Primitive vs Reference

Primitive types store **values**

Reference types store **addresses**

This is the fundamental difference between the 2

Why is that important?

Because a reference type stores an **address,** you can do things like create **aliases** to variables. Two variables could refer to the same address and thus access the same values.

This also allows us to make changes to input parameters to methods.

== with references

Whenever you use the == operator to compare two variables, you are always comparing the values stored in the variables.

However, because references store addresses, the values are actually addresses.

So if x and y are references, x==y is true only if x and y store the same **address**

== with references

This is why when comparing two *Strings*, we must use .equals() method instead.

With arrays, we must do the same thing.

Java: Passing arguments by value

In Java, any time you call a method, it passes the input to the method *by value*

This means that it evaluates the value of whatever expression you give and assigns the formal parameters of a method those values

This is true with both primitive and reference types. BUT with reference types the value is an address!

Arrays are references

Arrays are references

This means that the variables stores an **address** not a value.

Because of this, if I change the values stored at an address in a method, for example, the values “in” the original array also change

Example: Reversing the order of an array

Let's write a method to reverse the order of an array. The method should modify an existing array to which a reference will be passed.

What if I wanted to do this in a method?

(Bad) idea 1: Make a duplicate array and copy values into it.

```
public static void reverseArray(double[] a) {
   double[] tempArray = new double[a.length];
   for (int i=0; i < a.length; i++) {
         tempArray[a.length - i - 1] = a[i];
    }
   a = tempArray;
}
```

What if I wanted to do this in a method?

The problem is I never change that **values** stored at whatever address the variable *a* originally stores.

At the end I assign *a* to store a different address, but remember this is only changing a temporary variable within the method

What if I wanted to do this in a method?

To properly do this, I should change the values at the address originally stored by the variable a.

For example:

What if I wanted to do this in a method?

```
public static void reverseArray(double[] a){
   double[] temp = new double[a.length];
   //first copy into temp array in reverse
   //order
   for (int i=0; i < a.length; i++) {
     temp[a.length -1 -i] = a[i];
   }
   //now copy back into original array
   for (int i=0; i < a.length; i++) {
      a[i] = temp[i];
   }
}
```

Example: Writing a method to delete the i^th element from an array of chars

For starters, we can not do this by modifying the input parameter

Why not?

Example: Writing a method to delete the i^th element from an array of chars

Idea: Make a 2nd array which has size 1 less than the original array

For every element with index 0 to i-1, copy the values into the same index

For every element from index i+1 to the end, copy the values into the index one less

Return the resulting array.

Example: Writing a method to delete the i^th element

```
public static char[] deleteItem(char[] a, int i){
   char[] returnVal = new char[a.length -1];


   for (int j=0; j < i; j++) {
     returnVal[j] = a[j];
   }


  for (int j=i+1; j < a.length; j++) {
      returnVal[j-1] = a[j];
   }
   return returnVal;
}
```

Example: Writing a method to search for an element

What if we wanted to write a method to search for a specific element and return the **first** index where it is found?

Example: Writing a method to search for an element

```
public static int searchItem(char[] a, char
target){
   for (int j=0; j < a.length; j++) {
     if (a[j] == target) {
         return j;
     }
   }
  return -1; //“code” we use for not found
 }
```

What if we want to **delete** all elements from an array of a certain value?

Idea: We have already written a method to **search** for an element of a certain value

We have also written a method to delete an element at a specific location

What we can do is write a method that does the following:
1)Search for the element to delete
2)If found, delete the first occurrence and go back to step 1
3)If not found then we have finished.

Q: Should we **return** the array or modify via parameters?

What if we want to **delete** all elements from an array of a certain value?

```
public char[] deleteAllMatches(char[] a, char target) {
  char[] returnVal = a;
  int i = searchItem(a, target);

  while (i != -1) {
     returnVal = deleteItem(returnVal, i);
     i = searchItem(a, target);
  }
  return returnVal;
}
```

Ex: Converting an array of chars to lower case.

Idea: Recall that all chars have a particular encoding. We don't need to know the exact values, but remember that all letters are consecutive in the chart

Capitol letters come first. Then a few after that are the lower case letters.

Ex: Converting an array of chars to lower case.

Can we do this by modifying the original array as input argument

or do we need to return a new array?

Ex: Converting an array of chars to lower case.

```
public static void toLowerCase(char[] charArray) {
    for (int i=0; i < charArray.length; i++) {
        if (charArray[i] >= 'A' && charArray[i] <= 'Z') {
            charArray[i] += 'a' – 'A'; //the appropriate gap
        }
    }
}
```

Ex: Converting a string to lowercase

Converting a string to lower case is similar, but not the same.

You can use the .charAt() method to get the ith element.

However, you can't **set** the string to a different value using the .charAt() method. Thus we have to return a modified string.

Ex: Converting a string to lowercase

```
public String toLowerCase(String original) {
   String returnString = “”;

    for (int i=0; i < original.length(); i++) {
        if (original.charAt(i) >= 'A' && original.charAt(i) <= 'Z') {
              returnString += original.charAt(i) + ('a' – 'A');
        }
        else {
              returnString += original.charAt(i);
        }
    }
}
```

Keeping track of variables

A good way to keep track of variables, especially with references, is by making a table of all the variables in memory at the moment. Within this table, you can draw “arrows” to represent the values in reference types.

Java actually does something similar!

Your table should be a list of all variables and should include the following information:

Variable name, Variable type, The method the variableis in, Variable value

```
public class May 31{
   public static void main(String[] args) {
          twoDArrays();
          reviewReferenceTypes();
          Examples[] examples = new
            Examples[3];
          creatingYourOwnObjects();
   }
}
```

An array of arrays

Since arrays are an ordered collection of elements of the same type —and can be of any type---you can make an array of arrays.

This is known as a 2-dimensional array.

An array of arrays

To declare an array of type *t,* you write:

t[] *name;*

To initialize an array of type *t*, you can write:
t[] *name* = { *list of elements of type t* };

An array of arrays

So it shouldn't be surprising that to make an array of, for example, double arrays, we can do

```
double[][] doubleArray;
```

or

```
double[][] doubleArray = { {1,2,3}, {4,5,6}, {7,8,9,10} };
```

An array of arrays

One often calls an array of arrays a "2dimensional array"

We can use them the same way as before, except we now use 2 indices.

For example:

```
 double[][] doubleArray = { {1,2,3}, {4,5,6}, {7,8,9,10} };
```

Now doubleArray[0] refers to the first array stored at the address of doubleArray.

This means that I can refer to an index of doubleArray!

i.e. doubleArray[0][1] refers to the element at index 1 of the first array stored at the address stored in dobuleArray.

Creating a 2d array.

As with 1d arrays, there are 2 ways you can set a 2d array.

The first is by making an initializer list (of arrays!) when you declare the variable. (This is what we did on the previous slide)

The second is by using the *new* keyword.

There are actually 2 ways to use the *new* keyword.

Creating a rectangular 2d array.

To create a rectangular 2d array, you can simply do:

double[][] doubleArray;
......
doubleArray = new double[*size1*][*size2*] ;

This will make *size1* arrays of *size2*.

You can then set the values by referring to the 2 indices:

i.e. doubleArray[2][3] = 4.0;

Creating a “jagged” 2d array.

If you don't want a rectangular array, you can create each “subarray” separately.

For example:

```
double[][] doubleArray;
doubleArray = new double[size][] ;
//the above makes an array that will hold size arrays of doubles
//each of these arrays can have any size

//Then later you would do:
doubleArray[0] = new double[size2];
doubleArray[1] = new double[size3];
```

Writing equals on a 2d array

```
public static boolean array2dEquals(String[][] a1, String[][] a2) {
    if (a1.length != a2.length) {
        return false;
    }
    for (int i=0; i < a1.length; i++) {
        if (a1[i].length != a2[i].length) {
            return false;
        }
        for (int j=0; j < a1[i].length; j++ ) {
            if (!a1[i][j].equals(a2[i][j])) {
                return false;
            }
        }
    }
    return true;
}
```

Writing equals on a 2d array

```
public static boolean array2dEquals(String[][] a1, String[][] a2) {
    if (a1.length != a2.length) {
        return false;
    }
    for (int i=0; i < a1.length; i++) {
        if (a1[i].length != a2[i].length) {
            return false;
        }
        for (int j=0; j < a1[i].length; j++ ) {
            if (!a1.equals(a2[i][j])) { //careful not to do a1 != a2
                return false;
            }
        }
    }
    return true;
}
```

Objects in Java

-In Java, any data type that you can store a reference to is an *Object*

References, by definition, store the address of an *Object*

Which of the following variables could store the address of an Object?

-int x;

-double d;

-String s;

-int[] foo;

-Flamingo f;

-Pizza[][][][][][] r;

Which of the following variables could store the address of an Object?

-int x;

-double d;

-String s;

-int[] foo;

-Flamingo f;

-Pizza[][][][][][] r;

All of these are reference variables.

Remember there are exactly 8 primitives

Very important to understand:

-In the previous slides, s, foo, f, and r are NOT Objects. They are reference to Objects.

We sometimes will say “the String s” or “The int array foo” but really we mean the “Reference to a String s” or “The reference to the int array foo”

Strings are immutable

A String in Java is *immutable* . This means that it can not be changed.

But then, why are we able to write the following:

```
String s = “first value”;
s = “second value”;
```

Strings are immutable

Answer: A String is immutable in Java, but a *reference* to a String is not.

When I write s = “second value”;
I am creating a brand new String and assigning the address of the new String to the variable s.

“Who cares whether Strings are immutable? I wish you were mutable!”

Think about what must happen when we write something like:

String s = “hello”;
s = s + “adding more stuff”

In memory, a brand new String is created and “hello” is copied into it

“Who cares whether Strings are immutable? I wish you were mutable!”

-Then this is added to the String “adding more stuff” . This is SSSLLLLLOOOOOOOOWWWWWW

“Who cares whether Strings are immutable? I wish you were mutable!”

-This means if you had a program where you needed to add things to Strings a lot, you might be inclined to use a different approach. For example, you might create a char[] of a very large size and use a variable to keep track of what part of the char[] is “valid”

Then when you want to append to the “String” you don't need to copy everything.

Example: Changing the size of an array

-You can not change the size of an array once it is created.

-You can create a different array and store a different array in the same variable.

-The reasoning is the same as the example with String.

Example: Final with reference type

-Remember that a variable declared to be final can not change it's value.

```
final double PI = 3.14159;
PI = 2.71828; ----> Compiler error!
```

Example: Final with reference type

-What about using the keyword final with a reference type?

```
final int[] foo = {1,2,3,4,5};
foo[3] = 4;
```

Will the above compile?

Example: Final with reference type

-What about using the keyword final with a reference type?

```
final int[] foo = {1,2,3,4,5};
foo[3] = 4;
```

Will the above compile? Yes, because you are not changing the variable foo. foo still refers to the same address. The object has changed, not foo

Example: Final with reference type

-What about using the keyword final with a reference type?

```
final int[] foo = {1,2,3,4,5};
foo = new int[10];
```

Will the above compile?

Example: Final with reference type

-What about using the keyword final with a reference type?

```
final int[] foo = {1,2,3,4,5};
foo = new int[10];
```

Will the above compile? No. Here we are changing the value of the reference variable.

Example: Final with String

-What is the significance of using final with a String?

final String s = “hello”;

-We know that Strings are immutable (they can't change)

-We now made it so that s ALWAYS stores the same address.

DEAD
END

Creating Objects (1)

- Declaring a reference variable is a separate operation from creating an object whose address in memory will be stored in that reference variable
- In general, we use the `new` operator to create an object:

```
      numbers = new int[10];
```

- There are two situations where we create objects without using the `new` operator:
  - When we use array initializer lists
  - The first time we use a `String` literal, a `String` object representing this literal is automatically created

Creating Objects (2)

- When we create an object using the `new` operator, additional memory cells are allocated
  - The object is stored in these additional memory cells
  - These memory cells are *not* the same as the memory cells allocated when reference variables are declared
  - The address of the memory cells in which the object is stored will be stored in a reference variable

What is different about an Object from a primitive type?

In Java, Objects have 2 additional things that primitive types don't have:

1)Attributes: These are properties that are associated with a specific Object. Every Object will have it's own set of attributes.

2)Behaviors: These are *methods* that are associated with a specific Object. These methods have access to the attributes that are specific to that Object.

What is different about an Object from a primitive type?

You can access an attribute on an object by writing:

referencevariablename.attributename

Attributes

The first attribute we have seen of an Object is the length attribute of an array.

```
int[] foo = new int[10];
System.out.println(foo.length)
```

foo is the reference variable name

length is the name of the attribute

What is different about an Object from a primitive type?

You can call a behavior on an object by writing:

referencevariablename.behaviorname(input to method)

Behaviors

The first behaviors we have seen of an Object is the length and charAt behaviors of a String

```
String s = “yo yo waz up”;
System.out.println(s.length());
System.out.println(s.charAt(3))
```

s is the reference variable name

length / charAt are the names of the behavior

We know it's a behavior not an attribute because of the ()

The `null` Literal Value

- Sometimes, we want a reference variable to contain a special value to indicate that the variable intentionally does not contain the address in memory of a valid object
- The literal value `null` can be used for this purpose
- A reference variable containing the value `null` is sometimes said to "point nowhere"
  - In memory diagrams, `null` is often represented with a ground symbol

Using `null`

- One can assign `null` to a reference variable like one assigns an `int` literal like `42` to a variable of type `int`

```
    int[] a;
    a1 = null;
```

- One can check whether or not a reference variable contains the value `null` like one checks whether or not a variable of type `int` contains any value represented by an `int` literal like `42`

```
    if (a == null) {
       // do something
    } else {
       // do something else
    }
```

`null`-Related Caveats

- The address stored in a reference variable is always either the address in memory of a valid object of that variable's type, or `null`
  - In Java, you cannot store an arbitrary memory address in a reference variable, nor manipulate memory addresses directly
- If you attempt to use access an object using a reference variable which contains `null`, your program will crash:

```
    int[] a = null;
    a[0] = 1; // a1 contains null: crash!
```

- When this occurs, the error message will mention that the program threw a `NullPointerException`
  - The error message should also mention which line in your program caused the latter to crash

NullPointerDemo.java

```java
public class NullPointerDemo {
  public static void main(String[] args) {
    int[] a = null;

    System.out.println("Attempting to retrieve the " +
      "element stored at index 0 of an array through a " +
      "null reference variable...");
    System.out.println("The value stored at index 0 of " +
      "this array is: " + a[0]);
  }
}
```

What will happen if we run this program?

Defining your own types!

- So far, we have worked with types that are already defined for us.
-
- In the following several classes, we will discuss ways to work with objects that **we** define ourselves!

- Our types can consist of many different other types as variables.
-
- In a sense, they are just composite types.
-

- For example, if I wanted to define the type Human, I could decide that a Human consists of :
-
- 1 name
- 2 legs
- 2 arms
- 1 torso
- 1 head (unless it's Zaphod Beeblebrox)
- 2 feet
-

- Of course, if we wanted to do this in Java, we would have to define each of the types, Leg, Arm, Torso, Head, etc.
-
- Head could consist of
-
- Brain, Eyes, Ears, Nose, Mouth

- The brain stores lots of information, so it may consist of
-
- int age
- String name
- String[] friendsList
- String[] knowledgeList
- Neuron[] allNeurons
-
- etc.
-

- For starters, to make your own type in Java, you create a class with the name of the type you want to create.
-
- For example:
-
- public class Person {
-
-
- }
-

- Inside the class, you list all the other variables you want to store and give them names
-

```
public class Person {
        string name;
        private Arm leftArm;
        private Arm rightArm;
        public Brain brain;
          ....
}
```

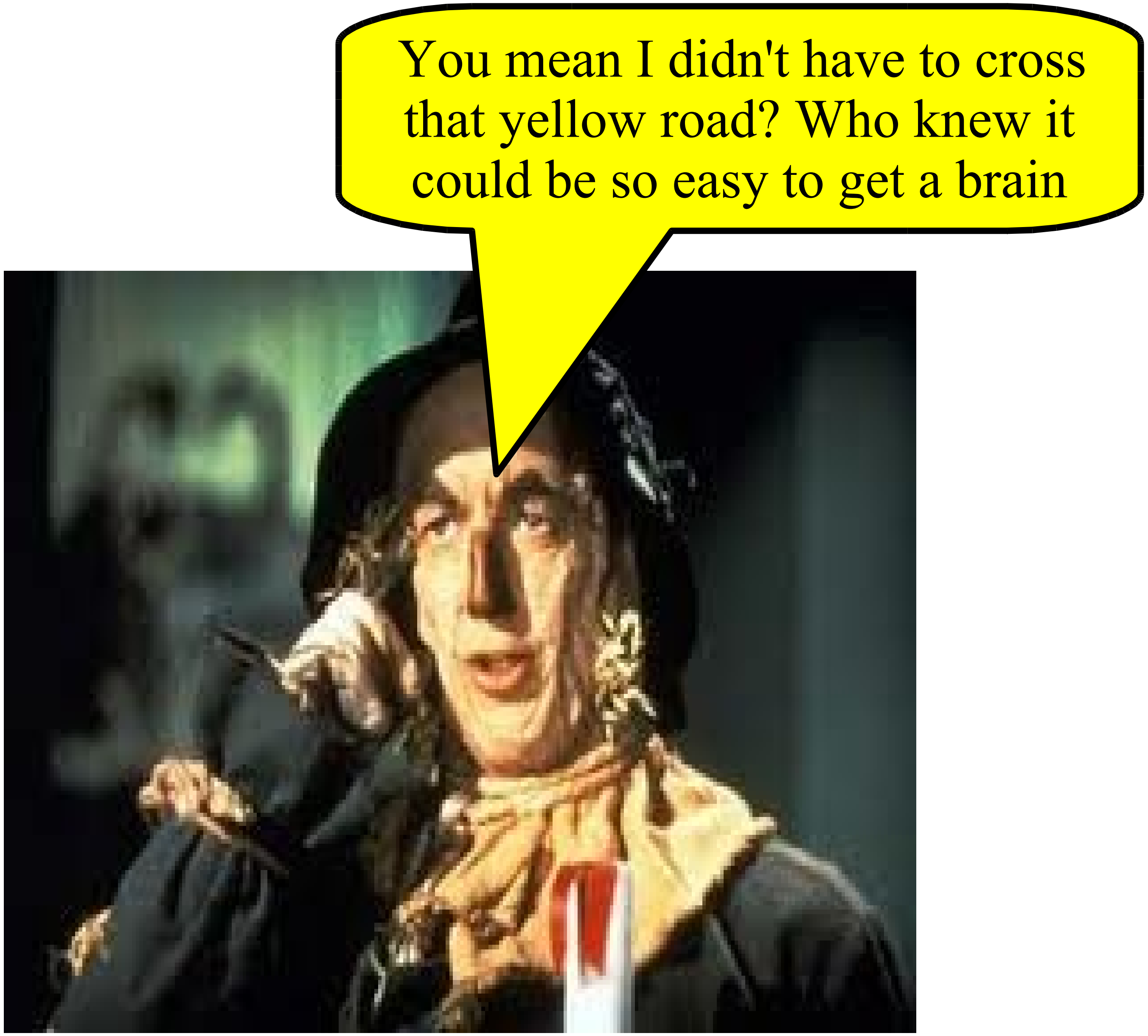
- In addition to storing **attributes**, your type can perform **behaviors** which will (usually) use or modify these attributes.
-
- For example, a person may have a behavior called “wave” which performs an action on his leftArm
-

- These *behaviors* are created using a method.
-
- Note that these methods are similar to before, but they do **NOT** have the word static before them.

```
• public class Person {
•         String name = “Dan”;
•         private Arm leftArm;
•         private Arm rightArm;
•         public Brain brain;
•
•         public void introduceSelf() {
•                 System.out.println(“Hi, my name is
  ” + name + “ . It is a pleasure to meet you”);
•          }
• }
```

- Now that you have created this type, you can use it in any other class the way we make other objects/variables. Note that your class will be a reference type.
-
- 1)Declare a variable of type Person
- 2)Create a new *instance* of the variable using the new keyword.
- 3)You can now access some of its attributes and behaviors.

```
• public class PersonDemo {
•       public static void main(Strings[] args) {
•             //declare a variable and create a
  person
•             Person scarecrow = new Person();
•
•             //call the behaviour introduceSelf()
•             p.introduceSelf();
•             //access the member variable brain
•             // and create a new brain
•             scarecrow.brain = new Brain();
```


You mean I didn't have to cross that yellow road? Who knew it could be so easy to get a brain

- Public vs. Private:
-
- Whenever you write the modifier *public* before either an attribute or behavior, it means that that attribute is *accessible* from a different class or instance.
-
- In other words, you can create an instances of the type, and then refer to the attribute or behavior via the . operator

```
public class Person {
    public String name;
    public int weight;

    public void introduceSelf() {
        System.out.println(“Hi. I'm ” + name);
    }

}
```

Now, somewhere else in your program (probably in a different file), you could have:

```
Person p = new Person();
p.name = “Dan”;

p.introduceSelf();
```

Now, somewhere else in your program (probably in a different file), you could have:

```
Person p = new Person(); /*create instance of
    person. Since a Person has to store both a
    String and an int, it will create a String and
    int variable corresponding to the new object
    */
p.name = “Dan”;


p.introduceSelf();
```

Now, somewhere else in your program (probably in a different file), you could have:

```
Person p = new Person();
p.name = “Dan”; /*p now refers to a specific
  Person which has a name and a weight. This
  says get the name associated with p and set it
  equal to Dan */

p.introduceSelf();
```

Now, somewhere else in your program (probably in a different file), you could have:

```
Person p = new Person();
p.name = “Dan”;

p.introduceSelf(); /*Since p is a person, it has
   a behaviour called introduceSelf() This calls
   the method introduceSelf with the object p---
   meaning that inside introduceSelf() name
   will equal p's name and weight will equal p's
   weight */
```

- Public vs. Private:
-
- If an attribute or behavior is *private* then we can only use if from the specific instance of the class. (Note: if you don't write anything then private is the default)

```
public class Person {
    public String name;
    int weight;

    public void introduceSelf() {
        System.out.println(“Hi. I'm ” + name);
    }

}
```

Now, somewhere else in your program (probably in a different file), you could have:

```
Person p = new Person();
p.name = “Dan”;

p.introduceSelf();
System.out.println(p.name + “'s weight is” +
  p.weight);
```

Now, somewhere else in your program (probably in a different file), you could have:

```
Person p = new Person();
p.name = “Dan”;

p.introduceSelf();
System.out.println(p.name + “'s weight is” +
    p.weight); /*ERROR! weight is a private
    field */
```

It is usually a good idea to make all your instance variables *private* rather than public (We'll talk about why later this class)

However, many times we need to get and set the values of these variables. To do this, we will write (usually) very short methods which are known as *getters* and *setters*

A *getter* is a behavior (i.e. method) of a class which simply gets a specific value and returns it. For example:

```
public class Person {
    public String name;
    int weight;

    public int getWeight(){
        return weight;
    }
}
```

Now, somewhere else in your program (probably in a different file), you could have:

```
Person p = new Person();
p.name = “Dan”;

p.introduceSelf();
System.out.println(p.name + “'s weight is” +
   p.getWeight());
```

A *setter* is a behavior (i.e. method) of a class which takes as input a new value and sets an instance variable to it. For example:

```
public class Person {
    public String name;
    int weight;

    public void setWeight(int newValue){
        weight = newValue;
    }
}
```

Now, somewhere else in your program (probably in a different file), you could have:

```
Person p = new Person();
p.name = “Dan”;

p.introduceSelf();
System.out.println(p.name + “'s weight is” +
  p.getWeight());
p.setWeight(5000); /*now I need to diet :( */
```

So what is the point of making it private if we then have these getters and setters anyway?

Mainly organization and *encapsulation*

1) If later on we decide to be more restrictive, we can do so more easily:

```
public int getWeight() {
    if (weight > 100) {
         return 100;
     }
     return weight;
}
```

Mainly organization and *encapsulation*

1) Everything in our class is nicely contained now.

Suppose I have a program with 25 .java files in it. If I make my variable ***public*** and its value isn't what I would expect, I have to go through 25 files and figure out where I set it incorrectly.

Now though, I know for sure where it is set. It would be easier to add a print statement at that point.

Example: Suppose I knew that weight was getting set to a value of 200 when I know it shouldn't be

```
public void setWeight(int newWeight) {
    if (newWeight > 200) {
          /*print some values here*/
     }
}
```

- Constructors:
-
- In Java, there is one special type of method called a constructor.
-
- A constructor is called exactly one time for every object you create with the *new* keyword.
-
- There is a default constructor for each object you create. You can also write your own.

- public class Person {
- String name = “Dan”;
- int weight;
- public Person() {
- weight = ??
- }
- }
-
- Interesting things to notice:
- 1)Name is the same as the class
- 2) No return type!

- Constructors:
-
- The main use of a constructor is to initialize the attributes that are part of the object.
-
- In the previous case, want to make sure that every Person has a weight, so we initialize it.

- Constructors with arguments:
-
- You can also create constructors with arguments. For example:
-

```
public class Person {
    String name;
     int weight = 0;
     boolean isMale;
     public Person(String theName, boolean isBoy) {
           name = theName;
           isMale = isBoy;
    }}
```

- Constructors with arguments:
-
- Now to use this constructor, you add the arguments right after the *new* keyword.
-
- Person p = new Person(“Darcy Tucker”, false);

- The *this* keyword.
-
- In Java, when you write *this* inside of a class, it means accessing the object on which the behavior was called.
-
- This is the default behavior if you omit the word *this*

A *getter* is a behavior (i.e. method) of a class which simply gets a specific value and returns it. For example:

```
public class Person {
    public String name;
    int weight;

    public int getWeight(){
        return this.weight;
    }
}
```

- Static vs non-static
-
- A behavior/method or attribute is *non-static* if it requires an *object* to operate on. In other words, you use it by first creating an object, and then using the dot operator *on an object*
-
- i.e. Person p = new Person("Foo", true);
- p.introduceSelf();
-
-

- Static vs non-static
-
- A behavior/method or attribute is *static* if it requires a *type* to operate on. In other words, you use it by writing the name of a *type* and then the name of the attribute/behavior. Because of this nature, there is exactly *one* of these per type.
-
- i.e. Math.sqrt(4);
-
- We don't first create a Math object. Math is the *type*

- Calling non-static things from a static context.
-
- An error we see sometimes relates to calling non-static objects from a static context.
-
- If I have a method or property that is *not* static, then I MUST create an object first and then call it on that object.
-
-

- Calling static things from a non-static context:
-
- Similarly, if I have a *static* attribute or behavior, then I must call it using the name of the type.
-
-

- In either case, if I make the call in the same instance of the same class, then I can omit this.
-
- i.e. inside the class Person, I can just write “name” not “this.name”
-
-
-

A *setter* is a behavior (i.e. method) of a class which takes as input a new value and sets an instance variable to it. For example:

```
public class Person {
    public String name;
    int weight;

    public void setWeight(int newValue){
        weight = newValue;
    }
}
```

Now, somewhere else in your program (probably in a different file), you could have:

```
Person p = new Person();
p.name = “Dan”;

p.introduceSelf();
System.out.println(p.name + “'s weight is” +
   p.weight); ---->ERROR: Can't access private
                   field
p.weight =5000; /*now I need to diet :( */
```

It is good style to make all member variables private.

This ensures better encapulation of our code---meaning that it is more self-contained.

If we need to access the values or set the values of member variables, we can create a public method inside our class to do so:

These are referred to as getters and setters.

A *getter* is a behavior (i.e. method) of a class which simply gets a specific value and returns it. For example:

```
public class Person {
    public String name;
    int weight;

    public int getWeight(){
        return this.weight;
    }
}
```

A *setter* is a behavior (i.e. method) of a class which takes as input a new value and sets an instance variable to it. For example:

```
public class Person {
   public String name;
   int weight;

   public void setWeight(int newValue){
      weight = newValue;
   }
}
```

Now, somewhere else in your program (probably in a different file), you could have:

```
Person p = new Person();
p.name = “Dan”;

p.introduceSelf();
System.out.println(p.name + “'s weight is” +
  p.getWeight());
p.setWeight(5000); /*now I need to diet :( */
```

Suppose I have a reference variable x of type t.

If the class t is defined to have private attribute a and private behavior b, normally I can not write

x.a

or

x.b()

I will get an error because a and b are private

One exception occurs if you are inside the class but inside a different instance of the object. For example:

```
public class Person {
    private String name;
    int weight;

    //the following method prints an introduction to another person
    public void introduceTo(Person p) {
        System.out.println("Hi " + p.name + "! I am " + this.name+ "!");
    }
}
```

Notice that name is a private field but we can still access p.name on it. This is only because p is a Person and we are in Person class