Wrappers

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

The table below shows the primitive type and the equivalent wrapper class:

Primitive Data Type	Wrapper Class
Byte	Byte
Short	Short
Int	Integer
Long	Long
Float	Float
Double	Double
Boolean	Boolean
Char	Character

Use of Wrapper classes

- ✓ **Change the value in Method:** Java supports only call by value. So, if we pass a primitive value, it will not change the original value. But, if we convert the primitive value in an object, it will change the original value.
- ✓ **Serialization:** We need to convert the objects into streams to perform the serialization. If we have a primitive value, we can convert it in objects through the wrapper classes.
- ✓ **Synchronization:** Java synchronization works with objects in Multithreading.
- ✓ **java.util package:** The java.util package provides the utility classes to deal with objects.
- ✓ **Collection Framework:** Java collection framework works with objects only. All classes of the collection framework (ArrayList, LinkedList, Vector, HashSet, LinkedHashSet, TreeSet, PriorityQueue, ArrayDeque, etc.) deal with objects only.

Example:

```
//Java Program to convert all primitives into its corresponding
//wrapper objects and vice-versa
public class WrapperExample3{
public static void main(String args[]){
byte b=10;
short s=20;
int i=30;
long l=40;
float f=50.0F;
double d=60.0D;
char c='a';
boolean b2=true;
```

```
//Autoboxing: Converting primitives into objects
Byte byteobj=b;
Short shortobj=s;
Integer intobj=i;
Long  longobj=l;
Float floatobj=f;
Double doubleobj=d;
Character charobj=c;
Boolean boolobj=b2;

//Printing objects

System.out.println("---Printing object values---");
System.out.println("Byte object: "+byteobj);
System.out.println("Short object: "+shortobj);
System.out.println("Integer object: "+intobj);
System.out.println("Long object: "+longobj);
System.out.println("Float object: "+floatobj);
System.out.println("Double object: "+doubleobj);
System.out.println("Character object: "+charobj);
System.out.println("Boolean object: "+boolobj);

//Unboxing: Converting Objects to Primitives
byte bytevalue=byteobj;
short shortvalue=shortobj;
int intvalue=intobj;
long longvalue=longobj;
float floatvalue=floatobj;
double doublevalue=doubleobj;
char charvalue=charobj;
boolean boolvalue=boolobj;

//Printing primitives
System.out.println("---Printing primitive values---");
System.out.println("byte value: "+bytevalue);
System.out.println("short value: "+shortvalue);
System.out.println("int value: "+intvalue);
System.out.println("long value: "+longvalue);
System.out.println("float value: "+floatvalue);
System.out.println("double value: "+doublevalue);
System.out.println("char value: "+charvalue);
System.out.println("boolean value: "+boolvalue);
}
}
```

Output

```
---Printing object values---
Byte object: 10
Short object: 20
Integer object: 30
```

```
Long object: 40
Float object: 50.0
Double object: 60.0
Character object: a
Boolean object: true
---Printing primitive values---
byte value: 10
short value: 20
int value: 30
long value: 40
float value: 50.0
double value: 60.0
char value: a
boolean value: true
```

3.14: Autoboxing

Autoboxing

The automatic conversion of primitive data type into its corresponding wrapper class is known as autoboxing, for example, byte to Byte, char to Character, int to Integer, long to Long, float to Float, boolean to Boolean, double to Double, and short to Short.

Example:

```
public class WrapperExample1{
public static void main(String args[]){
//Converting int into Integer
int a=20;
Integer i=Integer.valueOf(a);//converting int into Integer explicitly
Integer j=a;//autoboxing, now compiler will write Integer.valueOf(a) internally
System.out.println(a+" "+i+" "+j);
}
}
```

Output

```
20  20  20
```

Unboxing

The automatic conversion of wrapper type into its corresponding primitive type is known as unboxing. It is the reverse process of autoboxing.

Example:

```
//Unboxing example of Integer to int
public class WrapperExample2
{
public static void main(String args[])
{
//Converting Integer to int
```

```
Integer a=new Integer(3);
int i=a.intValue();          //converting Integer to int explicitly
int j=a;                     //unboxing, now compiler will write a.intValue() internally
System.out.println(a+" "+i+" "+j);
}
}
```

Output

3 3 3

A1: STACK TRACE ELEMENTS

A Stack Trace is a list of method calls from the point when the application was started to the current location of execution within the program. A Stack Trace is produced automatically by the Java Virtual Machine when an exception is thrown to indicate the location and progression of the program up to the point of the exception. They are displayed whenever a Java program terminates with an uncaught exception.

- ✓ We can access the text description of a stack trace by calling the **printStackTrace()** method of the **Throwable** class.
- ✓ The **java.lang.StackTraceElement is a** class where each element represents asingle stack frame.
- ✓ We can call the **getStackTrace()** method to get an array of **StackTraceElement** objects that we want analyse in our program.

Class Declaration

Following is the declaration for **java.lang.StackTraceElement** class

public final class StackTraceElement extends **Object** implements Serializable

Class constructors

Constructor & Description
StackTraceElement(String declaringClass, String methodName, String fileName, int lineNumber) This creates a stack trace element representing the specified execution point.

Parameters:

- **declaringClass** – the fully qualified name of the class containing the execution point represented by the stack trace element.
- **methodName** – the name of the method containing the execution point represented by the stack trace element.

- **fileName** – the name of the file containing the execution point represented by the stack trace element, or null if this information is unavailable

- **lineNumber** – the line number of the source line containing the execution point represented by this stack trace element, or a negative number if thisinformation is unavailable. A value of -2 indicates that the method containing the execution point is a native method.
- **Throws:** NullPointerException – if declaringClass or methodName is null.

Methods in StackTraceElement class:

Method Name	Description
String getFileName()	Gets the name of the source file containing the execution point represented by the **StackTraceElement.**
int getLineNumber()	Gets the line number of the source file containing the execution point represented by the **StackTraceElement.**
String getClassName()	Gets the fully qualified name of the class containing the execution point represented by the **StackTraceElement.**
String getMethodName()	Gets the name of the method containing the execution point represented by the **StackTraceElement.**
boolean isNativeMethod()	Returns true if the execution point of the **StackTraceElement** is inside a native method.
String toString()	Returns a formatted string containing the class name, method name, file name and the line number, if available.

Example:

The following program for finding factorial(using recursion) prints the stack trace of a recursive factorial function.

```
import java.util.Scanner;
public class StackTraceTest
{
   public static int factorial(int n)
   {
      System.out.println(" Factorial ("+n+"):");
      Throwable t=new Throwable();
      StackTraceElement[] frames=t.getStackTrace();
      for(StackTraceElement f:frames)
      {
         System.out.println(f);
```

```
    }
    int r;

    if(n<=1)
      r=1;
    else
      r=n*factorial(n-1);

      System.out.println("return "+r);
      return r;
  }
  public static void main(String[] args)
  {
    Scanner in=new Scanner(System.in);
    System.out.println("Enter n: ");
    int n=in.nextInt();
    factorial(n);
  }
}
```

Output:

Enter n: 3

Factorial (3):

StackTraceTest.factorial(StackTraceTest.java:10)
StackTraceTest.main(StackTraceTest.java:30)

Factorial (2):

StackTraceTest.factorial(StackTraceTest.java:10)
StackTraceTest.factorial(StackTraceTest.java:20)
StackTraceTest.main(StackTraceTest.java:30)

Factorial (1):

StackTraceTest.factorial(StackTraceTest.java:10)
StackTraceTest.factorial(StackTraceTest.java:20)
StackTraceTest.factorial(StackTraceTest.java:20)
StackTraceTest.main(StackTraceTest.java:30)

return 1
return 2
return 6

A2: "assert" Keyword

- **Assertions are Boolean expressions that are used to test/validate the code.**
- **It is a statement in java that can be used to test your assumptions about the program.**
- **Java** assert **keyword is used to create assertions in Java, which enables us to test the assumptions about our program.**
- **For example, an assertion may be to make sure that an employee's age is positive number.**

- Assertion is achieved using "assert" keyword in java.
- While executing assertion, it is believed to be true. If it fails, JVM will throw an error named AssertionError. It is mainly used for testing purpose.
- **Following are the situations in which we can use the assertions:**
  - For making the program more readable and user friendly, the assert statements are used.
  - For validating the internal control flow and class invariant, the assertions are used.
- **Syntax of using Assertion:**

  There are two ways to use assertion.

  First way:

  1. **assert expression;**

  Here the **Expression** is evaluated by the JVM and if any error occurs then **AssertionError** occurs.

  Second way:

  2. **assert expression1 : expression2;**

  In this, Expression1 is evaluated and if it is false then the error message is displayed with the help of Expression2.
- **Assertion Enabling and Disabling:**

  By default, assertions are disabled. They have to be enabled explicitly

  For Enabling:

  We can enable the assertions by running the java program with the -**enableassertions** (or) **-ea** option:

```
java –enableassertions AssertionDemo
        (or)
java –ea AssertionDemo
```

For Disabling: **-disableassertions** (or) **-da**

```
java –disableassertions AssertionDemo
        (or)
java –da AssertionDemo
```

When assertions are disabled, the class loader strips out the assertion code so that it won't slow excetuion.

Example:

```
// Java program to demonstrate syntax of assertion
import java.util.Scanner;
class Test
{
public static void main( String args[] )
{
int value = 15;




assert value >= 20 : " Underweight";
System.out.println("value is "+value);
}
}
```

Output:

value is 15

After enabling assertions

Output:

Exception in thread "main" java.lang.AssertionError:
Underweight

- **Advantage of Assertions:**
  - It provides an effective way to detect and correct programming errors.

- **Where not to use Assertions**
  - Assertions should not be used to replace error messages
  - Do not use assertions for argument checking in public methods. Because if arguments are erroneous then that situation result in appropriate runtime exception such as **IllegalArgumentException, IndexOutOfBoundsException orNullPointerException.**