Java Lecture-3

Wrapper Classes in Java

- A Wrapper class in Java is a class whose object wraps or contains primitive data types.

- When we create an object to a wrapper class, it contains a field and in this field, we can store primitive data types.

Need of Wrapper Classes

1. They convert primitive data types into objects. Objects are needed if we wish to modify the arguments passed into a method (because primitive types are passed by value).
2. The classes in java.util package handles only objects and hence wrapper classes help in this case also.
3. Data structures in the Collection framework, such as ArrayList and Vector, store only objects (reference types) and not primitive types.
4. An object is needed to support synchronization in multithreading.

Advantages of Wrapper Classes

1. Collections allowed only object data.
2. On object data we can call multiple methods compareTo(), equals(), toString()
3. Cloning process only objects
4. Object data allowed null values.
5. Serialization can allow only object data

Primitive Data Types and their Corresponding Wrapper Class

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

Type Wrappers (Page 272, Complete Reference)

- Despite the performance benefit offered by the primitive types, there are times when you will need an object representation.

- For example, you can't pass a primitive type by reference to a method.

- many of the standard data structures implemented by Java operate on objects, which means that you can't use these data structures to store primitive types.

- To handle these (and other) situations, Java provides *type wrappers*, which are classes that encapsulate a primitive type within an object.

Character

Character is a wrapper around a **char**.

The constructor for **Character** is Character(char *ch*)

Here, *ch* specifies the character that will be wrapped by the **Character** object being created.

To obtain the **char** value contained in a **Character** object, call **charValue( )**, shown here:

char charValue( )

It returns the encapsulated character.

Boolean

Boolean is a wrapper around **boolean** values. It defines these constructors:
Boolean(boolean *boolValue*)

Boolean(String *boolString*)

In the first version, *boolValue* must be either **true** or **false**.

In the second version, if *boolString* contains the string "true" (in uppercase or lowercase), then the new **Boolean** object will be true. Otherwise, it will be false.

To obtain a **boolean** value from a **Boolean** object, use **booleanValue( )**, shown here:

boolean booleanValue( )

It returns the **boolean** equivalent of the invoking object.

The Numeric Type Wrappers

- All of the numeric type wrappers inherit the abstract class **Number**.

- **Number** declares methods that return the value of an object in each of the different number formats.

These methods are:

- byte byteValue( )
- double doubleValue( )
- float floatValue( )
- int intValue( )
- long longValue( )
- short shortValue( )

doubleValue( ) returns the value of an object as a **double**, **floatValue( )** returns the value as a **float**, and so on

All of the numeric type wrappers define constructors that allow an object to be constructed from a given value, or a string representation of that value.

For example, here are the constructors defined for **Integer**:

Integer(int *num*)

Integer(String *str*)

If *str* does not contain a valid numeric value, then a **NumberFormatException** is thrown.

```
// Demonstrate a type wrapper.
class Wrap {

  public static void main(String args[]) {

    Integer iOb = new Integer(100);

    int i = iOb.intValue();

    System.out.println(i + " " + iOb); // displays 100 100
  }
}
```

This program wraps the integer value 100 inside an **Integer** object called **iOb**. The program then obtains this value by calling **intValue( )** and stores the result in **i**.

The process of encapsulating a value within an object is called *boxing*.

Thus, in the program, this line boxes the value 100 into an **Integer**:

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

The process of extracting a value from a type wrapper is called *unboxing*.

For example, the program unboxes the value in **iOb** with this statement:

```
int i = iOb.intValue();
```

```
// Demonstrate autoboxing/unboxing.
class AutoBox {

 public static void main(String args[]) {

   Integer iOb = 100;            // autobox an int

   int i = iOb;                  // auto-unbox

   System.out.println(i + " " + iOb);          // displays 100 100
  }
}
```