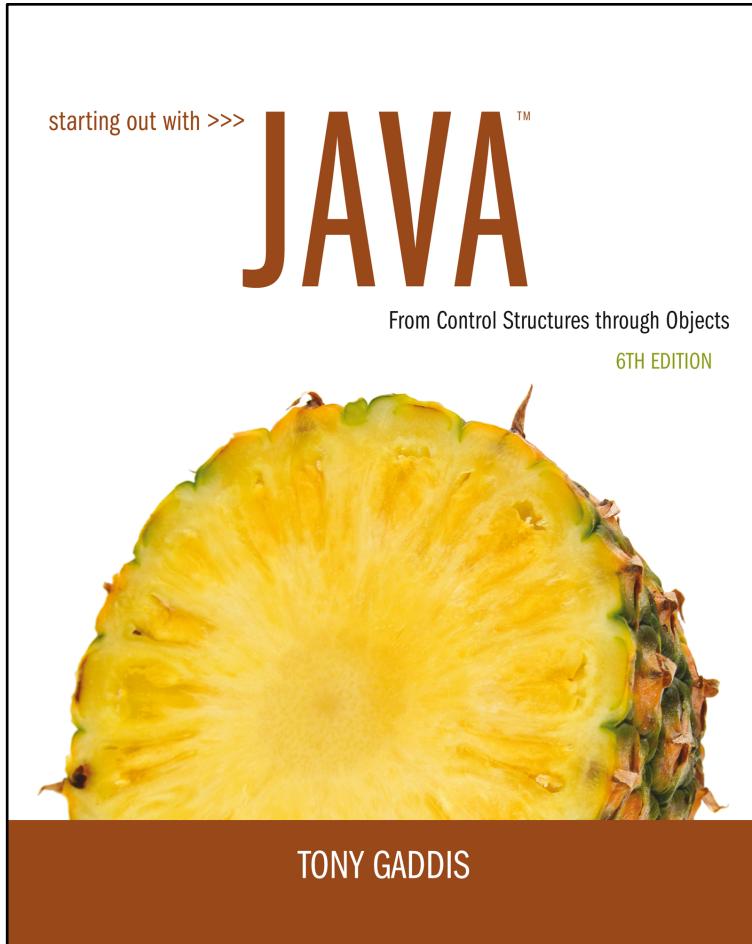


# Starting Out with Java: From Control Structures Through Objects

Sixth Edition



## Chapter 9

Text Processing and More  
about Wrapper Classes

# Chapter Topics

**9.1** Introduction to Wrapper Classes

**9.2** Character Testing and Conversion with the Character Class

**9.3** More String Methods

**9.4** The StringBuilder Class

**9.5** Tokenizing Strings

**9.6** Wrapper Classes for the Numeric Data Types

**9.7** Focus on Problem Solving: The TestScoreReader Class

## 9.1 Introduction to Wrapper Classes

- Java provides 8 primitive data types.
- They are called “primitive” because they are not created from classes.
- Java provides wrapper classes for all of the primitive data types.
- A **wrapper class** is a class that is “wrapped around” a primitive data type.
- The wrapper classes are part of `java.lang` so to use them, there is no `import` statement required.

## 9.2 Character Testing and Conversion with the Character Class (1 of 2)

- The Character class allows a char data type to be **wrapped** in an object.
- The Character class provides methods that allow easy testing, processing, and conversion of character data.

# The Character Class

Method	Description
boolean isDigit( char ch)	Returns true if the argument passed into <i>ch</i> is a digit from 0 through 9. Otherwise returns false.
boolean isLetter( char ch)	Returns true if the argument passed into <i>ch</i> is an alphabetic letter. Otherwise returns false.
boolean isLetterOrDigit( char ch)	Returns true if the character passed into <i>ch</i> contains a digit (0 through 9) or an alphabetic letter. Otherwise returns false.
boolean isLowerCase( char ch)	Returns true if the argument passed into <i>ch</i> is a lowercase letter. Otherwise returns false.
boolean isUpperCase( char ch)	Returns true if the argument passed into <i>ch</i> is an uppercase letter. Otherwise returns false.
boolean isSpaceChar( char ch)	Returns true if the argument passed into <i>ch</i> is a space character. Otherwise returns false.

## 9.3 Character Testing and Conversion with the **Character** Class (2 of 2)

- Example:
  - CharacterTest.java
  - CustomerNumber.java
- The `Character` class provides two methods that will change the case of a character.

Method	Description
<code>char toLowerCase(           char ch)</code>	Returns the lowercase equivalent of the argument passed to <code>ch</code> .
<code>char toUpperCase(           char ch)</code>	Returns the uppercase equivalent of the argument passed to <code>ch</code> .

- See example: CircleArea.java

# Substrings

- The `String` class provides several methods that search for a string inside of a string.
- A **substring** is a string that is part of another string.
- Some of the substring searching methods provided by the `String` class:

```
boolean startsWith(String str)
boolean endsWith(String str)
boolean regionMatches(int start, String str, int start2,
                      int n)
boolean regionMatches(boolean ignoreCase, int start,
                      String str, int start2, int n)
```

# Searching Strings (1 of 5)

- The `startsWith` method determines whether a string begins with a specified substring.

```
String str = "Four score and seven years ago";
if (str.startsWith("Four"))
    System.out.println("The string starts with Four.");
else
    System.out.println("The string does not start with
Four.");
```

- `str.startsWith("Four")` returns true because str does begin with “Four”.
- `startsWith` is a case sensitive comparison.

# Searching Strings (2 of 5)

- The `endsWith` method determines whether a string ends with a specified substring.

```
String str = "Four score and seven years ago";
if (str.endsWith("ago"))
    System.out.println("The string ends with ago.");
else
    System.out.println("The string does not end with ago.");
```

- The `endsWith` method also performs a case sensitive comparison.
- Example: PersonSearch.java

## Searching Strings (3 of 5)

- The `String` class provides methods that will if specified regions of two strings match.
  - `regionMatches(int start, String str, int start2, int n)`
    - returns true if the specified regions match or false if they don't
    - **Case sensitive comparison**
  - `regionMatches(boolean ignoreCase, int start, String str, int start2, int n)`
    - If `ignoreCase` is true, it performs case insensitive comparison

## Searching Strings (4 of 5)

- The `String` class also provides methods that will locate the position of a substring.
  - `indexOf`
    - returns the first location of a substring or character in the calling `String` Object.
  - `lastIndexOf`
    - returns the last location of a substring or character in the calling `String` Object.

# Searching Strings (5 of 5)

```
String str = "Four score and seven years ago";
int first, last;
first = str.indexOf('r');
last = str.lastIndexOf('r');
System.out.println("The letter r first appears at "
                  + "position " + first);
System.out.println("The letter r last appears at "
                  + "position " + last);
```

---

```
String str = "and a one and a two and a three";
int position;
System.out.println("The word and appears at the "
                  + "following locations.");
position = str.indexOf("and");
while (position != -1)
{
    System.out.println(position);
    position = str.indexOf("and", position + 1);
}
```

## Table 9-4 String Methods for Getting Character or Substring Location (1 of 2)

Method	Description
<code>int indexOf(char ch)</code>	Searches the calling String object for the character passed into <i>ch</i> . If the character is found, the position of its first occurrence is returned. Otherwise, 21 is returned.
<code>int indexOf(char ch, int start)</code>	Searches the calling String object for the character passed into <i>ch</i> , beginning at the position passed into <i>start</i> and going to the end of the string. If the character is found, the position of its first occurrence is returned. Otherwise, 21 is returned.
<code>int indexOf(String str)</code>	Searches the calling String object for the string passed into <i>str</i> . If the string is found, the beginning position of its first occurrence is returned. Otherwise, 21 is returned.
<code>int indexOf(String str, int start)</code>	Searches the calling String object for the string passed into <i>str</i> . The search begins at the position passed into <i>start</i> and goes to the end of the string. If the string is found, the beginning position of its first occurrence is returned. Otherwise, 21 is returned.

## Table 9-4 String Methods for Getting Character or Substring Location (2 of 2)

Method	Description
<code>int lastIndexOf(char ch)</code>	Searches the calling String object for the character passed into <i>ch</i> . If the character is found, the position of its last occurrence is returned. Otherwise, 21 is returned.
<code>int lastIndexOf(char ch, int start)</code>	Searches the calling String object for the character passed into <i>ch</i> , beginning at the position passed into <i>start</i> . The search is conducted backward through the string, to position 0. If the character is found, the position of its last occurrence is returned. Otherwise, 21 is returned.
<code>int lastIndexOf(String str)</code>	Searches the calling String object for the string passed into <i>str</i> . If the string is found, the beginning position of its last occurrence is returned. Otherwise, 21 is returned.
<code>int lastIndexOf(String str,                   int start)</code>	Searches the calling String object for the string passed into <i>str</i> , beginning at the position passed into <i>start</i> . The search is conducted backward through the string, to position 0. If the string is found, the beginning position of its last occurrence is returned. Otherwise, 21 is returned.

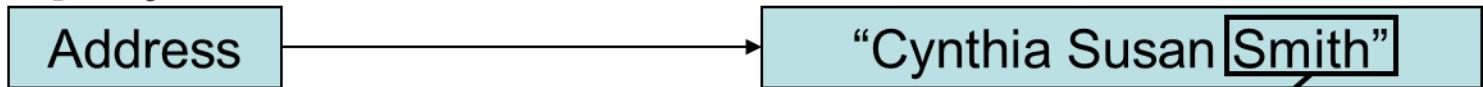
# Extracting Substrings (1 of 2)

- The `String` class provides methods to extract substrings in a `String` object.
  - The `substring` method returns a substring beginning at a start location and an optional ending location.

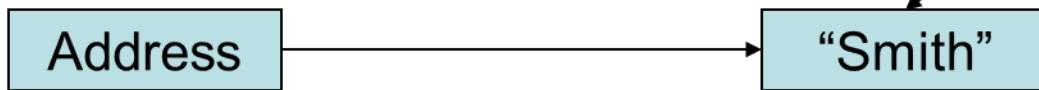
```
String fullName = "Cynthia Susan Smith";
String lastName = fullName.substring(14);
System.out.println("The full name is "
    + fullName);
System.out.println("The last name is "
    + lastName);
```

# Extracting Substrings (2 of 2)

The `fullName` variable holds the address of a String object.



The `lastName` variable holds the address of a String object.



# Extracting Characters to Arrays

- The `String` class provides methods to extract substrings in a `String` object and store them in `char` arrays.
  - `getChars`
    - Stores a substring in a `char` array
  - `toCharArray`
    - Returns the `String` object's contents in an array of `char` values.
- Example: `StringAnalyzer.java`

# Returning Modified Strings

- The `String` class provides methods to return modified `String` objects.
  - `concat`
    - Returns a `String` object that is the concatenation of two `String` objects.
  - `replace`
    - Returns a `String` object with all occurrences of one character being replaced by another character.
  - `trim`
    - Returns a `String` object with all leading and trailing whitespace characters removed.

## The `valueOf` Methods (1 of 2)

- The `String` class provides several overloaded `valueOf` methods.
- They return a `String` object representation of
  - a primitive value or
  - a character array.

`String.valueOf(true)` will return "true".

`String.valueOf(5.0)` will return "5.0".

`String.valueOf('C')` will return "C".

## The `valueOf` Methods (2 of 2)

```
boolean b = true;
char [] letters = { 'a', 'b', 'c', 'd', 'e' };
double d = 2.4981567;
int i = 7;
System.out.println(String.valueOf(b));
System.out.println(String.valueOf(letters));
System.out.println(String.valueOf(letters, 1,
3));
System.out.println(String.valueOf(d));
System.out.println(String.valueOf(i));
```

- Produces the following output:

```
true
abcde
bcd
2.4981567
7
```

## 9.4 The `StringBuilder` Class

- The `StringBuilder` class is similar to the `String` class.
- However, you may change the contents of `StringBuilder` objects.
  - You can change specific characters,
  - insert characters,
  - delete characters, and
  - perform other operations.
- A `StringBuilder` object will grow or shrink in size, as needed, to accommodate the changes.

# StringBuilder Constructors

- `StringBuilder()`
  - This constructor gives the object enough storage space to hold 16 characters.
- `StringBuilder(int length)`
  - This constructor gives the object enough storage space to hold `length` characters.
- `StringBuilder(String str)`
  - This constructor initializes the object with the string in `str`.
  - The object will have at least enough storage space to hold the string in `str`.

# Other **StringBuilder** Methods (1 of 3)

- The **String** and **StringBuilder** also have common methods:

```
char charAt(int position)
void getChars(int start, int end,
              char[] array, int arrayStart)
int indexOf(String str)
int indexOf(String str, int start)
int lastIndexOf(String str)
int lastIndexOf(String str, int start)
int length()
String substring(int start)
String substring(int start, int end)
```

# Appending to a `StringBuilder` Object (1 of 4)

- The `StringBuilder` class has several overloaded versions of a method named `append`.
- They append a string representation of their argument to the calling object's current contents.
- The general form of the `append` method is:  
`object.append(item);`
  - where `object` is an instance of the `StringBuilder` class and `item` is:
    - a primitive literal or variable.
    - a char array, or
    - a String literal or object.

# Appending to a `StringBuilder` Object (2 of 4)

- After the `append` method is called, a string representation of item will be appended to object's contents.

```
StringBuilder str = new StringBuilder();
```

```
str.append("We sold ");
str.append(12);
str.append(" doughnuts for $");
str.append(15.95);
```

```
System.out.println(str);
```

- This code will produce the following output:

We sold 12 doughnuts for \$15.95

# Appending to a `StringBuilder` Object (3 of 4)

- The `StringBuilder` class also has several overloaded versions of a method named `insert`
- These methods accept two arguments:
  - an `int` that specifies the position to begin insertion, and
  - the value to be inserted.
- The value to be inserted may be
  - a primitive literal or variable.
  - a `char` array, or
  - a `String` literal or object.

# Appending to a `StringBuilder` Object (4 of 4)

- The general form of a typical call to the `insert` method.
  - `object.insert(start, item);`
    - where **object** is an instance of the `StringBuilder` class, **start** is the insertion location, and **item** is:
      - a primitive literal or variable.
      - a `char` array, or
      - a `String` literal or object.
- Example:
  - Telephone.java
  - TelephoneTester.java

# Replacing a Substring in a `StringBuilder` Object (1 of 2)

- The `StringBuilder` class has a `replace` method that replaces a specified substring with a string.
- The general form of a call to the method:
  - `object.replace(start, end, str);`
    - `start` is an `int` that specifies the starting position of a substring in the calling object, and
    - `end` is an `int` that specifies the ending position of the substring. (The starting position is included in the substring, but the ending position is not.)
    - The `str` parameter is a `String` object.
  - After the method executes, the substring will be replaced with `str`.

# Replacing a Substring in a `StringBuilder` Object (2 of 2)

- The `replace` method in this code replaces the word “Chicago” with “New York”.

```
StringBuilder str = new StringBuilder(  
        "We moved from Chicago to Atlanta.");  
str.replace(14, 21, "New York");  
System.out.println(str);
```

- The code will produce the following output:

We moved from New York to Atlanta.

# Other **StringBuilder** Methods (2 of 3)

- The **StringBuilder** class also provides methods to set and delete characters in an object.

```
StringBuilder str = new StringBuilder(  
                    "I ate 100 blueberries!");  
// Display the StringBuilder object.  
System.out.println(str);  
// Delete the '0'.  
str.deleteCharAt(8);  
// Delete "blue".  
str.delete(9, 13);  
// Display the StringBuilder object.  
System.out.println(str);  
// Change the '1' to '5'  
str.setCharAt(6, '5');  
// Display the StringBuilder object.  
System.out.println(str);
```

# Other **StringBuilder** Methods (3 of 3)

- The `toString` method
  - You can call a `StringBuilder`'s `toString` method to convert that `StringBuilder` object to a regular `String`

```
StringBuilder strb = new StringBuilder("This is a test.");  
String str = strb.toString();
```

## 9.5 Tokenizing Strings

- Use the `String` class's `split` method
- Tokenizes a `String` object and returns an array of `String` objects
- Each array element is one token.

```
// Create a String to tokenize.  
String str = "one two three four";  
// Get the tokens from the string.  
String[] tokens = str.split(" ");  
// Display each token.  
for (String s : tokens)  
    System.out.println(s);
```

- This code will produce the following output:

```
one  
two  
three  
four
```

# Numeric Data Type Wrappers

- Java provides wrapper classes for all of the primitive data types.
- The numeric primitive wrapper classes are:

Wrapper Class	Numeric Primitive Type It Applies To
Byte	byte
Double	double
Float	float
Integer	int
Long	long
Short	short

# Creating a Wrapper Object

- To create objects from these wrapper classes, you can pass a value to the constructor:

```
Integer number = new Integer(7);
```

- You can also assign a primitive value to a wrapper class object:

```
Integer number;  
number = 7;
```

# The Parse Methods (1 of 2)

- Recall from Chapter 2, we converted `String` input (from `JOptionPane`) into numbers. Any `String` containing a number, such as “127.89”, can be converted to a numeric data type.
- Each of the numeric wrapper classes has a static method that converts a string to a number.
  - The `Integer` class has a method that converts a `String` to an `int`,
  - The `Double` class has a method that converts a `String` to a `double`,
  - etc.
- These methods are known as **parse methods** because their names begin with the word “parse.”

## The Parse Methods (2 of 2)

```
// Store 1 in bVar.  
byte bVar = Byte.parseByte("1");  
// Store 2599 in iVar.  
int iVar = Integer.parseInt("2599");  
// Store 10 in sVar.  
short sVar = Short.parseShort("10");  
// Store 15908 in lVar.  
long lVar = Long.parseLong("15908");  
// Store 12.3 in fVar.  
float fVar = Float.parseFloat("12.3");  
// Store 7945.6 in dVar.  
double dVar = Double.parseDouble("7945.6");
```

- The parse methods all throw a `NumberFormatException` if the `String` object does not represent a numeric value.

# The `toString` Methods

- Each of the numeric wrapper classes has a static `toString` method that converts a number to a string.
- The method accepts the number as its argument and returns a string representation of that number.

```
int i = 12;  
double d = 14.95;  
String str1 = Integer.toString(i);  
String str2 = Double.toString(d);
```

# The `toBinaryString`, `toHexString`, and `toOctalString` Methods

- The `Integer` and `Long` classes have three additional methods:
  - `toBinaryString`, `toHexString`, and `toOctalString`

```
int number = 14;  
System.out.println(Integer.toBinaryString(number));  
System.out.println(Integer.toHexString(number));  
System.out.println(Integer.toOctalString(number));
```

- This code will produce the following output:

```
1110  
e  
16
```

# **MIN\_VALUE and MAX\_VALUE**

- The numeric wrapper classes each have a set of static final variables
  - MIN\_VALUE and
  - MAX\_VALUE.
- These variables hold the minimum and maximum values for a particular data type.

```
System.out.println("The minimum value for an "
                    + "int is "
                    + Integer.MIN_VALUE);
System.out.println("The maximum value for an "
                    + "int is "
                    + Integer.MAX_VALUE);
```

# Autoboxing and Unboxing (1 of 2)

- You can declare a wrapper class variable and assign a value:

```
Integer number;  
number = 7;
```

- You may think this is an error, but because number is a wrapper class variable, **autoboxing** occurs.
- **Unboxing** does the opposite with wrapper class variables:

```
Integer myInt = 5;           // Autoboxes the value 5  
int primitiveNumber;  
primitiveNumber = myInt;    // unboxing
```

# Autoboxing and Unboxing (2 of 2)

- You rarely need to declare numeric wrapper class objects, but they can be useful when you need to work with primitives in a context where primitives are not permitted
- Recall the `ArrayList` class, which works only with objects.

```
ArrayList<int> list =  
    new ArrayList<int>();           // Error!  
  
ArrayList<Integer> list =  
    new ArrayList<Integer>(); // OK!
```

- Autoboxing and unboxing allow you to conveniently use `ArrayLists` with primitives.

# Problem Solving

- Dr. Harrison keeps student scores in an Excel file. This can be exported as a comma separated text file. Each student's data will be on one line. We want to write a Java program that will find the average for each student. (The number of students changes each year.)
- Solution: TestScoreReader.java, TestAverages.java

# Copyright

