UNIT 4
TYPE WRAPPERS

Type Wrappers

Java uses primitive types such as int or double, to hold the basic data types supported by the language. Primitive types, rather than objects, are used for these quantities for the **sake of performance**. Using objects for these values would add an **unacceptable overhead** to even the simplest of calculations.

The primitive types are **not part of the object hierarchy**, and they do not inherit Object. 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. Also, **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.**

The type wrapper classes.

The type wrappers are Double, Float, Long, Integer, Short, Byte, Character, and Boolean. These classes offer a wide array of methods that allow you to fully integrate the primitive types into Java's object hierarchy.

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

Commonly used numeric type wrappers are **Byte, Short, Integer, Long, Float, and Double**. 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 shown here:

byte byteValue( )

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

For example, doubleValue( ) returns the value of an object as a double, floatValue( ) returns the value as a float, and so on. These methods are implemented by each of the numeric type wrappers.
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. All of the type wrappers override toString( ). It returns the human-readable form of the value contained within the wrapper. This allows you to output the value by passing a type wrapper object to println( ), for example, without having to convert it into its primitive type. The following program demonstrates how to use a numeric type wrapper to encapsulate a value and then extract that value.

```
// 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();
The same general procedure used by the preceding program to box and unbox values has been employed since the original version of Java. However, since JDK 5, Java fundamentally improved on this through the addition of autoboxing.

Autoboxing

Beginning with JDK 5, Java added two important features: autoboxing and auto-unboxing. ***Autoboxing is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever an object of that type is needed***. There is no need to explicitly construct an object.

Auto-unboxing is the process by which the value of a boxed object is automatically extracted (unboxed) from a type wrapper when its value is needed. There is no need to call a method such as intValue( ) or doubleValue( ).

Advantagesof atoboxing and auto-unboxing:

- Greatly ***streamlines the coding*** of several algorithms, removing the tedium of manually boxing and unboxing values. You need only assign that value to a type-wrapper reference. Java automatically constructs the object for you.
- It also ***helps prevent errors***.
- It is very ***important to generics, which operate only on objects***.
- Autoboxing ***makes working with the Collections Framework much easier***.

For example, here is the modern way to construct an Integer object that has the value 100:

Integer iOb = 100; // autobox an int

Notice that the object is not explicitly created through the use of new. Java handles this for you, automatically.

To unbox an object, simply assign that object reference to a primitive-type variable. For example, to unbox iOb, you can use this line:

int i = iOb; // auto-unbox

Java handles the details for you.

Here is the preceding program rewritten to use autoboxing/unboxing:

```
// 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
 }
}
```

Autoboxing and Methods

Autoboxing/unboxing might occur when an argument is passed to a method, or when a value is returned by a method. For example, consider this:

```
// Autoboxing/unboxing takes place with method parameters and return values.
class AutoBox2 {
 // Take an Integer parameter and return an int value;
 static int m(Integer v) {
 return v ; // auto-unbox to int
 }
 public static void main(String args[]) {
```

```
 // Pass an int to m() and assign the return value to an Integer. Here, the argument 100 is
//autoboxed into an Integer. The return value is also autoboxed into an Integer.
 Integer iOb = m(100);
 System.out.println(iOb);
 }
}
```

This program displays the following result:

```
100
```

In the program, notice that m( ) specifies an Integer parameter and returns an int result. Inside main( ), m( ) is passed the value 100. Because m( ) is expecting an Integer, this value is automatically boxed. Then, m( ) returns the int equivalent of its argument. This causes v to be auto-unboxed. Next, this int value is assigned to iOb in main( ), which causes the int return value to be autoboxed.

Autoboxing/Unboxing Occurs in Expressions

In general, autoboxing and unboxing take place whenever a conversion into an object or from an object is required. This applies to expressions. Within an expression, a numeric object is automatically unboxed. The outcome of the expression is reboxed, if necessary. For example, consider the following program:

```
// Autoboxing/unboxing occurs inside expressions.
class AutoBox3 {
 public static void main(String args[]) {
 Integer iOb, iOb2;
 int i;
 iOb = 100;
 System.out.println("Original value of iOb: " + iOb);
 // The following automatically unboxes iOb, performs the increment, and then reboxes
 // the result back into iOb.
 ++iOb;
 System.out.println("After ++iOb: " + iOb);
 // Here, iOb is unboxed, the expression is evaluated, and the result is reboxed and
 // assigned to iOb2.
 iOb2 = iOb + (iOb / 3);
 System.out.println("iOb2 after expression: " + iOb2);
 // The same expression is evaluated, but the result is not reboxed.
 i = iOb + (iOb / 3);
 System.out.println("i after expression: " + i);
 }
}
```

The output is shown here:

```
 Original value of iOb: 100
 After ++iOb: 101
 iOb2 after expression: 134
 i after expression: 134
```

In the program, pay special attention to this line:

```
++iOb;
```

This causes the value in iOb to be incremented. It works like this: iOb is unboxed, the value is incremented, and the result is reboxed.
Auto-unboxing also allows you to mix different types of numeric objects in an expression. Once the values are unboxed, the standard type promotions and conversions are applied. For example, the following program is perfectly valid:

```
class AutoBox4 {
 public static void main(String args[]) {
 Integer iOb = 100;
 Double dOb = 98.6;
 dOb = dOb + iOb;
 System.out.println("dOb after expression: " + dOb);
 }
}
```

The output is shown here:

```
 dOb after expression: 198.6
```

As you can see, both the Double object dOb and the Integer object iOb participated in the addition, and the result was reboxed and stored in dOb.

Because of auto-unboxing, **you can use Integer numeric objects to control a switch statement.** For example, consider this fragment:

```
Integer iOb = 2;
switch(iOb) {
 case 1: System.out.println("one");
 break;
 case 2: System.out.println("two");
 break;
 default: System.out.println("error");
}
```

When the switch expression is evaluated, iOb is unboxed and its int value is obtained.

Autoboxing/Unboxing Boolean and Character Values

Java also supplies wrappers for boolean and char. These are Boolean and Character. Autoboxing/unboxing applies to these wrappers, too. For example, consider the following program:

```
// Autoboxing/unboxing a Boolean and Character.
class AutoBox5 {
 public static void main(String args[]) {
 // Autobox/unbox a boolean.
 Boolean b = true;
 // Below, b is auto-unboxed when used in a conditional expression, such as an if.
 if(b) System.out.println("b is true");
 // Autobox/unbox a char.
 Character ch = 'x'; // box a char
 char ch2 = ch; // unbox a char
 System.out.println("ch2 is " + ch2);
 }
}
```

The output is shown here:

```
 b is true
 ch2 is x
```

The most important thing to notice about this program is the auto-unboxing of b inside the if conditional expression. Because of auto-unboxing, the boolean value contained within b is automatically unboxed when the conditional expression is evaluated. Thus, with the advent of autoboxing/unboxing, a Boolean object can be used to control an if statement. Because of auto-unboxing, a Boolean object can now also be used to control any of Java's loop statements. When a Boolean is used as the conditional expression of a while, for, or do/while, it is automatically unboxed into its boolean equivalent. For example, this is now perfectly valid code:

```
Boolean b;
// ...
while(b) { // ...
```

<u>Autoboxing/Unboxing Helps Prevent Errors</u>

In addition to the convenience that it offers, autoboxing/unboxing can also help prevent errors. For example, consider the following program:

```
// An error produced by manual unboxing.
class UnboxingError {
 public static void main(String args[]) {
 Integer iOb = 1000; // autobox the value 1000
 int i = iOb.byteValue(); // manually unbox as byte !!!
 System.out.println(i); // does not display 1000 !
 }
}
```

This program displays not the expected value of 1000, but –24! The reason is that the value inside iOb is manually unboxed by calling byteValue( ), which causes the truncation of the value stored in iOb, which is 1,000. This results in the garbage value of –24 being assigned to i. Auto-unboxing prevents this type of error because the value in iOb will always autounbox into a value compatible with int.

In general, because autoboxing always creates the proper object, and auto-unboxing always produces the proper value, there is no way for the process to produce the wrong type of object or value. In the rare instances where you want a type different than that produced by the automated process, you can still manually box and unbox values. Of course, the benefits of autoboxing/unboxing are lost. In general, new code should employ autoboxing/unboxing. It is the way that modern Java code is written.

Exercise

Demonstrates the use of the auto-unboxing process in a switch statement. The day of the week entered is of the wrapper type Integer. In the switch statement, use this day of the week as a control expression. When comparing in case statements, automatic unboxing from type Integer to type int must occur. For the day entered display the day as day of the week(ex: For 1 display Monday)

```
import java.util.Scanner;
public class AutoPacking1 {
  public static void main(String[] args) {
    // Auto-boxing and unboxing in a switch statement
    // 1. Enter the day of the week, which is of type Integer
    Scanner input = new Scanner(System.in);
    System.out.print("day = ");
    Integer dayWeek = input.nextInt();
    // 2. Using the Integer type as a control expression in a switch statement
    // Here, the value of the dayWeek object is unpacked into a value of type int
    switch (dayWeek) {
      case 1:
        System.out.println("Monday");
        break;
      case 2:
        System.out.println("Tuesday");
        break;
      case 3:
        System.out.println("Wednesday");
        break;
      case 4:
        System.out.println("Thursday");
        break;
      case 5:
        System.out.println("Friday");
        break;
      case 6:
        System.out.println("Saturday");
        break;
      case 7:
        System.out.println("Sunday");
        break;
      default:
        System.out.println("Incorrect day");
    }
    input.close();
  }
}
```

OUTPUT

```
day = 7
Sunday
```

Exercise

```
package UNIT4;
/*Demonstrate the use of an  Boolean wrapper type object as control expression in the for,
while  and do-while loop statements for the following task.
 * (i)for loop--to find sum of numbers from 1 to 100.
 * (ii) while loop -- to find sum of even numbers from 1 to 100.
   (iii) do while loop--the sum of numbers entered from the keyboard until a 0 is entered*/
```

```java
import java.util.Scanner;
public class AutoPack2{
  public static void main(String[] args) {
    // Autoboxing and unboxing for Boolean type.
    // Using in loops for, while, do-while

    // 1. The for loop. The sum calculation s = 1 + 2 + ... + 100
    int i = 1;
    Boolean objB = i<=100;
    int sum = 0;
    // Unboxing the value objB: Boolean => boolean
    for (; objB; i++) {
      sum += i;
      // forming the condition result
      objB = i<100;
    }

    System.out.println("sum = " + sum);

    // 2. Loop while. Calculation the sum: 2 + 4 + 6 + ... + 100
    sum = 0;
    i = 1;
    objB = i<=100;
    // Unboxing objB: Boolean => boolean
    while (objB) {
      sum += i;
      i += 2;
      objB = i<=100;
    }
    System.out.println("sum = " + sum);
    // 3. Loop do-while.
    // Task. Calculate the sum of numbers entered from the keyboard.
    // The end of the input is the number 0.
    Scanner input = new Scanner(System.in);
    int number;
    System.out.println("Input numbers:");
    sum = 0;
    do {
      number = input.nextInt(); // input the number
      sum += number; // increase the sum
      objB = number != 0; // form a condition
    } while (objB); // automatic unboxing Boolean => boolean

    System.out.println("sum = " + sum);
  }
}
```
OUTPUT
```
sum = 5050
sum = 2500
Input numbers:
```

```
20
30
0
sum = 50
--------------------------------------------------------------
```

EXERCEISE

```
package UNIT4;
import java.util.Scanner;
/*Demonstrate the processes of  auto-boxing and
auto-unboxing of char and Character types for  the switch statement in a simple calculator
problem.*/
public class AutoPack3{
              public static void main(String[] args) {
                // Autoboxint and unboxing for char and Character types in the switch statement
                Character objC ;
        double a,b;
        Scanner input = new Scanner(System.in);
        System.out.println("Input 2 numbers:");
        a=input.nextDouble();
        b=input.nextDouble();
        System.out.println("Input operator character(+  -  /  *):");
        objC=input.next().charAt(0);
                // here is the auto-unboxing of objC into char type
                switch (objC) {
                  case '+':
                    System.out.println("Sum= "+ (a+b));
                    break;
                  case '-':
                    System.out.println("a-b= " +(a-b));
                    break;
                  case '*':
                    System.out.println("a*b= "+(a*b));
                    break;
                  case '/':
                    System.out.println("a/b= "+(a/b));
                    break;
                  default:
                    System.out.println("Undefined operation");
                }
              }
            }
```

OUTPUT

```
Input 2 numbers:
2
3
Input operator character(+  -  /  *):
+
Sum= 5.0
```