



# Wrapper Classes

# **Objectives**

At the end of this session, you will be able to:

- Describe the need for wrapper classes
- Define wrapper classes
- Understand Autoboxing & Unboxing
- Understand cloning

# Wrapper Classes



# Wrapper Classes

- For all the primitive data types available in Java, there is a corresponding Object representation available which is known as Wrapper Classes
- *Need for Wrapper Classes*
- All Collection classes in Java can store only Objects
- Primitive data types cannot be stored directly in these classes and hence the primitive values need to be converted to objects
- We have to wrap the primitive data types in a corresponding object, and give them an object representation

## Wrapper Classes (Contd.).

- Definition: The process of converting the primitive data types into objects is called *wrapping*
- To declare an integer 'i' holding the value 10, you write
  - `int i = 10;`
- The object representation of integer 'i' holding the value 10 will be:  
`Integer iref = new Integer(i);`
- Here, class Integer is the wrapper class wrapping a primitive data type i

## Wrapper Classes (Contd.).

- The Java API has provided a set of classes that make the process of wrapping easier. Such classes are called wrapper classes.
- For all the primitive data types, there are corresponding wrapper classes. Storing primitive types in the form of objects affects the performance in terms of memory and speed.
- Representing an integer via a wrapper takes about 12-16 bytes, compared to 4 in an actual integer. Also, retrieving the value of an integer uses the method **Integer.intValue()**.
- The wrapper classes are very useful as they enable you to manipulate primitive data types.

## Wrapper Classes (Contd.).

- For example, you can take the integer input from the user in the form of a **String**, and convert it into integer type using the following statements:

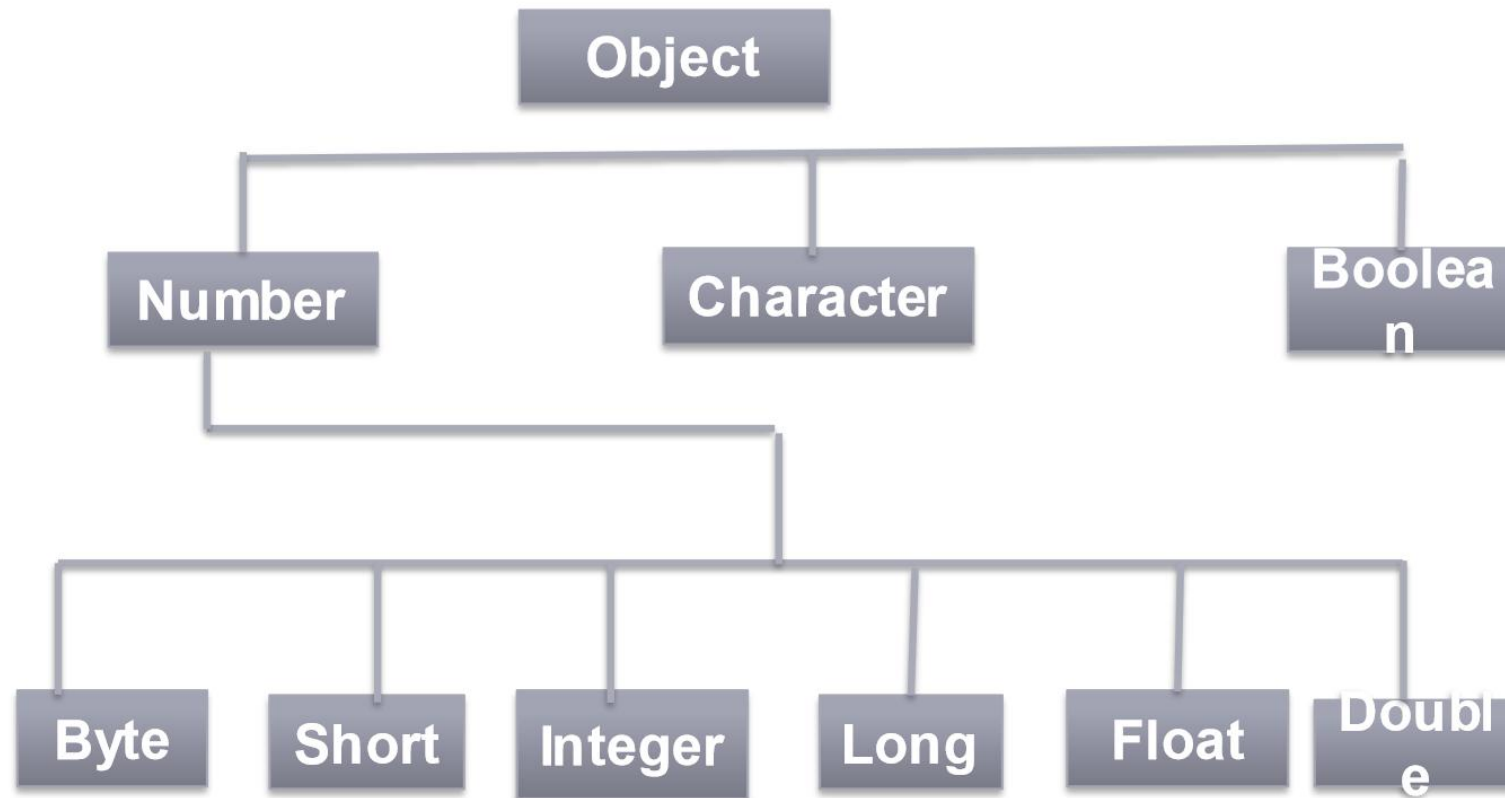
```
String str = "100";
```

```
int j = Integer.parseInt(str);
```

- There are many more methods in the wrapper classes that help you do several operations with the data types.
- The wrapper classes also have constants like :
- **MAX\_VALUE**, **MIN\_VALUE**, **NaN** (Not a Number), **POSITIVE\_INFINITY**, and **NEGATIVE\_INFINITY**.



## Wrapper Classes (Contd.).





# The Integer Class

- Class Integer is a wrapper for values of type int
- Integer objects can be constructed with a int value, or a string containing a int value
- The constructors for Integer are shown here:

```
Integer( int num)
```

```
Integer(String str) throws NumberFormatException
```

- Some methods of the Integer class:

```
static int parseInt(String str) throws  
NumberFormatException
```

```
int intValue( )
```

returns the value of the invoking object as a **int** value

*Refer Java documentation for other important methods*

## The Integer Class(Contd.).

```
Integer i1=new Integer(100);
```

```
Integer i2=new Integer("100");
```

Few more method of Integer class (These methods also available in Long, Short, Byte, Float, Double wrapper class)

**byteValue():** Returns the value of the invoking object as a byte.

**doubleValue():** Returns the value of the invoking object as a double.

**floatValue():** Returns the value of the invoking object as a float.

**longValue():** Returns the value of the invoking object as a long.

**shortValue():** Returns the value of the invoking object as a short.

**E.g.**

```
Integer i1=new Integer(20);
```

```
double d1=i1.doubleValue();
```

# The Character Class

- Character class is a wrapper class for character data types.
- The constructor for Character is:
  - **Character(char c)**
  - Here, c specifies the character to be wrapped by the Character object
- After a Character object is created, you can retrieve the primitive character value from it using:
  - **char charValue()**

*Refer Java documentation for other important methods*

# The Character Class

The **Character** class contains the following constants:

**MAX\_VALUE** - The largest character value.

**MIN\_VALUE** - The smallest character value.

**TYPE** - The Class object for char.

**Few more functions from Character class:**

**static String toString(char c)**

Returns a String object representing the specified char.

**static char toLowerCase(char ch)**

Converts the character argument to lowercase

**static char toUpperCase(char ch)**

Converts the character argument to uppercase .

# The Boolean Class

- The Boolean class is a wrapper class for boolean values
- It has the following constructors:
  - **Boolean(boolean bValue)**
    - Here, bValue can be either true or false
  - **Boolean(String str)**
    - The object created by this constructor will have the value true or false depending upon the string value in str – “true” or “false”
    - The value of str can be in upper case or lower case

*Refer Java documentation for other important methods*



# The Float Class

- Class Float is a wrapper for floating-point values of type float
- Float objects can be constructed with a float value, or a string containing a floating-point value
- The constructors for float are shown here:

`Float( float num)`

`Float( String str) throws NumberFormatException`

- Some methods of the Float class:

`static Float valueOf( String str) throws NumberFormatException`

`float floatValue( )`

returns the value of the invoking object as a **float** value

*Refer Java documentation for other important methods*

# The Double Class

- Class Double is a wrapper for floating-point values of type double
- Double objects can be constructed with a double value, or a string containing a floating-point value
- The constructors for double are shown here:

`Double( double num)`

`Double( String str) throws NumberFormatException`

- Some methods of the Double class:

`static Double valueOf( String str) throws NumberFormatException`

`double doubleValue( )`

returns the value of the invoking object as a **double** value

*Refer Java documentation for other important methods*



# The Long Class

- Class Long is a wrapper for values of type long
- Long objects can be constructed with a long value, or a string containing a long value
- The constructors for long are shown here:  
`Long( long num)`  
`Long( String str) throws NumberFormatException`
- Some methods of the Long class:  
`static Long valueOf(String str) throws NumberFormatException`  
`long longValue( )`  
returns the value of the invoking object as a **long** value

*Refer Java documentation for other important methods*

## Example

```
long ln=999;  
Long lng=new Long(ln);  
Long ls=new Long("666");  
System.out.println("long value="+lng.longValue());  
System.out.println("long value from string  
version="+ls.longValue());
```

### **Output:**

long value=999

long value from string version=666

# The Short Class

- Class Short is a wrapper for values of type short
- Short objects can be constructed with a short value, or a string containing a long value
- The constructors for short are shown here:

```
Short( short num)
```

```
Short( String str) throws NumberFormatException
```

- Some methods of the Short class:

```
static Short valueOf( String str) throws NumberFormatException
```

```
short shortValue( )
```

returns the value of the invoking object as a **short** value

*Refer Java documentation for other important methods*

## Example

```
short s=9;  
    Short sh=new Short(ln);  
    Short ls=new Short("6");  
    System.out.println("short value="+lng.shortValue());  
    System.out.println("short value from string  
version="+ls.shortValue());
```

### **Output:**

short value=999

short value from string version=666

# The Byte Class

- Class Byte is a wrapper for values of type byte
- Byte objects can be constructed with a byte value, or a string containing a long value
- The constructors for byte are shown here:

```
Byte( byte num)
```

```
Byte( String str) throws NumberFormatException
```

- Some methods of the Byte class:

```
static Byte valueOf( String str) throws NumberFormatException
```

```
byte byteValue( )
```

returns the value of the invoking object as a **byte** value

Refer Java documentation for other important methods

# AutoBoxing & UnBoxing

- Java 5.0 introduced automatic conversion between a primitive type and the corresponding wrapper class
- During assignment , the automatic transformation of primitive type to corresponding wrapper type is known as autoboxing
- Primitive types -----→ wrapper type  
(autoboxing)
- E. g.        `Integer i1=10;`
- During assignment , the automatic transformation of wrapper type into their primitive equivalent is known as Unboxing
- wrapper type -----→ primitive type  
(unboxing)
- E. g.        `int i=0;`  
              `i=new Integer(10) ;`



## AutoBoxing & UnBoxing (Contd.).

- Boxing conversion converts values of primitive type to corresponding values of reference type. But the primitive types can not be widened/ Narrowed to the Wrapper classes and vice versa.

**Wrong!!!**

```
byte b = 12;  
Integer l1=b;
```

**Wrong!!!**

```
byte b = 12;  
Integer l1=(Integer)b;
```

**Right!!!**

```
byte b = 12;  
Integer l1=(int)b;
```



## Quiz

What is the output of the following code?

```
class Test {  
    void m1(Integer i1) {  
        System.out.println("int value=" + i1);  
    }  
  
    public static void main(String a[]) {  
        Test t = new Test();  
        t.m1(10);  
    }  
}
```

## Quiz

What is the output of the following code?

```
class Test {  
    public void m1(Double x) {  
        System.out.println("Double");  
    }  
    public void m1(long x) {  
        System.out.println("long");  
    }  
    public static void main(String[] args) {  
        int x = 0;  
        Test t = new Test();  
        t.m1(x);  
        Long l1 = 10L;  
        t.m1(l1);  
    }  
}
```

In Function Overloading Widening  
/Narrowing Beats Boxing/UnBoxing

## Quiz(Contd.).

What is the output of the following code?

```
class Test {  
    static void fun(int i) {  
        System.out.println("int");  
    }  
    static void fun(Integer i) {  
        System.out.println("Integer");  
    }  
    public static void main(String args[]) {  
        byte b = 10;  
        fun(b);  
    }  
}
```

## Quiz(Contd.).

What is the output of the following code?

```
class Test {  
    public static void main(String ar[]) {  
        int x = 10;  
        Integer y = new Integer(10);  
        System.out.println(x == y);  
    }  
}
```

## Quiz(Contd.).

What is the output of the following code?

```
class Test {  
    public static void main(String ar[]) {  
        int x = 10;  
        Integer y = new Integer(10);  
        System.out.println(x == y);  
    }  
}
```

## Quiz(Contd.).

Which of the following is not a Wrapper Class?

1. Byte
2. Short
3. Integer
4. Long
5. String
6. Float
7. Double
8. Character
9. Boolean

# The Cloneable Interface

- When you make a copy of an object reference:
  - The original and copy are references to the same object
  - This means a change to either variable also affect the other
  
- The clone( ) method:
  - is a protected member of Object,
  - can only be invoked on an object that implements Cloneable
  
- Object cloning performs a bit-by-bit copy



## The Cloneable Interface(contd.).

- Objects can be cloned only of those classes that implement the **Cloneable** interface.
- The **Cloneable** interface has no members. It is a marker interface and is used to indicate that a class allows a bitwise copy of an object.
- If you call **clone( )** on a class that does not implement **Cloneable**, a **CloneNotSupportedException** is thrown.
- **When a clone is made, the constructor for the object being cloned is not called.**
- A clone is simply an exact copy of the original.

## Example on cloning

```
class XYZ implements Cloneable {  
    int a;  
    double b;  
  
    XYZ cloneTest() {  
        try {  
            return (XYZ) super.clone();  
        } catch (CloneNotSupportedException e) {  
            System.out.println("Cloning Not Allowed");  
            return this;  
        }  
    }  
}
```

## Example on cloning (Contd.).

```
class CloneDemo1 {  
    public static void main(String args[]) {  
        XYZ x1 = new XYZ();  
        XYZ x2;  
        x1.a = 10;  
        x1.b = 20;  
        x2 = x1.cloneTest(); // cloning x1  
        System.out.println("x1 : " + x1.a + " " + x1.b);  
        System.out.println("x2 : " + x2.a + " " + x2.b);  
        x1.a = 100;  
        x1.b = 200;  
        System.out.println("x1 : " + x1.a + " " + x1.b);  
        System.out.println("x2 : " + x2.a + " " + x2.b);  
    }  
}
```

### Output:

```
x1 : 10 20.0  
x2 : 10 20.0  
x1 : 100 200.0  
x2 : 10 20.0
```

# Summary

In this module, you were able to:

- Describe the need for wrapper classes
- Define wrapper classes
- Understand Autoboxing & Unboxing
- Understand cloning



**Thank you**