CalPolyPomona

Text Processing and Wrapper Classes

Lecture 11a

Topics

- Introduction to Wrapper Classes
- Character Testing and Conversion with the Character Class
- More `String` Methods
- Tokenizing Strings
- Wrapper Classes for the Numeric Data Types
- The `StringBuilder` Class

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 the primitive data types.
- A *wrapper class* is a class that is “wrapped around” a primitive data type, containing not only a value but also methods that perform operations related to the type.
- The wrapper classes are part of `java.lang`, so to use them, there is no `import` statement required.

Wrapper Classes

- Wrapper classes allow you to create objects to represent a primitive instead of variables.
- Wrapper classes are immutable, which means that once you create an object, you cannot change the object's value.
- To get the value stored in an object you must call a method.
- Wrapper classes provide static methods that perform useful operations on primitive values.

Character Testing and Conversion With The `Character` Class

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

Character Testing and Conversion With The `Character` Class (1 of 2)

- Example:

  CharacterTest.java

  CustomerNumber.java

- The `Character` class provides two methods that will change the case of a character.

Method	Description
`char toLowerCase(char ch)`	Returns the lowercase equivalent of the argument passed to *ch*.
`char toUpperCase(char ch)`	Returns the uppercase equivalent of the argument passed to *ch*.

Character Testing and Conversion With The `Character` Class (2 of 2)

- Example:

```
System.out.println(Character.toLowerCase('A'));   a
System.out.println(Character.toUpperCase('a'));   A
```

- Any non-letter argument passed to `toLowerCase` or `toUpperCase` is returned as it is. Each of the following statements displays the method argument without any change:

```
System.out.println(Character.toLowerCase('*'));  *
System.out.println(Character.toLowerCase('$'));  $
System.out.println(Character.toUpperCase('&'));  &
System.out.println(Character.toUpperCase('%'));  %
```

More `String` Methods

- 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 6)

- 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 6)

- 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 6)

- The `regionMatches` method determines whether specified regions of two strings match.

```
String str = "Four score and seven years ago";
String str2 = "Those seven years passed quickly";
if (str.regionMatches(15, str2, 6, 11))
    System.out.println("The regions match.");
else
    System.out.println("The regions do not match.");
```

- This code will display “The regions match.” The specified region of the str string begins at position 15, and the specified region of the str2 string begins at position 6. Both regions consist of 11 characters forming “seven years”.

Searching Strings (4 of 6)

- An overloaded version of the `regionMatches` `method` accepts an additional argument indicating whether to perform a case-insensitive comparison.

```
String str = "Four score and seven years ago";
String str2 = "THOSE SEVEN YEARS PASSED QUICKLY";
if (str.regionMatches(true, 15, str2, 6, 11))
    System.out.println("The regions match.");
else
    System.out.println("The regions do not match.");
```

- This code will display “The regions match.”

Searching Strings (5 of 6)

- 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.
- If the item being searched is not found, `-1` is returned.

Searching Strings (6 of 6)

```
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);               3
System.out.println("The letter r last appears at "
                   + "position " + last);                 24
```

```
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.print(position + " ");
  position = str.indexOf("and", position + 1);
}                                                        0 10 20
```

String Methods For Getting Character Or Substring Location

Method	Description
int indexOf(char *ch*)	Searches the calling String object for the character passed into *ch*. If the character is found, the position of its first occurrence is returned. Otherwise, -1 is returned.
int indexOf(char *ch*, int *start*)	Searches the calling String object for the character passed into *ch*, beginning at the position passed into *start* and going to the end of the string. If the character is found, the position of its first occurrence is returned. Otherwise, -1 is returned.
int indexOf(String *str*)	Searches the calling String object for the string passed into *str.* If the string is found, the beginning position of its first occurrence is returned. Otherwise, -1 is returned.
int indexOf(String *str*, int *start*)	Searches the calling String object for the string passed into *str*. The search begins at the position passed into *start* and goes to the end of the string. If the string is found, the beginning position of its first occurrence is returned. Otherwise, -1 is returned.

`String` Methods For Getting Character Or Substring Location

Method	Description
int lastIndexOf(char *ch*)	Searches the calling String object for the character passed into *ch*. If the character is found, the position of its last occurrence is returned. Otherwise, -1 is returned.
int lastIndexOf(char *ch*, int *start*)	Searches the calling String object for the character passed into *ch*, beginning at the position passed into *start*. 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, -1 is returned.
int lastIndexOf(String *str*)	Searches the calling String object for the string passed into *str*. If the string is found, the beginning position of its last occurrence is returned. Otherwise, -1 is returned.
int lastIndexOf(String *str*, int *start*)	Searches the calling String object for the string passed into *str*, beginning at the position passed into *start*. 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, -1 is returned.

Extracting Substrings (1 of 3)

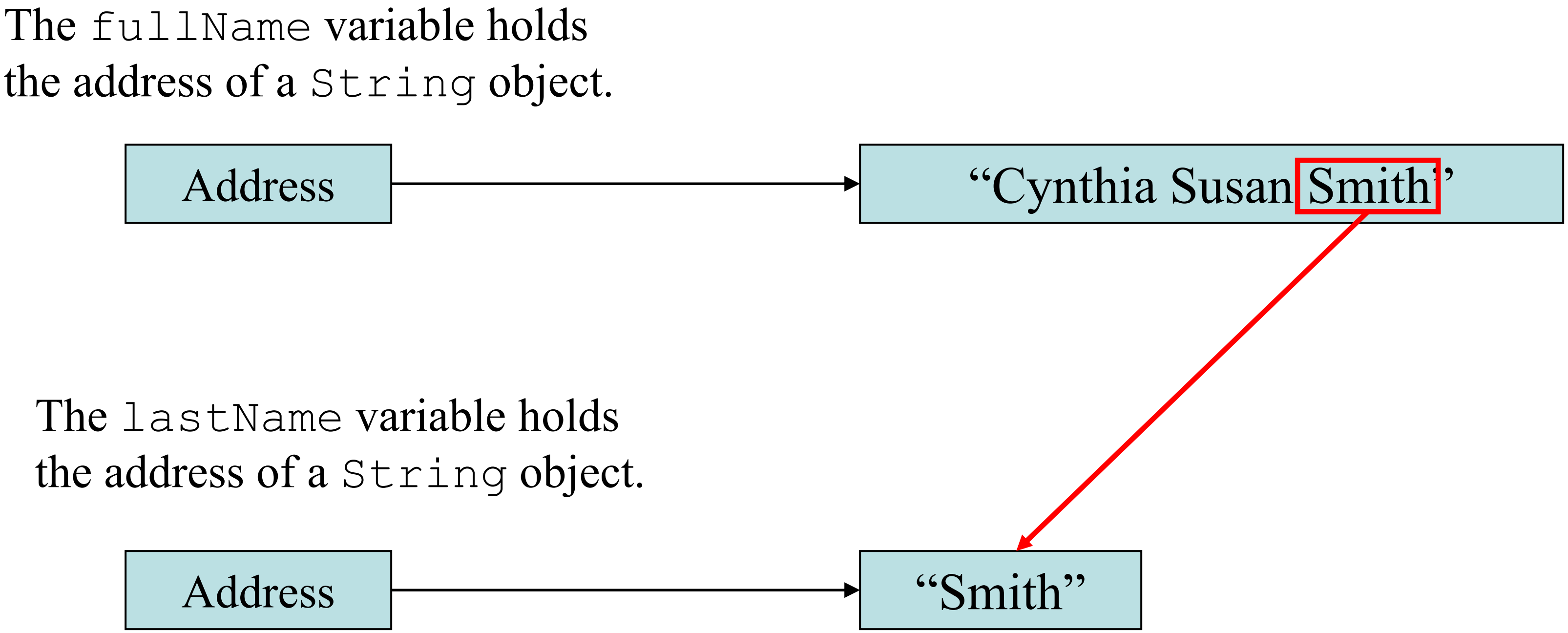
- The `String` class provides methods to extract substrings in a `String` object.
  - The `substring` method returns a substring beginning at a start location and an optional ending location.

```
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);
```

- This code will display:

```
The full name is Cynthia Susan Smith
The last name is Smith
```

Extracting Substrings (2 of 3)



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

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

Extracting Substrings (3 of 3)

- The second version of the method accepts two int arguments. The first specifies the substring's starting position (inclusive) and the second specifies the substring's ending position (exclusive).

```
String fullName = "Cynthia Susan Smith";
String middleName = fullName.substring(8, 13);
System.out.println("The full name is " + fullName);
System.out.println("The middle name is " + middleName);
```

- This code will display:

```
The full name is Cynthia Susan Smith
The middle name is Susan
```

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 entire string contents in a `char` array.
- 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.

```
System.out.println("John".concat(" Williams")); John Williams
```

  - `replace`
    - Returns a `String` object with all occurrences of one character being replaced by another character.

```
System.out.println("Hello-World".replace("-", " ")); Hello World
```

  - `trim`
    - Returns a `String` object with all leading and trailing whitespace characters removed.

```
System.out.println("   Hello World! ".trim()); Hello World
```

The `valueOf` Methods (1 of 3)

- 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".
```

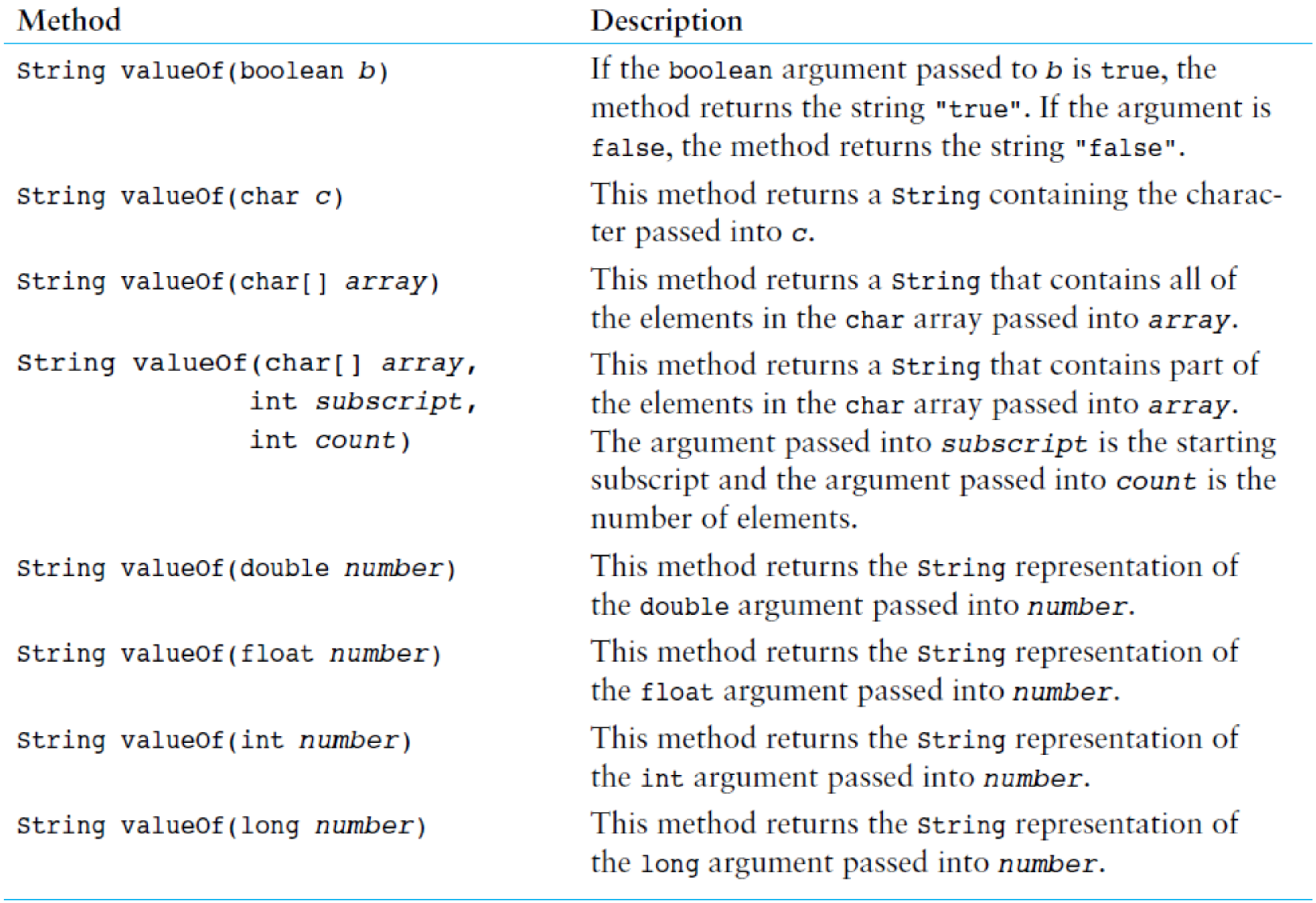
The valueOf Methods (2 of 3)

```
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
```

The `valueOf` Methods (3 of 3)



Method	Description
`String valueOf(boolean b)`	If the `boolean` argument passed to `b` is `true`, the method returns the string `"true"`. If the argument is `false`, the method returns the string `"false"`.
`String valueOf(char c)`	This method returns a `String` containing the character passed into `c`.
`String valueOf(char[] array)`	This method returns a `String` that contains all of the elements in the `char` array passed into `array`.

`String valueOf(char[] array,`
`int subscript,`
`int count)`
	This method returns a `String` that contains part of the elements in the `char` array passed into `array`. The argument passed into `subscript` is the starting subscript and the argument passed into `count` is the number of elements.

`String valueOf(double number)`	This method returns the `String` representation of the `double` argument passed into `number`.
`String valueOf(float number)`	This method returns the `String` representation of the `float` argument passed into `number`.
`String valueOf(int number)`	This method returns the `String` representation of the `int` argument passed into `number`.
`String valueOf(long number)`	This method returns the `String` representation of the `long` argument passed into `number`.

Tokenizing Strings (1 of 3)

- Process of breaking a string down into its components, which are called tokens.
- The character that separates tokens is known as a delimiter. Commonly used characters: ‘,’, ‘;’, ‘/’, ‘ ’, ‘-’, etc.
- The String class’s split method can be used to tokenize strings.
- For instance:

```
String str = "one two three four";
// Get the tokens, using a space delimiter.
String[] tokens = str.split(" ");
// Display the tokens.
 for (String s : tokens)
          System.out.println(s);
```

- Output:

```
One
Two
Three
four
```

- Example: SplitDemo1.java

Tokenizing Strings (2 of 3)

- The split method also allows you to use multi-character delimiters.
- For instance:

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

- Output:

```
one
two
three
four
```

- Example: SplitDemo2.java

Tokenizing Strings (3 of 3)

- You can also specify a series of characters where each individual character is a delimiter by enclosing them in brackets.
- For instance:

```
// Create a string to tokenize.
String str = "joe@gaddisbooks.com";
// Get the tokens, using @ and . as delimiters.
String[] tokens = str.split("[@.]");
// Display the tokens.
for (String s : tokens)
        System.out.println(s);
```

- Output:

```
joe
gaddisbooks
com
```

- Example: SplitDemo3.java

CalPolyPomona

Text Processing and Wrapper Classes

Lecture 11b

Topics

- Introduction to Wrapper Classes
- Character Testing and Conversion with the Character Class
- Tokenizing Strings
- More `String` Methods
- Wrapper Classes for the Numeric Data Types
- The `StringBuilder` Class

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`

The Parse Methods (1 of 2)

- We can convert `String` containing numbers, such as “127.89”, into 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)

Examples:

```
// 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 (1 of 2)

- 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);
```

MIN_VALUE and MAX_VALUE (2 of 2)

- Point to ponder #1:

Where the instruction Integer.MAX_VALUE can be useful for instance?

When initializing the variable min that will later have the smallest value in an array.

Creating a Wrapper Object

- It is possible to create objects from the wrapper classes passing a value to the constructor:

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

- This creates an Integer object initialized with the value 7, referenced by the variable number.

Autoboxing and Unboxing (1 of 2)

- Another way is to simply declare a wrapper class variable, and then assign a primitive value to it. For example, look at the following code:

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

- The first statement declares an Integer variable (not an object) named number.
- The second statement is a simple assignment statement. It assigns the primitive value 7 to the variable.
- You may think this is an error, but because number is a wrapper class variable, *autoboxing* occurs. Autoboxing is Java’s process of automatically “boxing up” a value inside an object.

Autoboxing and Unboxing (2 of 2)

- *Unboxing* does the opposite with wrapper class variables. It is the process of converting a wrapper class object to a primitive type.

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

Wrapper classes and ArrayLists (1 of 2)

- An ArrayList is an array-like object that can be used to store other objects.
- You cannot, directly, store primitive values in an ArrayList. It is intended for objects only. If you try to compile the following statement, an error will occur:

```
ArrayList<int> list = new ArrayList<int>(); // ERROR!
```

- However, you can store wrapper class objects in an `ArrayList`. If we need to store `int` values in an `ArrayList`, we have to specify that the `ArrayList` will hold `Integer` objects. Here is an example:

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

Wrapper classes and ArrayLists (2 of 2)

- To store an `int` value in an `ArrayList`, we need to instantiate an `Integer` object, initialize it with the desired `int` value, and then pass the `Integer` object to the `ArrayList`'s `add` method.

```
ArrayList<Integer> list = new ArrayList<Integer>();
Integer myInt = 5;
list.add(myInt);
```

- However, Java's autoboxing and unboxing features make it unnecessary to create the `Integer` object. If you add an `int` value to the `ArrayList`, Java will autobox the value. The following code works without any problems:

```
ArrayList<Integer> list = new ArrayList<Integer>();
list.add(5);
int primitiveNumber = list.get(0);
```

- The last statement in this code retrieves the item at index 0. Because the item is being assigned to an int variable, Java unboxes it and stores the primitive value in the int variable.

Other uses of the `valueOf` Methods

- It can be used for the other wrapper classes. Example:

```
Integer.valueOf("5") will return 5.
Double.valueOf("5.5") will return 5.5.
```

- Point to ponder #2:

What is the difference between `Integer.valueOf("5")` and `Integer.parseInt("5")`?

parseInt returns a primitive int while valueOf returns a new Integer().

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 (1 of 2)

- `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`* plus 16.

StringBuilder Constructors (2 of 2)

- Example of use:

```
StringBuilder city = new StringBuilder("Charleston");
System.out.println(city);
```

- One limitation of the StringBuilder class is that you cannot use the assignment operator to assign strings to StringBuilder objects:

```
StringBuilder city = "Charleston"; //ERROR!!!
```

Other StringBuilder Methods

- 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 ");           // Append a String object.
str.append(12);                   // Append an int.
str.append(" doughnuts for $");   // Append another String.
str.append(15.95);                // Append a double.

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:

```
StringBuilder str = new StringBuilder("New City");
str.insert(4, "York ");
System.out.println(str);
```

- This code will produce the following output:

```
New York City
```

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”.

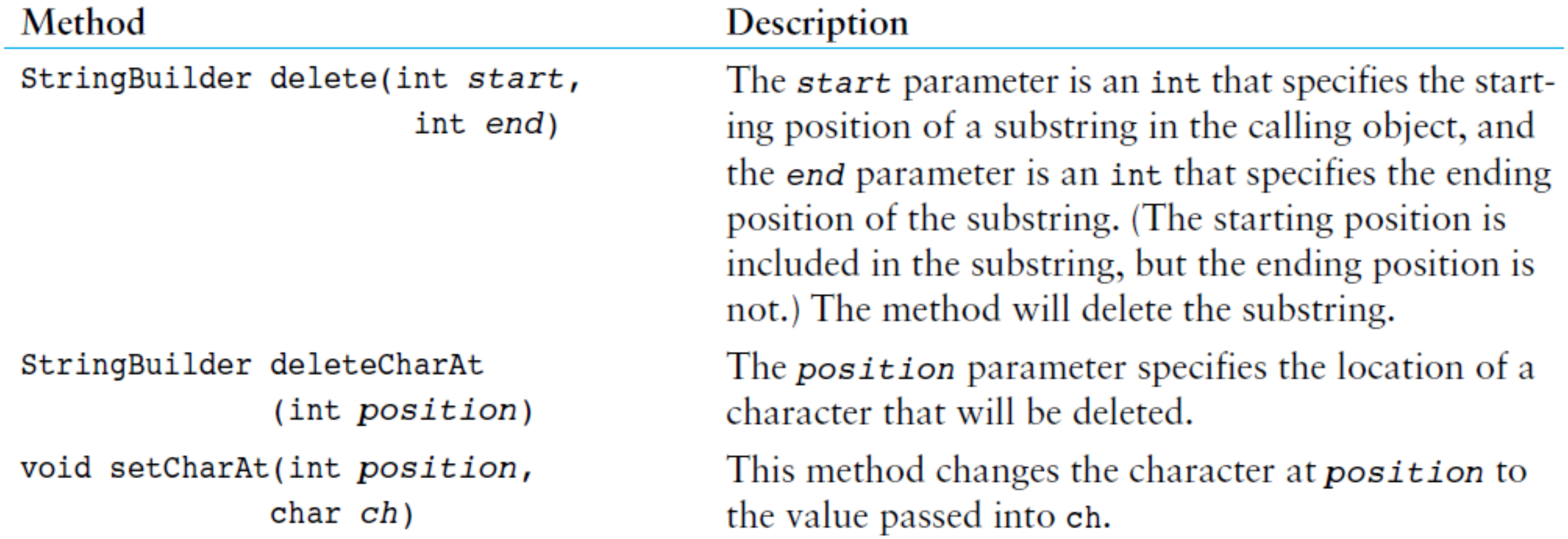
```
StringBuilder str = new StringBuilder("We moved from Chicago to Atlanta.");
str.replace(14, 21, "New York");
System.out.println(str);
```

- The code will produce the following output:

```
We moved from New York to Atlanta.
```

Other `StringBuilder` Methods (1 of 3)

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



Method	Description
`StringBuilder delete(int start, int end)`	The *`start`* parameter is an `int` that specifies the starting position of a substring in the calling object, and the *`end`* parameter 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 method will delete the substring.
`StringBuilder deleteCharAt (int position)`	The *`position`* parameter specifies the location of a character that will be deleted.
`void setCharAt(int position, char ch)`	This method changes the character at *`position`* to the value passed into `ch`.

Other `StringBuilder` Methods (2 of 3)

- Example.

```
StringBuilder str = new StringBuilder("I ate 100 blueberries!");
// Display the StringBuilder object.
System.out.println(str);          //I ate 100 blueberries!
// Delete the '0'.
str.deleteCharAt(8);
System.out.println(str);          //I ate 10 blueberries!
// Delete "blue".
str.delete(9, 13);
System.out.println(str);          //I ate 10 berries!
// Change the '1' to '5'
str.setCharAt(6, '5’);
System.out.println(str);          //I ate 50 berries!
```

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();
```

See Telephone.java, TelephoneTester.java

Other `StringBuilder` Methods (3 of 3)

- Point to ponder #3:

Overall, when we should use `StringBuilder` instead of `String` class?

When the program needs to make a lot of changes to one or more strings. This will improve the program's efficiency by reducing the number of String objects that must be created and then removed by the garbage collector.