For the primitive types, actual values are copied and passed. For any other 'types', since Java doesn't really know how big they might be, instead of passing/copying whatever constitutes value, Java passes/copies the address/reference to the object, hence the object wherever it happens to sit in memory. Indeed...

```
Car ferrari = new Car(red);
```

does not make `ferrari` the actual car, it makes instead `ferrari` the address of the actual car.

```
Car bus = new Car(green);
```
This leads to various consequences...

Hat cheap = new Hat (white);
Hat riding = new Hat (black);

Then
Hat tasteless = new Hat (white);
gives us 2 white hats, but
cheap != tasteless
because the addresses are different.

However,
cheap = riding;
makes the changes in red which mean
cheap == riding
both being the same address, the address
of the riding hat — the address of the cheap hat
has been lost in the process!
This raises the obvious question of what address a reference holds if it's not currently referencing any particular object. The answer is:

```
null
```

If this situation is not picked up by the compiler, but only noticed at run-time, then we get the common error message:

```
NullPointerException
```

if a reference was declared but not assigned the address of an actual existing object.

It's also worth noting that once an object is no longer being referenced, as in cheap after the assignment

```
cheap = riding;
```

Java allows the memory allocated to this object cheap to be written over (whenever this may happen)—this is called automatic garbage collection.

A reference cannot reference a primitive variable.

As a final comment in this vein, suppose

```
yummy (rhubarb);
```

is a call to a method yummy with declaration

```
public int yummy ( custard ) {
```

then if custard is a primitive type the variable custard copies the value of rhubarb, but if custard is a reference then custard points to the same object rhubarb does.
We should observe that in

```
Car ferrari = new Car("red");
```

the word `new` creates an anonymous object according to the manufacturer class `Car` which has been told explicitly to make it red; the phrase `Car ferrari` makes `ferrari` an allowable reference to a `Car`-like object; finally, the `=` assigns the address of the anonymous car to the name `ferrari`!!!

**Strings**

We've seen already that there is a non-primitive class `String` for which the operator `+` is defined by 'concatenation'. Some other things about the `String` class are worth listing...

```
String vide = "";
```

creates `vide` as an empty string. Beware that '

`a` is a single character whereas "a" is a `String` with length one, and `char` is primitive but `String` is reference, so they cannot be the same.

Beware also that `==` and `!=` apply to Strings as they do to all other references — they refer to addresses, not values. To test values, use either

- binary `trois.equals(two) --> true/false`
- lexicographic `trois.compareTo(two) --> neg/0/pos`
Other String tricks are...

```java
myriad.length();
```

which, if the String myriad is "Did I blink" returns the integer value 11, and ...

```java
myriad.charAt(7);
```

which returns the character value 'b' since the count starts with D being position 0 — NOT 1!!

Also...

```java
myriad.substring(3, 8);
```

returns the string " I ble ". Any primitive type can be converted to a String by ...

```java
toString(97.46);
```

which returns the String (really the reference to the String!) "97.46", and if PI were to be defined in Math with the natural meaning, then

```java
toString(Math.PI);
```

would return the address of a horribly long String!

Finally, the following have equivalent effect:

```java
int n = Integer.parseInt("96");
int n = Integer.valueOf("96").intValue();
int n = new Integer("96").intValue();
```

However, for primitives other than integer (i.e., Short, int, long) only the analogues of the last two forms work.
Arrays

Arrays in Java are references, so

```java
int [] boxes; or int boxes[];
```

declares boxes to be the reference for an integer array which doesn't yet exist (so the value of boxes is null). To create we of course use new:

```java
boxes = new int [2010];
```

This array might be initialized by ...

```java
for (int i=0; i < boxes.length; i++)
    boxes [i] = i * i % 12;
```

To avoid the dreaded `NullPointerException`, if our array was instead

```java
Rhubarb [] holes = new Rhubarb [2010];
```

then the initialization would be ...

```java
for (int i=0; i < holes.length; i++)
    holes [i] = new Rhubarb (i);
```

assuming that the class `Rhubarb` knows what to do with an integer `i`!

Again, since arrays are references, be careful with `=`, `==`, and `!=`. Of course, arrays are passed by reference into methods just like any other reference — another standard source of errors.
Two (and higher) dimensional arrays are handled similarly...

```plaintext
Rhubarb [1][1] croquet;
```

`Rhubarb [1][1] croquet;` declares `croquet` to be the reference of a two dimensional array of `Rhubarb`s, and then
```
croquet = new Rhubarb[12][24];
```
creates an array prepared to hold two gross of `Rhubarb`s, and then it can be filled with any actually created (using `new`) actual `Rhubarb`s.

It's worth noting that the above two dimensional array `croquet` is really treated as if it were a one dimensional array holding 12 objects, where each of these objects is a one dimensional array holding 24 actual addresses of `Rhubarb`s. So...

```
croquet
```

[Diagram: Illustration of the `croquet` array structure, showing how it is treated as a one-dimensional array with addresses of `Rhubarb`s.]
This means we can write...

```java
croquet = new Rhundal[10][10];
```

which makes `croquet` the reference of a two
dimensional array of `Rhundals`, which can be
thought of as having 10 rows of as yet
undetermined length. Of course, the length
of each row will have to be fixed (using
`new`) before the rows can be filled...

```java
for (i=0; i < croquet.length; i++)
    [ croquet[i] = new Rhundal[i+i+1];
    for (j=0; j < croquet[i].length; j++)
        croquet[i][j] = new Rhundal(i+j);
```

assuming this makes
sense for `Rhundals`!!

Using this and similar tricks, we can size
our arrays **dynamically**, though always being careful
to remember that we're working with references. The
concrete size of the array must be fixed before
it's used, so really dynamic resizing is an exercise in
continual re-copying.
You must have wondered what the array args is used for in...

```java
public static void main(String[] args)
```

The answer is that it's designed to allow the use of command line arguments. So the program...

```java
public class Average {
    public static void main(String[] args)
    {
        double avg = 0.0;
        if (args.length != 0)
            for (int i = 0; i < args.length; i++)
                avg += Integer.parseInt(args[i]);
        avg = avg / args.length;
        System.out.println("The average is "+ avg);
    }
    else
        System.out.println("You didn't give me any numbers.");
}
```

This could be invoked by typing...

```
Average 3 17 4 9684 18
```