Lecture 8

Box and pointer notation

Many slides were borrowed from Josh Hug

Reminders

- Mic
- PA02 feedback should be on canvas.

Expected output?

```
public class Example {
    public static void test() {
        int a;
        int b;
        System.out.print(a + b);
    }
    public static void main(String[] args) {
        test();
    }
}
```

A: 0

B: 01

C: 1

D: Error

Expected output?

```
public class Example {
    int a;
    int b;
    public static void test() {
        System.out.print(a + b);
    }
    public static void main(String[] args) {
        test();
    }
```

A: 0

B: 01

C: 1

D: Error

Expected output?

```
public class Example {
    static int a;
    static int b;
    public static void test() {
        System.out.print(a + b);
    }
    public static void main(String[] args) {
        test();
    }
```

A: 0

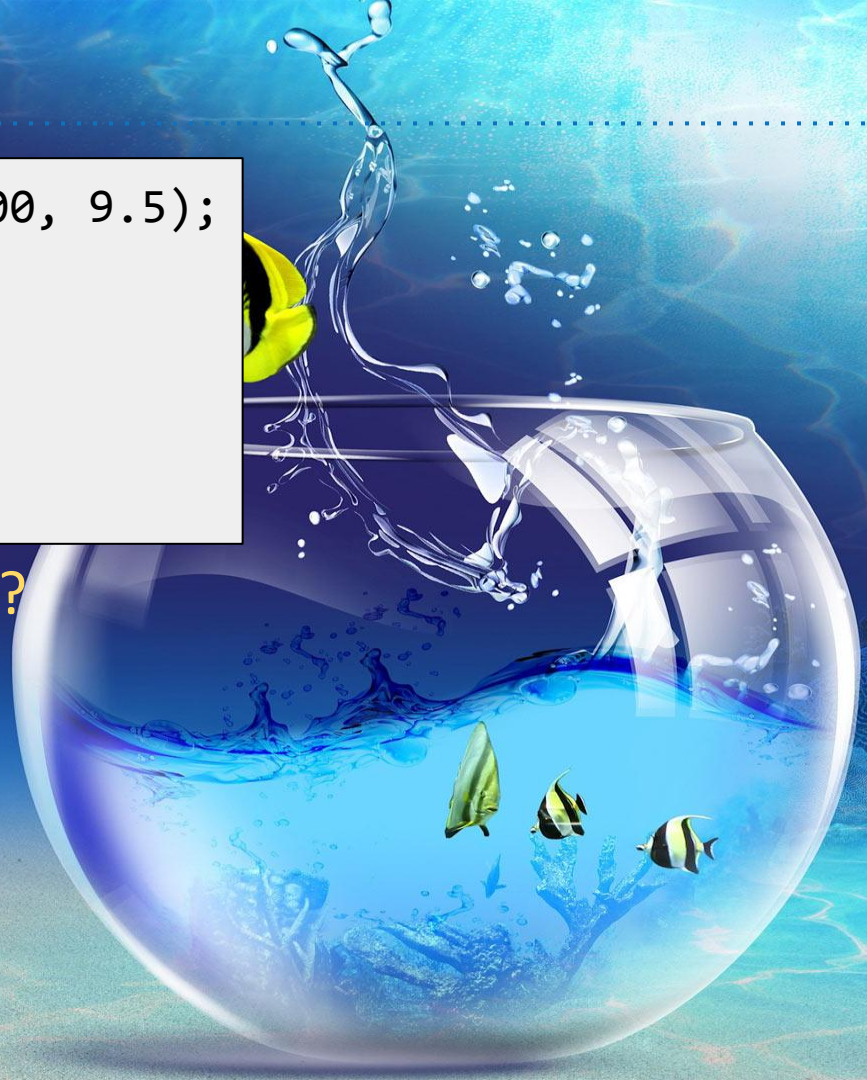
B: 01

C: 1

D: Error

Discussion Question

```
Aquarium a = new Aquarium(100, 9.5);
Aquarium b;
b = a;
b.price = 50;
System.out.println(a);
System.out.println(b);
```



Will the change to b affect a?

A. Yes
B. No

Answer: Visualizer

Discussion Question

```
Aquarium a = new Aquarium(100, 9.5);
Aquarium b;
b = a;
b.price = 50;
System.out.println(a);
System.out.println(b);
```

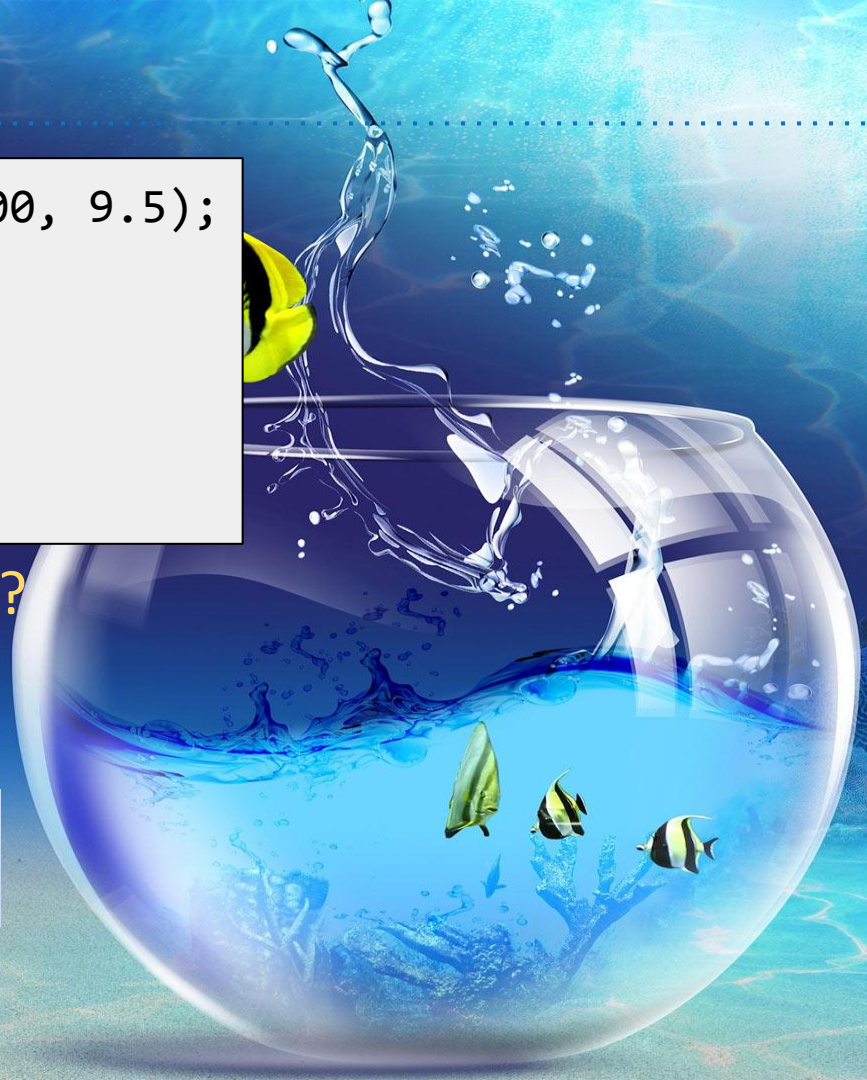
Will the change to b affect a?

A. Yes

B. No

price: 50, tank volume: 9.50
price: 50, tank volume: 9.50

Answer: Visualizer



Discussion Question

```
int x = 5;
int y;
y = x;
x = 2;
System.out.println("x is: " + x);
System.out.println("y is: " + y);
```

Will the change to x affect y?

A. Yes

B. No

Answer: Visualizer

Discussion Question

```
int x = 5;
int y;
y = x;
x = 2;
System.out.println("x is: " + x);
System.out.println("y is: " + y);
```

Will the change to x affect y?

A. Yes

B. No

x is: 2
y is: 5

Answer: Visualizer

Bits

Your computer stores information in “memory”.

- Information is stored in memory as a sequence of ones and zeros.
  - Example: 72 stored as 01001000
  - Example: 205.75 stored as … 01000011 01001101 11000000 00000000
  - Example: The letter H stored as 01001000 (same as the number 72)
  - Example: True stored as 00000001

Each Java type has a different way to interpret the bits:

- 8 primitive types in Java: byte, short, **int**, long, float, **double**, boolean, char
- We won’t discuss the precise representations in much detail in this class.

Note: Precise representations may vary from machine to machine.

Declaring a Variable (Simplified)

When you declare a variable of a certain type in Java:

- Your computer sets aside exactly enough bits to hold a thing of that type.
  - Example: Declaring an int sets aside a “box” of 32 bits.
  - Example: Declaring a double sets aside a box of 64 bits.
- Java creates an internal table that maps each variable name to a location.
- Java does NOT write anything into the reserved boxes.
  - For safety, Java will not let access a variable that is uninitialized.

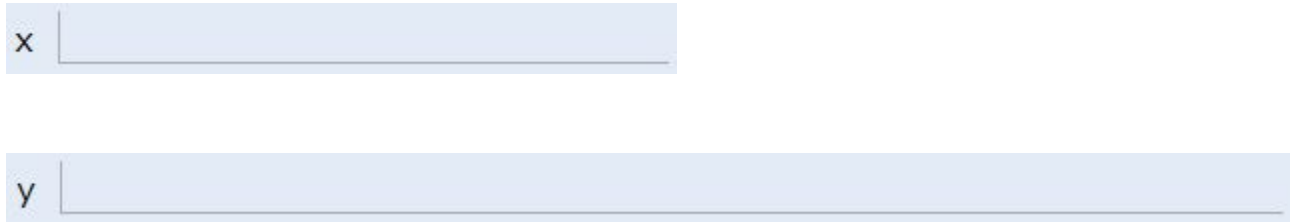
```
int x;
double y;
x = -1431195969;
y = 567213.112;
```

Declaring a Variable (Simplified)

When you declare a variable of a certain type in Java:

- Your computer sets aside exactly enough bits to hold a thing of that type.
  - Example: Declaring an int sets aside a “box” of 32 bits.
  - Example: Declaring a double sets aside a box of 64 bits.
- Java creates an internal table that maps each variable name to a location.
- Java does NOT write anything into the reserved boxes.
  - For safety, Java will not let access a variable that is uninitialized.

```
int x;
double y;
x = -1431195969;
y = 567213.112;
```

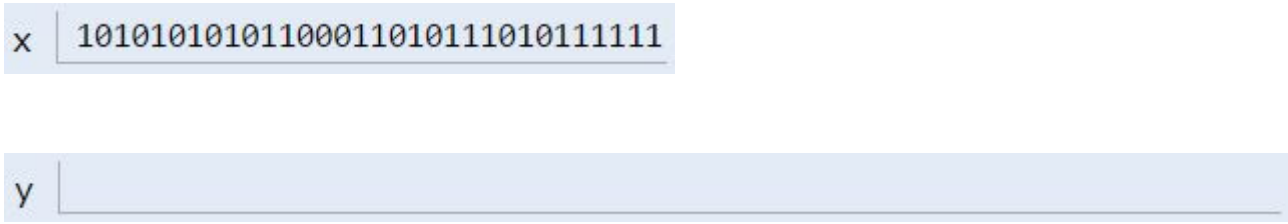


Declaring a Variable (Simplified)

When you declare a variable of a certain type in Java:

- Your computer sets aside exactly enough bits to hold a thing of that type.
  - Example: Declaring an int sets aside a “box” of 32 bits.
  - Example: Declaring a double sets aside a box of 64 bits.
- Java creates an internal table that maps each variable name to a location.
- Java does NOT write anything into the reserved boxes.
  - For safety, Java will not let access a variable that is uninitialized.

```
int x;
double y;
x = -1431195969;
y = 567213.112;
```



Declaring a Variable (Simplified)

When you declare a variable of a certain type in Java:

- Your computer sets aside exactly enough bits to hold a thing of that type.
  - Example: Declaring an int sets aside a “box” of 32 bits.
  - Example: Declaring a double sets aside a box of 64 bits.
- Java creates an internal table that maps each variable name to a location.
- Java does NOT write anything into the reserved boxes.
  - For safety, Java will not let access a variable that is uninitialized.

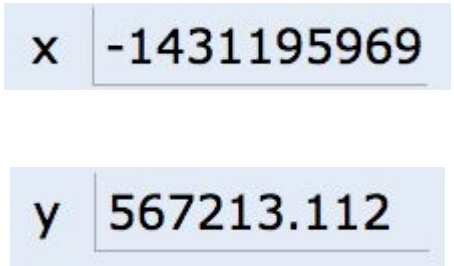
```
int x;
double y;
x = -1431195969;
y = 567213.112;
```

Simplified Box Notation

We'll use simplified box notation from here on out:

- Instead of writing memory box contents in binary, we'll write them in human readable symbols.

```
int x;
double y;
x = -1431195969;
y = 567213.112;
```



The Golden Rule of Equals (GRoE)

Given variables y and x:

- y = x **copies** all the bits from x into y.

Example from earlier: Link

References

Reference Types

There are 8 primitive types in Java:

- byte, short, **int**, long, float, **double**, boolean, char

Everything else, including arrays, is a **reference type**.

Class Instantiations

When we instantiate an Object (e.g. Car, Aquarium, Queue):

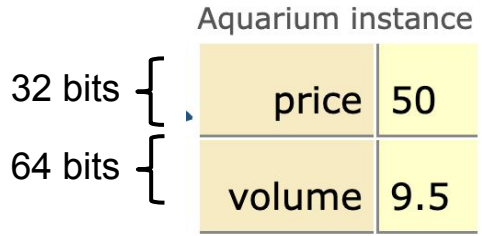
- Java first allocates a box of bits for each instance variable of the class and fills them with a default value (e.g. 0, null).
- The constructor then usually fills every such box with some other value.

```
public class Aquarium {
    public int price;
    public double volume;

    public Aquarium(int p, double v) {
        price = p;
        volume = v;
    }
}
```

```
new Aquarium(50, 9.5);
```

Demo Link



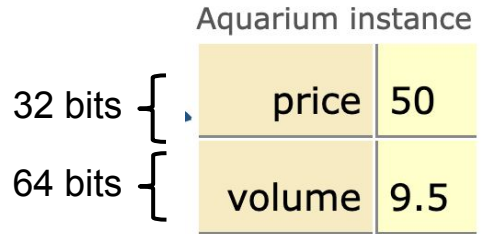
Class Instantiations

When we instantiate an Object (e.g. Dog, Aquarium, Queue):

- Java first allocates a box of bits for each instance variable of the class and fills them with a default value (e.g. 0, null).
- The constructor then usually fills every such box with some other value.

```
....00000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000001111101
00001000000001000001001100110011001100110
01100110011001100110100000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000....
```

```
new Aquarium(50, 9.5);
```



Green is price, blue is volume.

(In reality, total Aquarium size is slightly larger than 96 bits.)

Class Instantiations

Can think of new as returning the address of the newly created object.

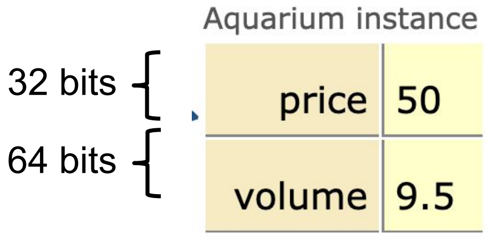
- Addresses in Java are 64 bits.
- Example (rough picture): If object is created in memory location 2384723423, then new returns 2384723423.

2384723423th bit

```
....00000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000001111101
000010000000010000010011001100110011001100110
011001100110011001101000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000000000000000000000000
000000000000000000000000....
```

2384723423

```java
new Aquarium(50, 9.5);
```

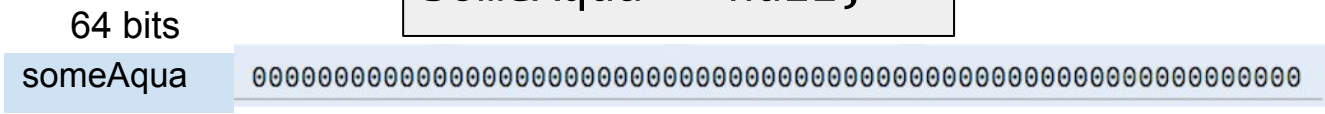


Reference Type Variable Declarations

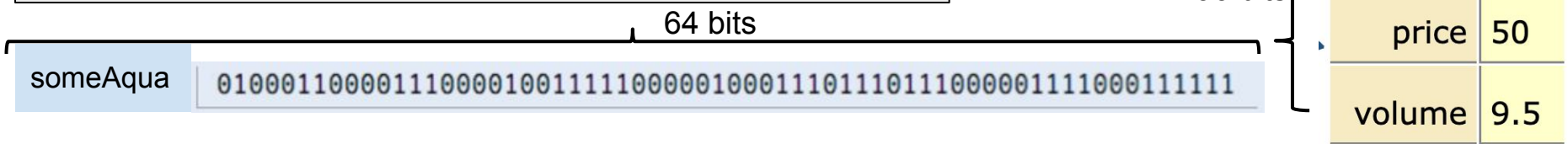
When we declare a variable of any reference type (Aquarium, Dog, Queue):

- Java allocates exactly a box of size 64 bits, no matter what type of object.
- These bits can be either set to:
  - Null (all zeros).
  - The 64 bit "address" of a specific instance of that class (returned by **new**).

```
Aquarium someAqua;
someAqua = null;
```




```
Aquarium someAqua;
someAqua = new Aquarium(50, 9.5);
```

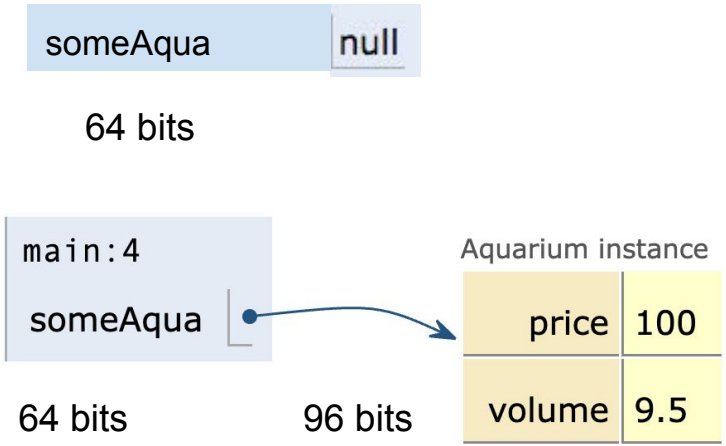


Reference Type Variable Declarations

The 64 bit addresses are meaningless to us as humans, so we'll represent:

- All zero addresses with "null".
- Non-zero addresses as arrows.

This is sometimes called "box and pointer" notation.

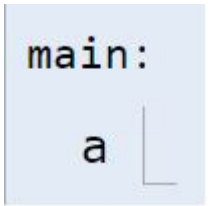


Reference Types Obey the Golden Rule of Equals

Just as with primitive types, the equals sign copies the bits.

- In terms of our visual metaphor, we “copy” the arrow by making the arrow in the b box point at the same instance as a.

```
Aquarium a;
a = new Aquarium(100, 9.5);
Aquarium b;
b = a;
```



a is 64 bits

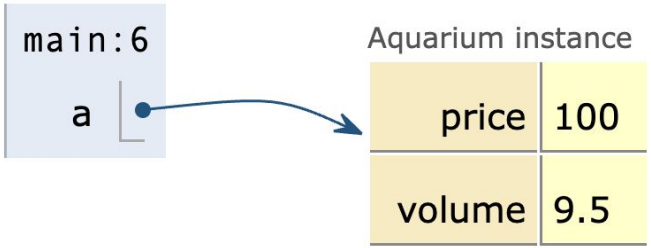
Reference Types Obey the Golden Rule of Equals

Just as with primitive types, the equals sign copies the bits.

- In terms of our visual metaphor, we “copy” the arrow by making the arrow in the b box point at the same instance as a.

```
Aquarium a;
a = new Aquarium(100, 9.5);
Aquarium b;
b = a;
```

The Aquarium shown is 96 bits.



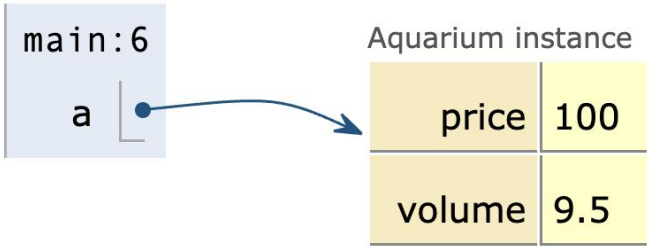
Reference Types Obey the Golden Rule of Equals

Just as with primitive types, the equals sign copies the bits.

- In terms of our visual metaphor, we “copy” the arrow by making the arrow in the b box point at the same instance as a.

```
Aquarium a;
a = new Aquarium(100, 9.5);
Aquarium b;
b = a;
```

The Aquarium shown is 96 bits.



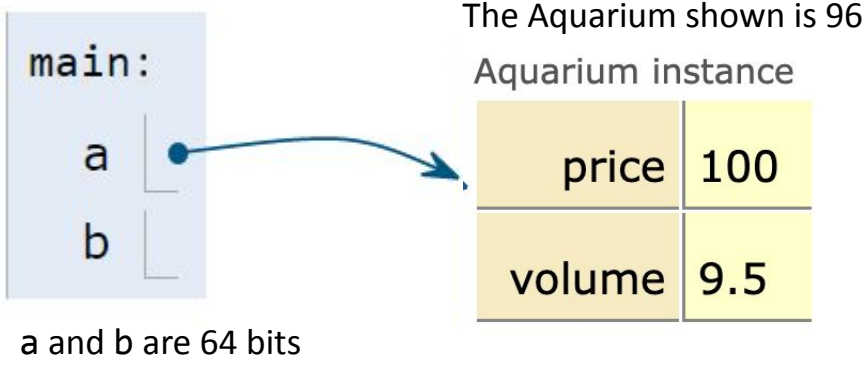
a is 64 bits

Reference Types Obey the Golden Rule of Equals

Just as with primitive types, the equals sign copies the bits.

- In terms of our visual metaphor, we "copy" the arrow by making the arrow in the b box point at the same instance as a.

```
Aquarium a;
a = new Aquarium(100, 9.5);
Aquarium b;
b = a;
```



Note: b is currently undefined, not null!

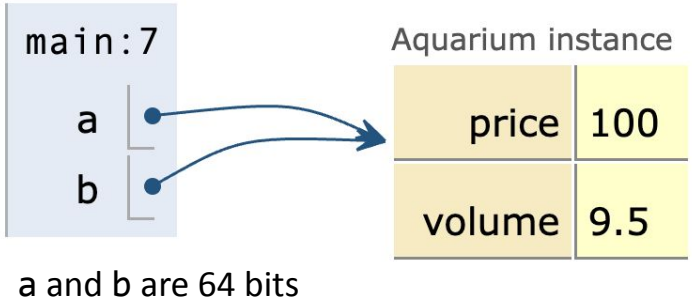
Reference Types Obey the Golden Rule of Equals

Just as with primitive types, the equals sign copies the bits.

- In terms of our visual metaphor, we “copy” the arrow by making the arrow in the b box point at the same instance as a.

```
Aquarium a;
a = new Aquarium(100, 9.5);
Aquarium b;
b = a;
```

The Aquarium shown is 96



a and b are 64 bits

Parameter Passing

The Golden Rule of Equals (and Parameter Passing)

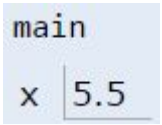
Given variables b and a:

- `b = a` **copies** all the bits from a into b.

Passing parameters obeys the same rule: Simply **copy the bits** to the new scope.

```
public static double average(double a, double b) {
    return (a + b) / 2;
}

public static void main(String[] args) {
    double x = 5.5;
    double y = 10.5;
    double avg = average(x, y);
}
```



The Golden Rule of Equals (and Parameter Passing)

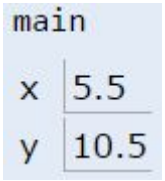
Given variables b and a:

- `b = a` **copies** all the bits from a into b.

Passing parameters obeys the same rule: Simply **copy the bits** to the new scope.

```
public static double average(double a, double b) {
    return (a + b) / 2;
}

public static void main(String[] args) {
    double x = 5.5;
    double y = 10.5;
    double avg = average(x, y);
}
```



The Golden Rule of Equals (and Parameter Passing)

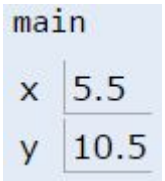
Given variables b and a:

- `b = a` **copies** all the bits from a into b.

Passing parameters obeys the same rule: Simply **copy the bits** to the new scope.

```
public static double average(double a, double b) {
    return (a + b) / 2;
}

public static void main(String[] args) {
    double x = 5.5;
    double y = 10.5;
    double avg = average(x, y);
}
```



The Golden Rule of Equals (and Parameter Passing)

Given variables b and a:

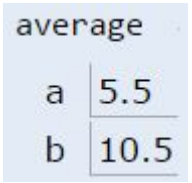
- `b = a` **copies** all the bits from a into b.

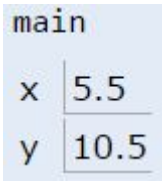
This is also called pass by value.

Passing parameters obeys the same rule: Simply **copy the bits** to the new scope.

```
public static double average(double a, double b) {
    return (a + b) / 2;
}

public static void main(String[] args) {
    double x = 5.5;
    double y = 10.5;
    double avg = average(x, y);
}
```





The Golden Rule: Summary

There are 9 types of variables in Java:

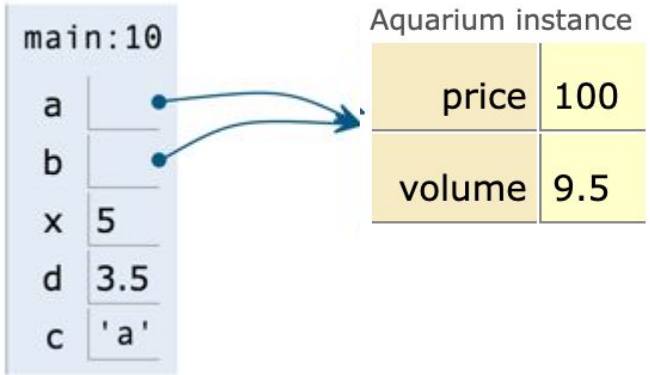
- 8 primitive types (byte, short, int, long, float, double, boolean, char).
- The 9th type is references to Objects (an arrow). References may be null.

In box-and-pointer notation, each variable is drawn as a labeled box and values are shown in the box.

- Addresses are represented by arrows to object instances.

The golden rule:

- b = a **copies the bits** from a into b.
- Passing parameters **copies the bits**.



Test Your Understanding:

Does the call to changePrice(aqua, p) have an affect on aqua and/or main's p?

A. Neither will change.
B. aqua will be cheaper by 50$, but main's p will not change.
C. aqua will not change, but main's p will decrease by 5.
D. Both will decrease.

Answer: run the code

```
public static void main(String[] args) {
    Aquarium aqua = new Aquarium(150, 10.5);
    int p = 9;
    changePrice(aqua, p);
    System.out.println(aqua);
    System.out.println(p);
}
public static void changePrice(Aquarium a, int p)
{
    a.price = a.price - 50;
    p = p - 5;
}
```

Instantiation of Arrays

Declaration and Instantiation of Arrays

Arrays are also Objects. As we've seen, objects are (usually) instantiated using the ***new*** keyword.

- `Animal p = new Animal(0, 0, 0, 0, 0, "blah.png");`
- `int[] x = new int[]{0, 1, 2, 95, 4};`

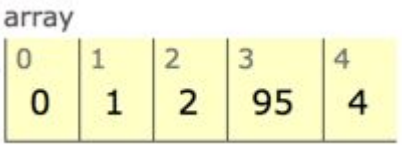
Declaration

```
int[] a;
```

- Declaration creates a 64 bit box intended only for storing a reference to an int array. **No object is instantiated.**


```
new int[]{0, 1, 2, 95, 4};
```

- Instantiates a new Object, in this case an int array.



Assignment of Arrays

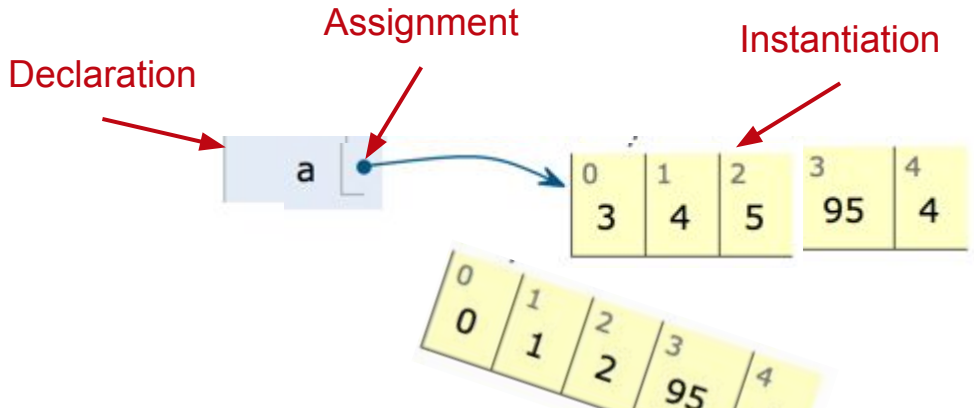
Declaration, instantiation, and assignment.

```
int[] a = new int[]{0, 1, 2, 95, 4};
```

- Creates a 64 bit box for storing an int array address. (declaration)
- Creates a new Object, in this case an int array. (instantiation)
- Puts the address of this new Object into the 64 bit box named a. (assignment)

Note: Instantiated objects can be lost!

- If we were to reassign a to something else, we'd never be able to get the original Object back!



Discussion Question

What gets printed?
(assume it is in main)

```
int[] bikes = new int[5];
System.out.println(bikes);
```

A) 0 0 0 0 0

B) null null null null null

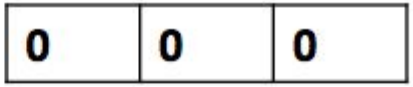
C) [I@32x34 (memory address)

D) Unknown

E) Compiler error

Primitive Arrays

```
int[] GTAPeople= new int[3];
```




```
int numCars = 5;
float[] carSpeeds;
carSpeeds = new float[numCars];
```


```
double[] playerDamage = {3.23, 4.44};
```

Memory

- Primitive type variables are not initialized to 0
- Compiler error if program uses variable before initialized
- Primitive arrays are initialized to 0
    - What are char[ ], int[ ], float[ ], double[ ] initialized to?

```
public class Test {
  public static void newMethod(int[] i, int j) {
    i[0]++;
    j++;
    return;
  }
  public static void main(String[] args) {
    int[] i = {10, 20};
    newMethod(i, i[1]);
    System.out.println(i[0] + " " + i[1]);
  }
}
```

A) 10 20
B) 11 20
C) 10 21
D) 11 21
E) None

```
int[] array1 = {1, 2, 3};
int[] array2 = {1, 2, 3};
int[] array3 = array1;

System.out.println(array1 == array2);
System.out.println(array1 == array3);
```

A) true, true
B) true, false
C) false, true
D) false, false
E) Compiler error

```
public class FinalQuestion {
  public static void changeArray(int[] i) {
    i = new int[10];
  }
  public static void main(String[] args) {
    int[] i = {10, 20};
    changeArray(i);
    System.out.println(i.length);
  }
}
```

A) 0
B) 1
C) 2
D) 10

(2 points) Fill in the contents of the boxes with the contents of the boo and woo variables at the indicated points in time.

```
class Skeleton {
   public int x, y;

   public Skeleton (int x, int y) {
      this.x = x;
         this.y = y;
   }
   public static void shaky (Skeleton a, Skeleton b) {
      Skeleton temp = b;
      b = a;
      a = temp;
   }
   public static void dizzy (Skeleton a, Skeleton b) {
      Skeleton temp = a;
      a.x = b.x;
      a.y = b.y;
      b.x = temp.x;
      b.y = temp.y;
   }

   public static void main(String [] args) {
      Skeleton boo = new Skeleton(66, 99);
      Skeleton woo = new Skeleton(33, 88);

}
```

Skeleton.shaky(boo, woo);

Your Answer			
boo.x	boo.y	woo.x	woo.y

Skeleton.dizzy(boo, woo);

Your Answer			
boo.x	boo.y	woo.x	woo.y

}