Value vs. reference semantics
Recall: Value semantics

- **value semantics**: Behavior where variables are copied when assigned to each other or passed as parameters.
  - Primitive types in Java use value semantics.
  - Modifying the value of one variable does not affect the other.

**Example:**

```java
int x = 5;
int y = x;  // x = 5, y = 5
y = 17;    // x = 5, y = 17
x = 8;     // x = 8, y = 17
```
Reference semantics

- **reference semantics**: Behavior where variables refer to a common value when assigned to each other or passed as parameters.
  - Object types in Java use reference semantics.
  - Object variables do not store an object; they store the *address* of an object's location in the computer memory. We graphically represent addresses as arrows.

- **Example:**

  ```java
  Point p1 = new Point(3, 8);
  ```

  ![Diagram of point p1 with x: 3 and y: 8]
Reference semantics

- If two object variables are assigned the same object, the object is **NOT** copied; instead, the object’s address is copied.
  - As a result, both variables will *point* to the same object.
  - Calling a method on either variable will modify the same object.

Example:

```java
Point p1 = new Point(3, 8);
Point p2 = p1;
p2.setLocation(1, 2);
```
Objects have reference semantics for several reasons:

- **efficiency**: Objects can be large and bulky. Having to copy them every time they are passed as parameters would slow down the program.

- **sharing**: Since objects hold important state, it is often more desirable for them to be shared by parts of the program when they're passed as parameters. Often we want the changes to occur to the same object.
Reference semantics: Example

Point p1 = new Point(3, 8);
Point p2 = new Point(2, -4);
Point p3 = p2;

- How many unique objects are there? How do you know that?
  - Two, because objects are only created with the `new` keyword.

- If we change p3, will p2 be affected and vice versa?
  - Yes.

![Diagram](x: 3 y: 8)

![Diagram](x: 2 y: -4)
If two variables refer to the same object, modifying one of them will also make a change in the other:

```java
p3.translate(5, 1);
System.out.println("(" + p2.x + " " + p2.y + ")");
```

Output: 

```
(7, -3)
```
Objects as parameters

- When an object is passed as a parameter, the object is *not* copied. The same object is referred to by both the original variable and the method's parameter.
  - If a method is called on the parameter, it *will* affect the original object that was passed to the method.
Objects as parameters: Example

Example:

```java
public static void main(String[] args) {
    Point p1 = new Point(2, 3);
    move(p1);
}

public static void move(Point p) {
    p.setLocation(-1, -2);
}
```

```
  p1
   ↓
  □ □

  p
   ↓
  □ □
```

```
  x: -2  y: -2
```

```java
p1 y
  ↓
  □ □

p x
  ↓
  □ □
```
What does this code do?

```java
public static void main(String[] args) {
    int a = 7;
    int b = 35;
    System.out.println(a + " "+ b);

    int temp = a;
    a = b;
    b = temp;

    System.out.println(a + " "+ b);
}
```
Swapping values

- Swapping is a common operation, so we might want to make it into a method.

```java
public static void main(String[] args) {
    int a = 7;
    int b = 35;
    System.out.println(a + " " + b);

    // swap a with b
    swap(a, b);

    System.out.println(a + " " + b);
}

public static void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

- Does this work? Why or why not?
public static void main(String[] args) {
    Point p1 = new Point(1, -1);
    Point p2 = new Point(2, 0);
    System.out.println(p1 + " " + p2);

    // swap a with b
    swap(p1, p2);
    System.out.println(p1 + " " + p2);
}

public static void swap(Point a, Point b) {
    Point temp = a;
    a = b;
    b = temp;
}

Does this work? Why or why not?
public static void main(String[] args) {
    Point p1 = new Point(1, -1);
    Point p2 = new Point(2, 0);
    System.out.println(p1 + " " + p2);

    // swap a with b
    swap(p1, p2);

    System.out.println(p1 + " " + p2);
}

public static void swap(Point a, Point b) {
    Point temp = new Point();
    temp.setLocation(a.getX(), a.getY());
    a.setLocation(b.getX(), b.getY());
    b.setLocation(temp.getX(), temp.getY());
}
Wrapper Classes
Wrapper classes

Wrapper classes are object types that correspond to each primitive type

<table>
<thead>
<tr>
<th>Primitive Type</th>
<th>Wrapper Class</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Integer</td>
</tr>
<tr>
<td>long</td>
<td>Long</td>
</tr>
<tr>
<td>double</td>
<td>Double</td>
</tr>
<tr>
<td>boolean</td>
<td>Boolean</td>
</tr>
<tr>
<td>char</td>
<td>Character</td>
</tr>
</tbody>
</table>

Two important methods in Wrapper classes:
- `parseInt(String s)` (or `parseDouble`, `parseLong` ...)
- `intValue()` (or `doubleValue()`, `longValue()`, ...)

Important fields: `MAX_VALUE`, `MIN_VALUE`
Converting between primitive and Wrapper types

- You can wrap a int in an Integer by assignment:

```java
int x = 17;
Integer wrapX = new Integer(x);
Integer wrapX2 = x; // shorthand
```

- To get an int value out of an Integer object:

```java
Integer x = new Integer(17);
int unwrappedX = x.intValue();
int unwrappedX2 = x; // shorthand
```
Wrapper classes for swapping ints

```java
public static void main(String[] args) {
    Integer a = 7;
    Integer b = 35;
    System.out.println(a + " " + b);

    // swap a with b
    swap(a, b);

    System.out.println(a + " " + b);
}

public static void swap(Integer a, Integer b) {
    Integer temp = a;
    a = b;
    b = temp;
}
```

- What’s wrong with this version?
Mutable and Immutable objects

- Definition: An **immutable** object is an object whose **state cannot be modified** after it is created.

- The **String** class and all the wrapper classes are **immutable** in Java.
  - As a result, we can’t use the wrapper classes to solve the swap problem for primitive types.
  - We’ll come back to swapping primitive objects later.

- Point objects are **mutable** (ie, not immutable)
  - setLocation and translate change their state
  - We can use this fact to make the swap method work
Madness to the method

- The methods that appear to modify a string (substring, toLowerCase, toUpperCase, etc.) actually create and return a new string.

```java
String s = "lil bow wow";
s.toUpperCase();
System.out.println(s);   // output: lil bow wow

vs.

String s = "lil bow wow";
s = s.toUpperCase();
System.out.println(s);   // output: LIL BOW WOW
```
Exercises

- Write a method to determine the slope of the line that passes between two points.

- Write a method to compute the dot-product (or inner product) of two points.

- Write a method to return a point’s mirror image across the x axis.

- Given an int called scale, write a method to magnify a point so that its distance from the origin is multiplied by scale.
Operators and object types
How not test equality of objects

- **DO NOT DO THIS:**

```java
public static boolean testEquals(Point p1, Point p2) {
    if(p1==p2) {
        return true;
    } else {
        return false;
    }
}
```
How not test equality of objects

- Objects store references, or addresses.
- Comparing two objects with == will test if they have the same address.
- This is NOT what you want.

```java
public static boolean testEquals(Point p1, Point p2) {
    if (p1 == p2) {
        return true;
    } else {
        return false;
    }
}
```

BAD
Point p1 = new Point(7, 2);
Point p2 = new Point(7, 2);

Is \( p1 == p2 \) true or false?
It’s false: they point to different spots in memory. So they store different addresses. But we want it to be true, obviously!
The `equals` method

- All classes in Java have a built-in `equals` method

- For the `Point` class:

  ```java
  p1.equals(p2)
  ```

  This returns true if `p1` and `p2` have the same data

- It doesn’t matter if they reference different locations in memory

- Likewise, use `!p1.equals(p2)` instead of `p1!=p2`
Object and primitive types: a comparison (so far)

<table>
<thead>
<tr>
<th>Object types</th>
<th>Primitive types</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructed with the new keyword</td>
<td>Values don’t need to be constructed</td>
</tr>
<tr>
<td>References to memory location that stores their data</td>
<td>Store data directly in memory slot</td>
</tr>
<tr>
<td>Can be null (have no data)</td>
<td>Cannot be null</td>
</tr>
<tr>
<td>Can cause NullPointerExceptions</td>
<td>Cannot cause NullPointerExceptions</td>
</tr>
<tr>
<td>Contain state and behavior</td>
<td>Contain state only</td>
</tr>
<tr>
<td>Use reference semantics</td>
<td>Use value semantics</td>
</tr>
<tr>
<td>Use equals() method</td>
<td>Use ==, !=</td>
</tr>
</tbody>
</table>
String and char methods
Text processing: Example

// Returns the count of occurrences of c in s.
public static int count(String s, char c) {
    int count = 0;
    for (int i = 0; i < s.length(); i++) {
        if (s.charAt(i) == c) {
            count++;
        }
    }
    return count;
}

For instance, count("mississippi", 'i') returns 4
Strings and chars: Exercises

- Write a method named `pigLatinWord` that accepts a `String` as a parameter and outputs that word in simplified Pig Latin, by placing the word's first letter at the end followed by the suffix `ay`.
  - `pigLatinWord("hello")` prints `ello-hay`
  - `pigLatinWord("goodbye")` prints `oodbye-gay`

- Write methods named `encode` and `decode` that accept a `String` as a parameter and outputs that `String` with each of its letters increased or decreased by 1.
  - `encode("hello")` prints `ifmmp`
  - `decode("ifmmp")` prints `hello`
Strings and chars: Exercises

- Write a method `printName` that accepts a full name as a parameter, and prints the last name followed by a comma, followed by the first name and middle initial.

```java
printName("Alexander Pieter Yates");
```

would output:

Yates, Alexander P.
printName: One possible solution

public static void printName(String fullName) {  
  int firstBlankIndex = fullName.indexOf(" ");
  String upToMiddleInitial = fullName.substring(0, firstBlankIndex + 2);

  String middleAndLastName = fullName.substring(firstBlankIndex + 1,
                                             fullName.length());
  int secondBlankIndex = middleAndLastName.indexOf(" ");

  // Notice that "secondBlankIndex" is used with "middleAndLastName" and NOT
  // "fullName". If you said
  //
  // fullName.substring(secondBlankIndex + 1, fullName.length())
  //
  // you wouldn't get the last name properly. Make sure you understand
  // why.
  String lastName = middleAndLastName.substring(secondBlankIndex + 1,
                                             middleAndLastName.length());

  System.out.println(lastName + ", " + upToMiddleInitial + ".");
}
Exercise: String Conversions

- Write a method that has a String argument that contains an int (eg, “123”) and returns a String containing the next number (eg, “124”).

- Write a method that converts a String to an int without using Integer.parseInt(). Use a loop!