OOP: Thinking in Classes

Lecture 3

Abstraction and Encapsulation

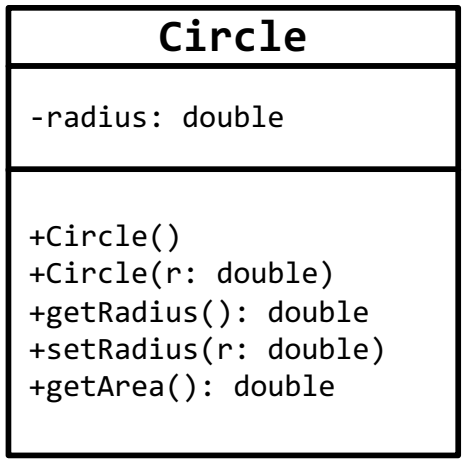
- Class **abstraction** means to separate class *implementation* from the *use* of the class

- The creator of the class provides a description of the class via public methods/variables
  - This lets the user know what the class can do

- The user of the class does not need to know *how* the class is implemented
  - Thus the details of class implementation are **encapsulated** and hidden from the user

Example

- The public methods form a contract via public (+) methods/constants
- Client interacts through these means and need not know the details of implementation



Wrapper Classes

- Java has built-in "wrapper" classes for all primitive types

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

General Wrapper Details

- These classes do NOT have no-arg constructors

- The instances of all wrapper classes are **immutable**
  - Their internal values cannot be changed once the objects are created

The `Integer` and `Double` Classes

java.lang.Integer
-value: int +MAX_VALUE: int +MIN_VALUE: int
+Integer(value: int) +Integer(s: String) +byteValue(): byte +shortValue(): short +intValue(): int +longValue(): long +floatValue(): float +doubleValue(): double +compareTo(o: Integer): int +toString(): String +valueOf(s: String): Integer +valueOf(s: String, radix: int): Integer +parseInt(s: String): int +parseInt(s: String, radix: int): int ...

java.lang.Double
-value: double +MAX_VALUE: double +MIN_VALUE: double
+Double(value: double) +Double(s: String) +byteValue(): byte +shortValue(): short +intValue(): int +longValue(): long +floatValue(): float +doubleValue(): double +compareTo(o: Double): int +toString(): String +valueOf(s: String): Double +valueOf(s: String, radix: int): Double +parseDouble(s: String): double +parseDouble(s: String, radix: int): double ...

Creating Wrappers

- You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value
- The **`valueOf`** static methods create new objects from a string representation
- The parseing static methods extract primitive values from a string representation
- Radix: numeric base

Exercise

What is the output of the following code…

```
System.out.printf("%d%n",
  Integer.parseInt("14"));
System.out.printf("%d%n",
  Integer.parseInt("14", 10));
System.out.printf("%d%n",
  Integer.parseInt("14", 16));
```

Answer

```
14
14
20
```

Conversion Methods

- The **`doubleValue`**, **`intValue`**, … allow you to convert objects to primitives

```
System.out.printf("%d%n",
   new Double("14.1").intValue());
```

Class Constants

- **`MAX_VALUE`**
  - Maximum value of the corresponding primitive data type

```
int iMax = Integer.MAX_VALUE; // 2147483647
double dMax = Double.MAX_VALUE; // 1.80e+308
```

- **`MIN_VALUE`**
  - **`Float`**/**`Double`**: minimum *positive* value

```
int iMin = Integer.MIN_VALUE; // -2147483648
double dMin = Double.MIN_VALUE;  // 4.90e-324
```

Comparison

The **`compareTo(o)`** method returns…

- 0: **`this`** and **`o`** are equal
- < 0: **`this`** is smaller/**`o`** is bigger
- > 0: **`this`** is bigger/**`o`** is smaller

```
final Integer i = new Integer(5);
System.out.printf("%d%n",
   i.compareTo(5)); // 0
System.out.printf("%d%n",
   i.compareTo(10)); // -1
System.out.printf("%d%n",
   i.compareTo(2)); // 1
```

Exercise

Write a method biggerOf that takes two int's and returns a String…

```
System.out.printf("%s%n",
  biggerOf(1, 100)); // b
System.out.printf("%s%n",
  biggerOf (100, 1)); // a
System.out.printf("%s%n",
  biggerOf(100, 100)); // equal
```

Answer

```
public static String biggerOf(int a, int b) {
    final int c =
        Integer.valueOf(a).compareTo(Integer.valueOf(b));

    return (c==0)?"equal":((c>0)?"a":"b");
}
```

The `String` Class

- Construction
```
String s1 = "Things";
String s2 = new String("Stuffs");
```
- Length, Character
```
s1.length(), s1.charAt(i)
```
- Concatenation
```
String s3 = s1.concat(s2) // s1 + s2
```
- Substring
```
s1.substring(2) // "ings"
s2.substring(1, 5) // "tuff"
```
- Comparisons
```
s1.equals(s2), s2.compareTo(s1)
```

Strings are Immutable

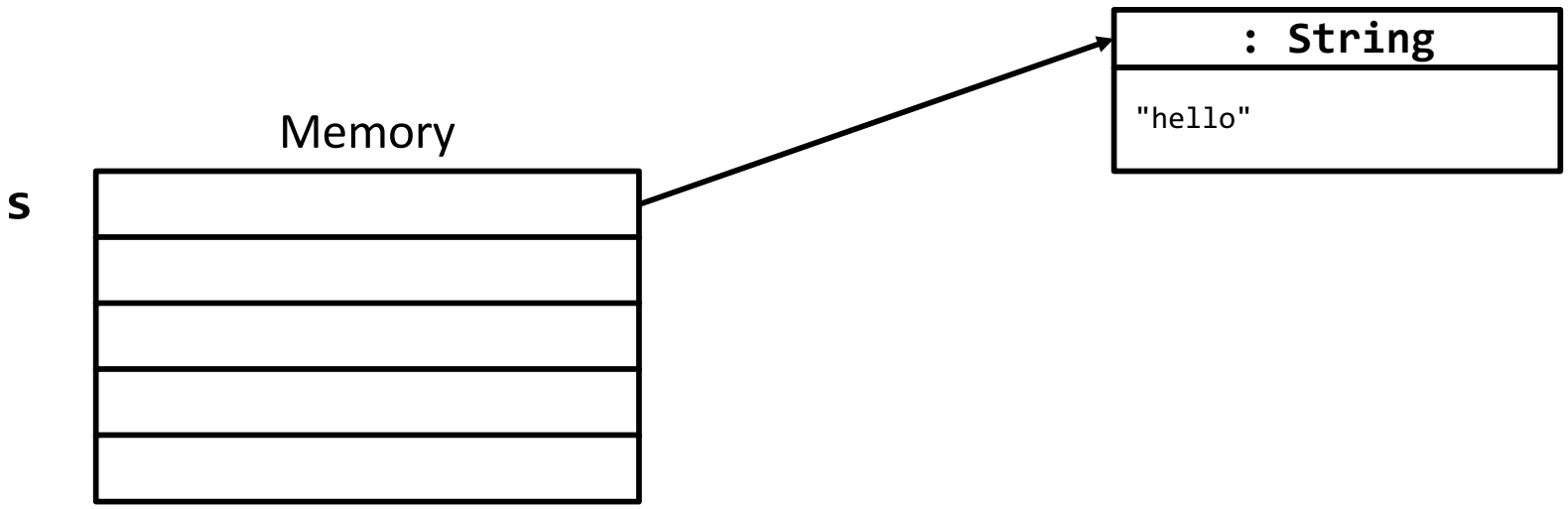
- A **`String`** object is **immutable** – once constructed, its contents *cannot* be changed

- What does the following code do?

```
String s = "hello";
s = "world";
```

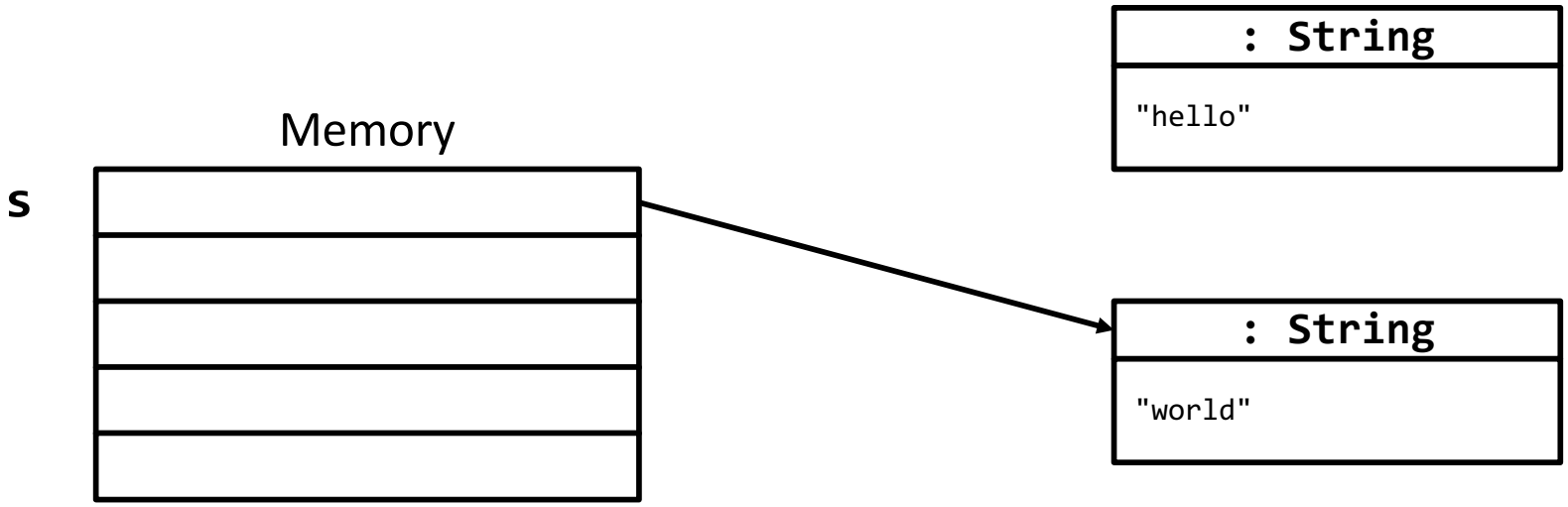
Trace Code (1)

```
String s = "hello";
s = "world";
```



Trace Code (2)

```
String s = "hello";
s = "world";
```



Interned Strings

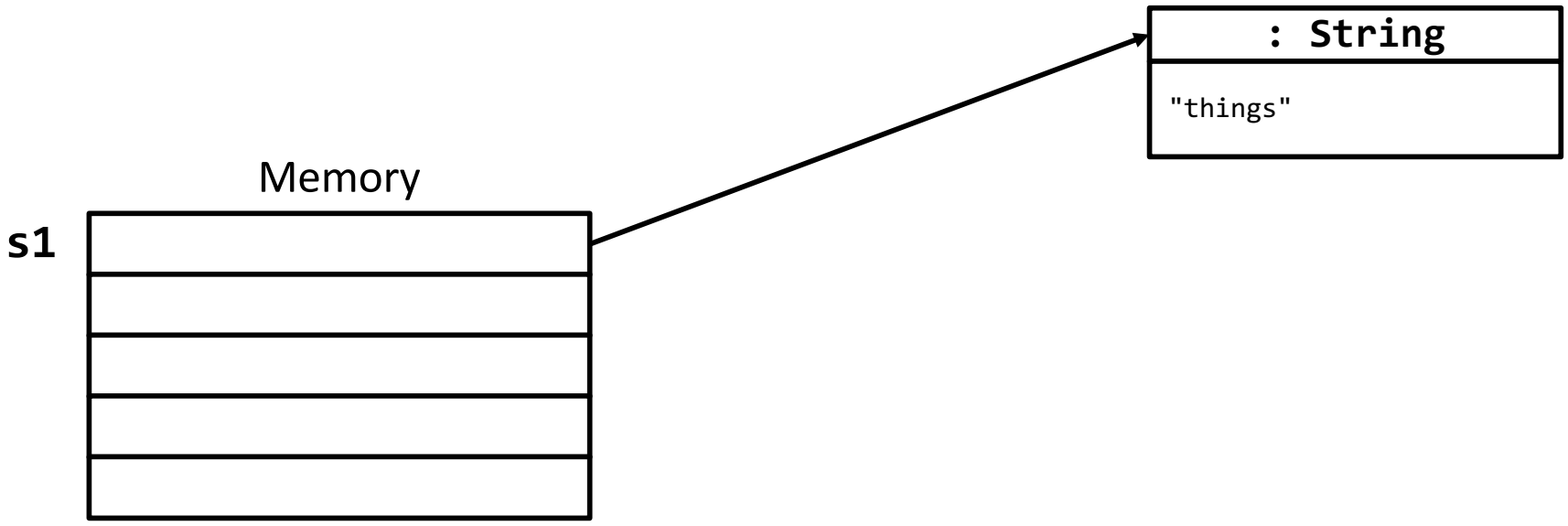
- Strings are *immutable* and are frequently used
- So to improve efficiency and save memory, the JVM uses a unique instance for string literals with the same character sequence
- Such an instance is called **interned**

Example

```
String s1 = "Things";
String s2 = new String("Things");
String s3 = "Things";
```

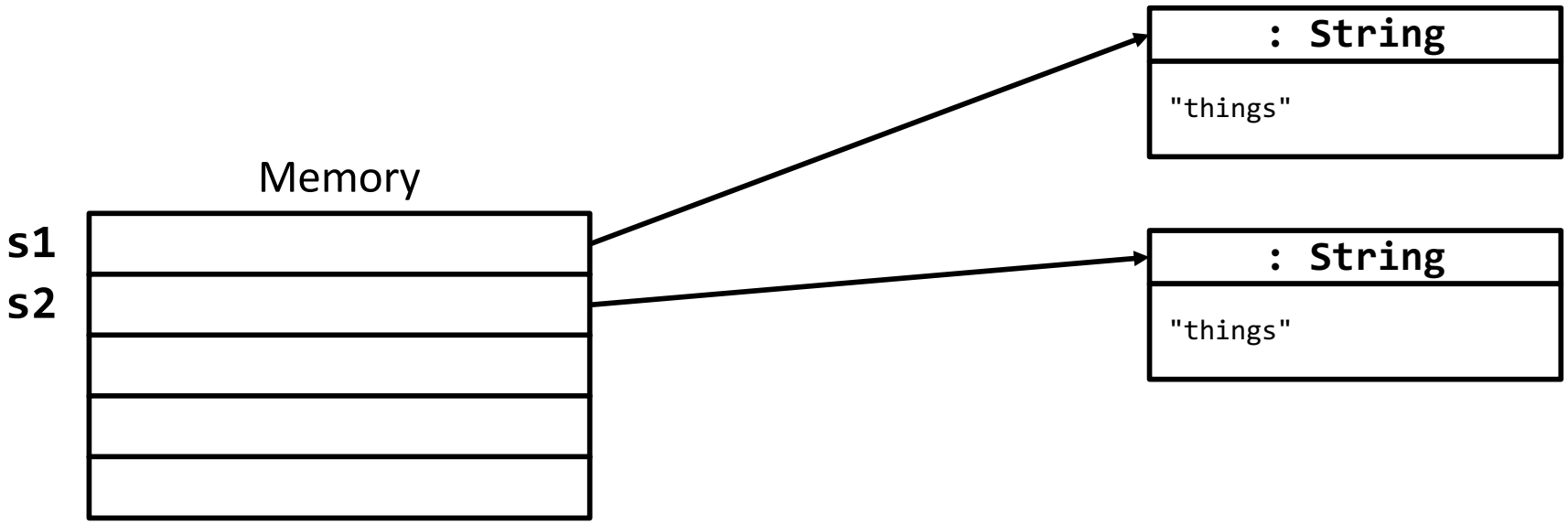
Trace Code (1)

```
String s1 = "Things";
String s2 = new String("Things");
String s3 = "Things";
```



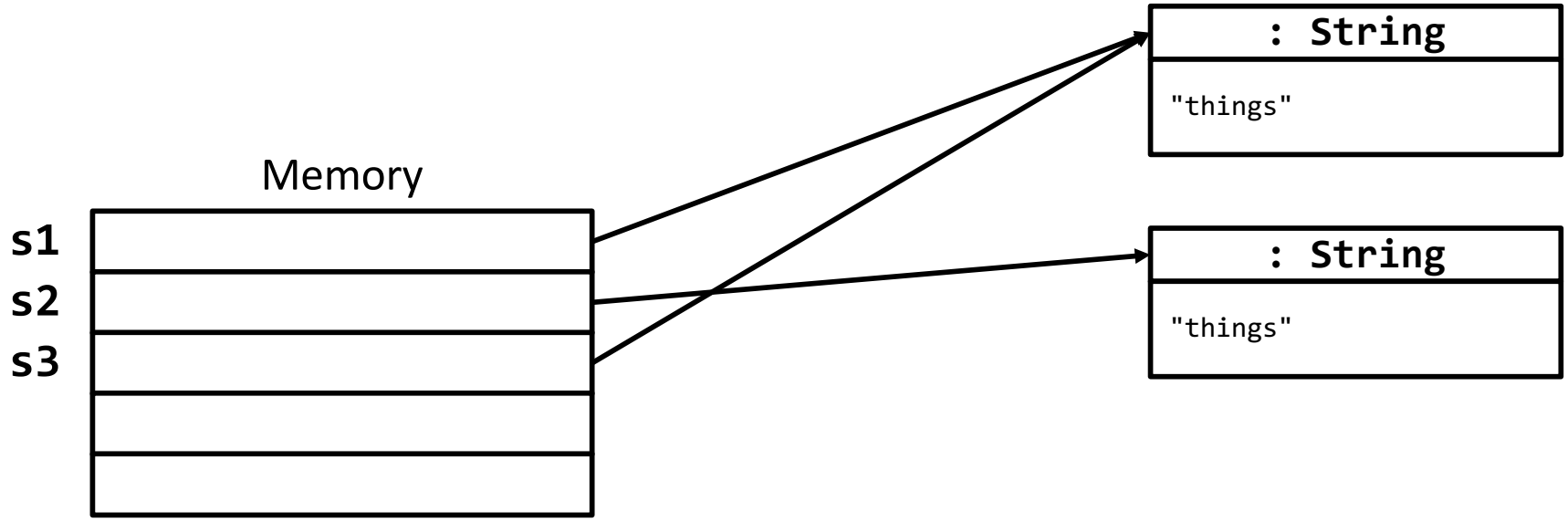
Trace Code (2)

```
String s1 = "Things";
String s2 = new String("Things");
String s3 = "Things";
```



Trace Code (3)

```
String s1 = "Things";
String s2 = new String("Things");
String s3 = "Things";
```



String Interning

- A new object is created if you use the **`new`** operator.

- When you use the string initializer (= **`""`**), no new object is created if the interned object already exists

Checkup

What is output to the terminal when the following code is executed?

```
String s1 = "Things";
String s2 = new String("Things");
String s3 = "Things";

System.out.printf("%b %b%n", s1==s2, s1.equals(s2));
System.out.printf("%b %b%n", s1==s3, s1.equals(s3));
System.out.printf("%b %b%n", s2==s3, s2.equals(s3));
```

Answer

```
false true
true true
false true
```

String Replacement

- **`replace(oldC: char, newC: char): String`**
  - Returns a new string that replaces all matching characters in this string with the new character
- **`replaceFirst(oldS: String, newS: String): String`**
  - Return a new string that replaces the first matching substring in this string with the new substring
- **`replaceAll(oldS: String, newS: String): String`**
  - Returns a new string that replaces all matching substrings in this string with a new substring

Exercise

What is output to the terminal when the following code is executed…

```
final String w = "Welcome";
System.out.printf("%s%n", w);
w.replace('e', 'E');
System.out.printf("%s%n", w);
```

Answer: Immutability!

```
Welcome
Welcome
```

Exercise

What is output to the terminal when the following code is executed…

```
System.out.printf("%s%n",
  "Welcome".replace('e', 'E'));
System.out.printf("%s%n",
  "Welcome".replaceFirst("e", "EE"));
System.out.printf("%s%n",
  "Welcome".replaceAll("e", "EE"));
System.out.printf("%s%n",
  "Welcome".replaceFirst("el", "EE"));
```

Answer

```
WElcomE
WEElcome
WEElcomEE
WEEcome
```

Splitting a String

`split(delimiter: String): String[]`

- Returns an array of strings consisting of the substrings split by the delimiter

```
for (String s : "Hello World".split(" ")) {
   System.out.printf("%s%n", s);
}
```

Regular Expressions

- A **regular expression** is a way of expressing a pattern of characters
  - Frequently used to test inputs, as well as search/replace string contents

- They are quite complex and flexible – you'll learn plenty about them in later classes
  - Here's just a taste…

Example RegEx's

- `"abc"`
  - Only matches "abc"
- `".*abc.*"`
  - Contains "abc"
  - `.` = any character, *=any number of times
- `"(abc)*"`
  - Either "" or "abc" or "abcabc" or …
- `"(abc)+"`
  - Either "abc" or "abcabc" or …
- `"[abc]"`
  - Either "a" or "b" or "c"
- `"[abc]+"`
  - A string composed of one or more a's, b's, and/or c's

Regular Expression Methods

- **`matches(regex: String): boolean`**
  - Returns true if the string matches the regular expression

```
"Java is fun".matches("Java.*"); // true
"Java is cool".matches("Java.*"); // true
```

- The **`split`**, **`replaceFirst`**, and **`replaceAll`** methods can also use regular expressions

Examples

```
System.out.printf(
   "a+b$#c".replaceAll("[$+#]", "NNN"));

for (String s : "a,b;c".split("[,;]")) {
   System.out.printf("%s%n", s);
}
```

Making Strings

- The **`String`** class is immutable, and so creating strings incrementally can be very inefficient (new instances are being created and thrown away)

- The **`StringBuilder`** class allows you to add/remove modify contents as you wish

Example (1)

```
StringBuilder sb = new StringBuilder();
sb.append("Welcome");
sb.append(' ');
sb.append("to");
sb.append(' ');
sb.append("Java!");

System.out.printf("%s%n", sb);
// Welcome to Java!
```

Example (2)

```
sb.insert(11, "HTML and "); // Welcome to HTML and Java!

sb.delete(8, 11); // Welcome HTML and Java!

sb.deleteCharAt(sb.length()-1); // Welcome HTML and Java

sb.reverse(); // avaJ dna LMTH emocleW

sb.reverse().replace(8, 16, "HTML"); // Welcome HTML Java

sb.setCharAt(0, 'w'); // welcome HTML Java
```

Exercise

Write a method that takes as a parameter an array of Strings and returns all the words concatenated into a single string using a **`StringBuilder`**

Answer

```
public static void main(String[] args) {
    String[] myWords = {"Dog","Cat","Fish","Bird","Horse"};
    System.out.printf("%s%n",makeSentence(myWords));
}

public static String makeSentence(String[] words) {
    StringBuffer sentence = new StringBuffer();
    for(String w: words){
        sentence.append(w);
    }
    return sentence.toString();
}
```

The `ArrayList` Class

- So far, when we wanted to store many values of the same type, we used an array
- However, we have seen that with arrays, we need to know the size ahead of time, and can't adjust later
- The `ArrayList` class contains an array, and supports array-like methods, but can grow and shrink as necessary
  - A great example of encapsulating complex behavior within a class

Creating an ArrayList

```
ArrayList<Type> a = new ArrayList<Type>();
```

- Like an array, when you create an **`ArrayList`**, you provide a data type for all elements, via the **`<Type>`**, which must be a class (note: wrapper classes come in handy!)
  - **`ArrayList`** is an example of a class that can be parameterized by a type, known as a **generic** class
- You must import **`ArrayList`** from **`java.util`**

Creating an Empty `ArrayList`

```
ArrayList<Integer> a = new ArrayList<Integer>();

ArrayList<Double> b = new ArrayList<Double>();

ArrayList<String> c = new ArrayList<String>();
```

Note: you can leave off the second <Type> if you wish...

```
ArrayList<Integer> a = new ArrayList<>();
```

`ArrayList` Size vs. Capacity

- Once an **`ArrayList`** is initialized, it is useful to think of it as encapsulating a partially filled array

- Two key properties:
  - **Size**: how many elements are in the list
    - Default constructor: 0; via **`size()`** and **`isEmpty()`**
  - **Capacity**: the size of the internal array
    - Default constructor: 10; not accessible

Adding Elements

- Add an element to the end of the list via the add method

```
ArrayList<Integer> a = new
   ArrayList<>();

System.out.println(a.size()); // 0
a.add(3);
a.add(1);
a.add(4);
System.out.println(a.size()); // 3
```

Resizing Behavior

- Whenever an element is added such that the new size would exceed the capacity of the underlying array, the **`ArrayList`** automatically resizes to accommodate, and copies old data

- Basic idea:

```
newArray = new Type[newSize];
for (int i=0; i<oldSize; i++)
  newArray[i] = listArray[i];
listArray = newArray;
```

Resizing Efficiently

- Copying arrays can become computationally expensive

- If you are about to add many elements, use the **`ensureCapacity(minSize)`** method to have the **`ArrayList`** resize to a desired capacity
  - Note: there is also a constructor that can set the initial capacity

Getting/Setting Elements

- To access the value of an existing element, use the **`get(index)`** method

- To change the value of an existing element, use the **`set(index,value)`** method

- For both methods, an **`IndexOutOfBoundsException`** is thrown if… **`(index < 0 || index >= size())`**

Example

```
ArrayList<Character> a = new ArrayList<>();


a.add('h');
a.add('i');
a.add('j');
a.set(2, '!');

System.out.printf("%c%c%c%n",
   a.get(0), a.get(1), a.get(2));
```

Removing Elements

- You can erase elements from the list via the remove and clear methods

```
clear()
remove(index)
remove(value)
```

  - Only first occurrence; returns true if list changed

- Note: removal requires copying all elements that *follow* the removed index, and can thus be slow in large lists

Shrinking the List

The **`trimtoSize()`** method reduces the capacity of the **`ArrayList`** to the current size, thereby saving memory

```
list.clear() // size = 0, capacity = ?
list.trimToSize() // capacity = 0
```

Example

```
public static void printList(ArrayList<Character> l) {
     for (Character c : l) {
          System.out.printf("%c",c);
     }
     System.out.printf("%n");
}

public static void removeAll(ArrayList<Character> l, Character c) {
     while (l.remove(c));
}

public static void main(String[] args) {
     ArrayList<Character> word = new ArrayList<>();
     word.add('h');
     word.add('e');
     word.add('l');
     word.add('l');
     word.add('o');
     printList(word); // hello (size=5, capacity=10)
     removeAll(word, 'l');
     word.trimToSize();
     printList(word);     // heo (size=3, capacity=3)
}
```

Take Home Points

- Class **abstraction** means to separate class implementation from the use of the class – **encapsulate** functionality and hide unnecessary details from users
- The wrapper and **`String`** classes have many useful methods and are all **immutable**
  - Note: String literals are **interned** automatically for reasons of efficiency
- To improve performance, the **`StringBuilder`** class should be used when there is a need to modify a string
- The **`ArrayList`** class has useful methods to allow you to grow and shrink an array of elements