Type Wrapper
&
String Handling

Dr. Alekha Kumar Mishra

Type wrappers

- Many of the standard data structures implemented by Java operate on objects
- It can’t be used to store primitive types
- Java provides type wrappers, which are classes that encapsulate a primitive type within an object
- The common type wrappers are Double, Float, Long, Integer, Short, Byte, Character, and Boolean.

Some Wrapper constructors

- **Character**
  - Character(char ch)
- **Boolean**
  - Boolean(boolean boolValue)
  - Boolean(String boolString)
- **Interger**
  - Integer(int num)
  - Integer(String str)

Number abstract class

- All of the numeric type wrappers inherit the **Number** abstract class.
- Number declares methods that return the value of an object
  - char charValue( )
  - boolean booleanValue( )
  - byte byteValue( )
  - double doubleValue( )
  - float floatValue( )
  - int intValue( )
  - long longValue( )
  - short shortValue( )

```
// 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);
        }
}
```

Autoboxing/Auto-unboxing

- It is the process by which a primitive type is automatically encapsulated (boxed) into its equivalent type wrapper whenever that type is needed.
- 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( ).
- No longer necessary to manually construct an object in order to wrap a primitive type

Example: Autoboxing and Methods

- Autobox an integer
  - Integer iOb = 100; // autobox an int
- Auto-unbox integer
  - int i = iOb;
- Java handles the detail about these processes.

```
class AutoBox2 {
	static int m(Integer v) {
		return v ;
	}

	public static void main(String args[]) {
		Integer iOb = m(100);
		System.out.println(iOb);

	}
}
```

Example: Autoboxing/Unboxing in expressions

```
class AutoBox3 {
    public static void main(String args[]) {
        Integer iOb, iOb2;
        int i;
        iOb = 100;
        System.out.println("Original value of iOb: " + iOb);
        ++iOb;
        System.out.println("After ++iOb: " + iOb);
        iOb2 = iOb + (iOb / 3);
        System.out.println("iOb2 after expression: " + iOb2);
        i = iOb + (iOb / 3); // result is not reboxed.
        System.out.println("i after expression: " + i);
    }
}
```

Remarks

- Autoboxing/Unboxing Helps Prevent Errors
- Each autobox and auto-unbox adds overhead that is not present if the primitive type is used.
- So, restrict your use of the type wrappers to only those cases in which an object representation of a primitive type is required.
- Dont use as an alternate way of eliminating the primitive types

String s1="bhawana".
String s2="bhawana".

string constant pool ----- area where string objects are placed.
bhawana

here bhawana is single and both s1 and s2 are referring to this.

String handling in Java

String, StringBuffer and StringBuilder are the three classes using which we can create strings in java.
String objects which are created using String class are immutable means if any modification required, a new string will be created.
But strings from StringBuffer and StringBuilder are modifiable.

String-- String is basically a sequence of charachters.
but in JAVA, String is an object that represents a sequence of charachters.

Introduction to String

- Unlike other programming language, Java implment strings as object of type **String**
- Once a String object has been created, we cannot change the characters that comprise that string. (**String is Immutable**)
- Each time an altered version of an existing string is needed, a new **String** object is created that contains the modifications. (The original string is left unchanged)
- Java modifiable strings
  - **StringBuffer** and **StringBuilder**
- The String, StringBuffer, and StringBuilder classes are defined in java.lang

How to craete string object ?
1. Using String Keyword ----- String s="Bhawana".
2. Using new keyword ------ String s=new String("Bhawana");

String Constructor

- **Default constructor**
  - String s = new String();
- **Parameterized**(initialized by an array of characters)
  - char chars[] = { 'a', 'b', 'c' };
  - String s = new String(chars);
- **Parameterized**(initialized by range of chars of an array)
  - String(char chars[ ], int startIndex, int numChars)
- **Copy constructor**
  - String(String strObj)
- **Copy constructor** from StringBuffer
  - String(StringBuffer strBufObj)
- **Parameterized** (from byte array)
  - String(byte asciiChars[ ])
  - String(byte asciiChars[ ], int startIndex, int numChars)

Example

```
class MakeString {
	public static void main(String args[]) {
		char c[] = {'J', 'a', 'v', 'a'};
		String s1 = new String(c);
		String s2 = new String(s1);
		System.out.println(s1);
		System.out.println(s2);

		byte ascii[] = {65, 66, 67, 68, 69, 70 };
		String s3 = new String(ascii);
		System.out.println(s3);
		String s4= new String(ascii, 2, 3);
		System.out.println(s4);
	}
}
```

String method

- length of a string
  - int length( )
- Use of string literal
  - String s2 = "abc";
- **Character extraction**
  - charAt(int indx) //single character
  - getChars(int sourceStart, int sourceEnd, char target[ ], int targetStart) // multiple characters
  - byte[ ] getBytes( ) // stores the characters in an array of bytes
  - toCharArray( ) //String object into a character array,

String concatenation

- Concatenation (‘+’ operator)
  - Allows to chain together a series of + operations
  - String s = “first part” + “second part” + “third part”;
- Concatenate strings with other types of data.
  - int age = 9;
  - String s = "He is " + age + " years old.";
  - String s = "four: " + 2 + 2;
  - String s = "four: " + (2 + 2);

String conversion

- valueOf( ) defined by String.
- valueOf( ) is overloaded for all the simple types and for type Object.
- For objects, valueOf( ) calls the toString( ) method on the object
- The default implementation of toString( ) may be sufficient for primitive type, but for user-defined type it is recommened to override toString( ) and provide your own string representations

Example: String conversion

```
// Override toString() for Box class.
class Box {
	double width;
	double height;
	double depth;
	Box(double w, double h, double d) {
		width = w;
		height = h;
		depth = d;
	}
	public String toString() {
		return "Dimensions are " + width + " by " +
		depth + " by " + height + ".";
	}
}

class toStringDemo {
	public static void main(String args[]) {
		Box b = new Box(10, 12, 14);
		String s = "Box b: " + b; // concatenate Box object
		System.out.println(b); // convert Box to string
		System.out.println(s);
	}
}
```

String comparison methods

- boolean equals(Object str)
- boolean equalsIgnoreCase(String str)
- boolean regionMatches(int startIndex, String str2, int str2StartIndex, int numChars)
- boolean regionMatches(boolean ignoreCase, int startIndex, String str2, int str2StartIndex, int numChars)
- boolean startsWith(String str)
- boolean endsWith(String str)
- int compareTo(String str)
  - < 0 implies the invoking string is less than str.
  - > 0 implies the invoking string is greater than str.
  - = 0 implies two strings are equal.

equals( ) Versus ==

- the equals( ) method compares the characters inside a String object.
- The == operator compares two object references to see whether they refer to the same instance.

```
class EqualsNotEqualTo {
	public static void main(String args[]) {
		String s1 = "Hello";
		String s2 = new String(s1);
		System.out.println(s1 + " equals " + s2 + " -> " + s1.equals(s2));
		System.out.println(s1 + " == " + s2 + " -> " + (s1 == s2));
	}
}
```

Searching strings

- int indexOf(int ch) //search for the first occurrence of a character
- int lastIndexOf(int ch) // search the last occurrence of a character
- int indexOf(String str) //search for the first occurrence of a substring
- int lastIndexOf(String str) //search for the last occurrence of a substring
- int indexOf(int ch, int startIndex) //search for the first occurrence of a character from the startIndex position
- int lastIndexOf(int ch, int startIndex) //search for the last occurrence of a character from the startIndex position
- int indexOf(String str, int startIndex) //search for the first occurrence of a substring from the startIndex position
- int lastIndexOf(String str, int startIndex) //search for the first occurrence of a substring from the startIndex position

Modifying a String

- Extract substring
  - String substring(int startIndex)
  - String substring(int startIndex, int endIndex)
- Concate two strings
  - String concat(String str)
- Replace string
  - String replace(char original, char replacement) //replaces all occurrences of one character in the invoking string with another character
  - String replace(CharSequence original, CharSequence replacement) // replaces one character sequence with another.
- Trim string
  - String trim( ) //returns a copy of the invoking string from which any leading and trailing
  - whitespace has been removed

StringBuffer

- StringBuffer is a peer class of String
- Provides most of the functionality of strings.
- StringBuffer represents growable and writeable character sequences.
- StringBuffer will automatically grow when new char or substring inserted
- Not so popular as programmers prefer String and + operator along with methods for string manipulation

StringBuffer

- Constructors
  - StringBuffer( )
  - StringBuffer(int size)
  - StringBuffer(String str)
  - StringBuffer(CharSequencechars)
- Further Readings
  - All methods of StringBuffer class