byte
short	Short
int	Integer
long	Long
float	Float
double	Double
boolean	Boolean
char	Character

WRAPPER CLASSES

- All the wrapper classes **Byte**, **Short**, **Integer**, **Long**, **Double** and, **Float**, are subclasses of the abstract class **Number**. While **Character** and **Boolean** wrapper classes are the subclasses of class **Object**.

Java Wrapper Class Hierarchy



<https://techvidvan.com/tutorials/java-wrapper-class/>

WRAPPER CLASSES

- Sometimes you must use wrapper classes, for example when working with **Collection** objects, such as **ArrayList**, where primitive types cannot be used (the list can only store objects)

```
ArrayList<int> myNumbers = new ArrayList<int>(); // Invalid
```

```
ArrayList<Integer> myNumbers = new ArrayList<Integer>(); // Valid
```

WRAPPER CLASSES

- To create a wrapper object, use the wrapper class instead of the primitive type. To get the value, you can just print the object:

```
public class Main {  
    public static void main(String[] args) {  
        Integer myInt = 5;  
        Double myDouble = 5.99;  
        Character myChar = 'A';  
        System.out.println(myInt);  
        System.out.println(myDouble);  
        System.out.println(myChar);  
    }  
}
```

WRAPPER CLASSES

- We use methods to get information about the specific object. For example, **intValue()**, **doubleValue()**, etc

```
public class Main {  
    public static void main(String[] args) {  
        Integer myInt = 5;  
        Double myDouble = 5.99;  
        Character myChar = 'A';  
        System.out.println(myInt.intValue());  
        System.out.println(myDouble.doubleValue());  
        System.out.println(myChar.charValue());  
    }  
}
```

WRAPPER CLASSES - AUTOBOXING

- The process to automatically convert the primitive data types into corresponding wrapper class objects is called Autoboxing in Java. This is Autoboxing because this is done automatically by the Java compiler.

```
//Using Collection Framework
```

```
ArrayList<Integer> arrayList = new ArrayList<Integer>();
```

```
arrayList.add(16); //Autoboxing
```

```
arrayList.add(35); //Autoboxing
```

WRAPPER CLASSES - UNBOXING

- Java Unboxing is the reverse process of Autoboxing. The process to convert the wrapper class object into its corresponding primitive data type is called Java Unboxing.

```
//Using Collection Framework
```

```
ArrayList<Integer> arrayList = new ArrayList<Integer>();
```

```
arrayList.add(50); //Autoboxing
```

```
//Unboxing object into int value
```

```
int num = arrayList.get(0);
```

```
System.out.println(num);
```

CS102

Programming II

GENERICS

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

GENERICIS

- Generics allow you to write classes, interfaces, and methods that work with different data types, without having to specify the exact type in advance.
- This makes your code more flexible, reusable, and type-safe.
- The benefit:
 - **Code Reusability:** Write one class or method that works with different data types.
 - **Type Safety:** Catch type errors at compile time instead of runtime.
 - **Cleaner Code:** No need for casting when retrieving objects.

GENERICIS

- Lets see how generics can help re-use code

```
Integer [] iArr = {1,2,3,4,5,6};  
Double [] dArr = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6};  
String [] sArr ={"a", "b", "c", "d", "e", "f"};  
for(int i=0; i< iArr.length; i++)  
    System.out.print(iArr[i] + " ");  
System.out.println();
```

```
for(int i=0; i< dArr.length; i++)  
    System.out.print(dArr[i] + " ");  
System.out.println();
```

```
for(int i=0; i< sArr.length; i++)  
    System.out.print(sArr[i] + " ");  
System.out.println();
```

GENERICS

- Generics <T> works with any type

```
public static <T> void printArr( T [] Arr){
```

```
    for(int i=0; i< Arr.length; i++)
```

```
        System.out.print(Arr[i] +" ");
```

```
    System.out.println();
```

```
}
```

```
// in the main method
```

```
Integer [] iArr = {1,2,3,4,5,6};
```

```
Double [] dArr = {1.1, 2.2, 3.3, 4.4, 5.5, 6.6};
```

```
String [] sArr ={"a", "b", "c", "d", "e", "f"};
```

```
printArr(iArr);
```

```
printArr(dArr);
```

```
printArr(sArr);
```

GENERICIS

- We can create a class that works with different data types using generics:

```
class Item <T>{  
    private T value;  
    T get() {  
        return value;  
    }  
    void set(T value) {  
        this.value = value;  
    }  
}
```

GENERICS

- We can create a class that works with different data types using generics:
- With `Item <String>`, `<T>` becomes `String`; with `Item <Integer>`, `<T>` becomes `Integer` and so on.
- Same class can be reused for different data types.

```
class Item <T>{  
    private T value;  
    T get() {  
        return value;  
    }  
    void set(T value) {  
        this.value = value;  
    }  
}
```

```
public static void main(String [] args) {  
    Item <Integer> i = new Item<>();  
    i.set(3);  
    System.out.println(i.get());  
  
    Item <String> s = new Item<>();  
    s.set("hello");  
    System.out.println(s.get());  
}
```

GENERICIS - EXTENDS

- Use the extends keyword to limit the types a generic class or method can accept. For example, we can require that the type must be a subclass of **Number**:

```
class Item <T extends Number>{
```

```
    T [] Arr;
```

```
    Item(T [] a){
```

```
        Arr = a;
```

```
    }
```

```
    double Average(){
```

```
        double sum = 0;
```

```
        for(int i=0;i<Arr.length;i++)
```

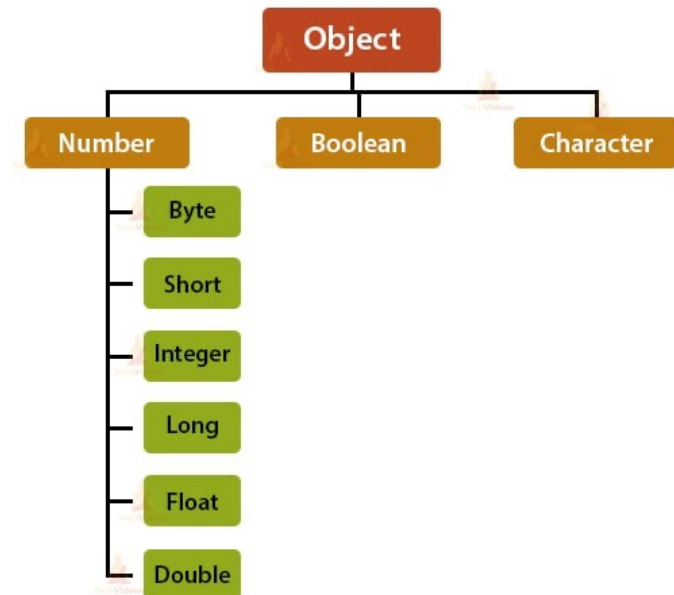
```
            sum += Arr[i].doubleValue();
```

```
        return sum/Arr.length;
```

```
    }
```

```
}
```

Java Wrapper Class Hierarchy



GENERICIS - EXTENDS

- Use the extends keyword to limit the types a generic class or method can accept. For example, we can require that the type must be a subclass of **Number**:

```
class Item <T extends Number>{
```

```
    T [] Arr;
```

```
    Item(T [] a){
```

```
        Arr = a;
```

```
    }
```

```
    double Average() {
```

```
        double sum = 0;
```

```
        for(int i=0;i<Arr.length;i++)
```

```
            sum += Arr[i].doubleValue();
```

```
        return sum/Arr.length;
```

```
    }
```

```
}
```

<T extends Number>: Restricts T to only work with numeric types like **Integer**, **Double**, or **Float**.

doubleValue(): Converts any number to a double for calculation.

Works for any array of numbers as long as the elements are subclasses of Number.

GENERIC - EXTENDS

- **<T extends Comparable<T>>** is a bounded type parameter in Java generics. It restricts T to types that implement the **Comparable** interface for their own type, allowing objects to be compared with one another.

```
public static <T extends Comparable<T>> T max(T a, T b) {  
    if (a.compareTo(b) > 0)  
        return a;  
    else  
        return b;  
}
```

```
System.out.println(max(10, 20));           // 20  
System.out.println(max(3.5, 2.5));         // 3.5  
System.out.println(max("Apple", "Banana")); // Banana
```

GENERIC - EXTENDS

- You can make your own class usable with this method by implementing **Comparable**.

```
class Student implements Comparable<Student> {
```

```
    int marks;
```

```
    Student(int marks) {
```

```
        this.marks = marks;
```

```
    }
```

```
    @Override
```

```
    public int compareTo(Student other) {
```

```
        return Integer.compare(this.marks, other.marks);
```

```
    }
```

```
}
```

```
System.out.println(max(new Student(30), new Student (90))); // 90
```

GENERIC - COLLECTIONS

- Java Collections like **ArrayList** and **HashMap** use generics internally:

```
ArrayList<String> list = new ArrayList<>();  
list.add("Apple");  
String fruit = list.get(0); // no need to type cast!
```

CS102

Programming II

GENERIC INTERFACES

Prof. Dr. Basit Qureshi
PhD FHEA SMIEEE MACM
www.drbasit.org

GENERICS - INTERFACE

- A generic interface is an interface that declares one or more type parameters. It allows the same interface to work with different data types while maintaining compile-time type safety.

```
interface Box<T> {  
    void set(T value);  
    T get();  
}
```

```
class IntegerBox implements Box<Integer> {  
    private Integer value;  
    public void set(Integer value) {  
        this.value = value;  
    }  
    public Integer get() {  
        return value;  
    }  
}
```

GENERIC - INTERFACE

- A generic interface is an interface that declares one or more type parameters. It allows the same interface to work with different data types while maintaining compile-time type safety.

```
interface Box<T> {  
    void set(T value);  
    T get();  
}
```

```
class IntegerBox implements Box<Integer> {  
    private Integer value;  
    public void set(Integer value) {  
        this.value = value;  
    }  
    public Integer get() {  
        return value;  
    }  
}
```

```
class StringBox implements Box<String> {  
    private String value;  
    public void set(String value) {  
        this.value = value;  
    }  
    public String get() {  
        return value;  
    }  
}
```

GENERICS - INTERFACE

- A generic interface is an interface that declares one or more type parameters. It allows the same interface to work with different data types while maintaining compile-time type safety.

```
interface Box<T> {  
    void set(T value);  
    T get();  
}
```

```
public static void main(String [] args){  
    Box<Integer> b1 = new IntegerBox();  
    b1.set(50);  
    System.out.println(b1.get()); // 100  
  
    Box<String> b2 = new StringBox();  
    b2.set("Java");  
    System.out.println(b2.get()); // Hello  
}
```

```
class IntegerBox implements Box<Integer> {  
    private Integer value;  
    public void set(Integer value) {  
        this.value = value;  
    }  
    public Integer get() {  
        return value;  
    }  
}
```

```
class StringBox implements Box<String> {  
    private String value;  
    public void set(String value) {  
        this.value = value;  
    }  
    public String get() {  
        return value;  
    }  
}
```

GENERICS - INTERFACE

- A generic interface is an interface that declares one or more type parameters. It allows the same interface to work with different data types while maintaining compile-time type safety.

```
interface Box<T> {  
    void set(T value);  
    T get();  
}
```

```
public static void main(String [] args){  
    Box<Integer> b1 = new GenericBox<>();  
    b1.set(50);  
  
    Box<String> b2 = new GenericBox<>();  
    b2.set("Java");  
  
    System.out.println(b1.get()); // 50  
    System.out.println(b2.get()); // Java  
}
```

```
class GenericBox<T> implements Box<T> {  
    private T value;  
  
    public void set(T value) {  
        this.value = value;  
    }  
  
    public T get() {  
        return value;  
    }  
}
```

A better
solution!

GENERICIS – INTERFACE WITH MORE TYPES

- An interface can declare more than one type parameter.

```
interface Pair<K, V> {  
    K getId();  
    V getName();  
}
```

```
class Student implements Pair<Integer, String> {  
    public Integer getId() {  
        return 101;  
    }  
  
    public String getName() {  
        return "Ali";  
    }  
}
```

```
public static void main(String [] args){  
    Pair<Integer, String> p = new Student();  
  
    System.out.println(p.getId());    // 101  
    System.out.println(p.getName());  // Ali    }  
  
}
```

GENERICS – INTERFACE WITH BOUNDED TYPE

- An interface can declare more than one type parameter.

```
interface Calculator<T extends Number> {  
    double add(T a, T b);  
}
```

```
class BasicCalculator<T extends Number>  
    implements Calculator<T> {  
  
    public double add(T a, T b) {  
        return a.doubleValue() + b.doubleValue();  
    }  
}
```

```
public static void main(String [] args){  
    Calculator<Integer> c1 = new BasicCalculator<>();  
    System.out.println(c1.add(10, 20)); // 30.0  
  
    Calculator<Double> c2 = new BasicCalculator<>();  
    System.out.println(c2.add(2.5, 3.5)); // 6.0  
  
    // Compilation error:  
    // Calculator<String> c4 = new BasicCalculator<>();  
}
```