COMP 250

Lecture 7

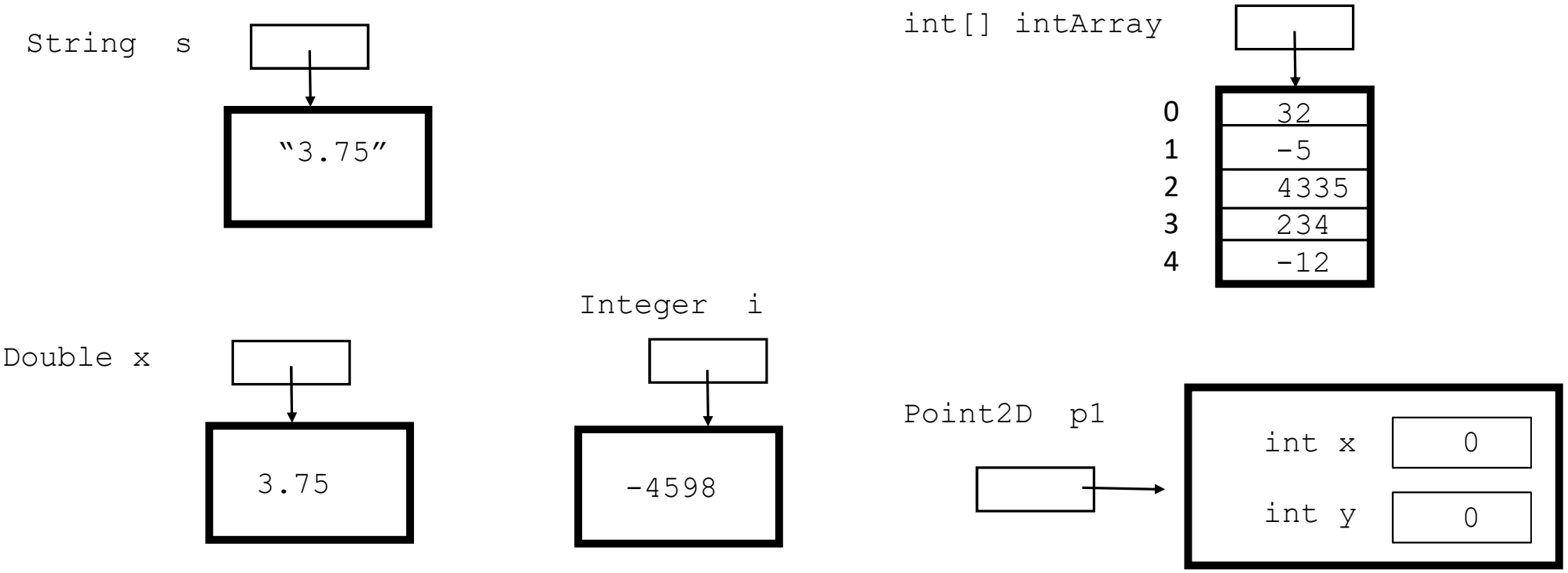
Objects & Classes 2:

`null`, aliasing,

`static`, variable scope

Jan. 21, 2022

Reference variables and objects (“instances”)



We say that these objects are *instances* of their class.

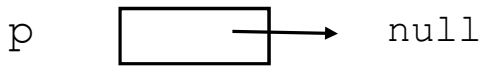
Keyword `null`

When a reference variable does *not* reference any object, we say the value of this variable is `null`. For example,

```
Point2D   p;   // value initialized to null     (typo, not initialized)

p  =  new  Point2D();

p  = null;        //   we can assign a reference to null
```

The original object (not shown) would be garbage collected.

I will draw a reference variable with value `null` sometimes as follows:



Variable initialization

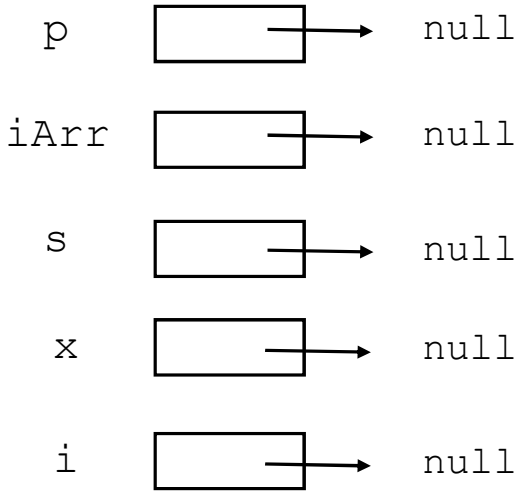
```
class  Test {
   int      i1;      //  initialized by default to 0
   String   s1;      //  initialized by default to null
   double[]  dArr1;   //  initialized by default to null

   myMethod(){
       int  i2;              //  not initialized !
       String  s2;           //  not initialized !
       double[]  dArry2;     //  not initialized !
       // bla bla
   };

   Test() {}          //  constructor does not need to initialize
                      //  the fields, since they have default values
                      //  (see above)
}
```

Keyword `null`

~~When *any* reference type variable is declared, its default value is `null`.~~

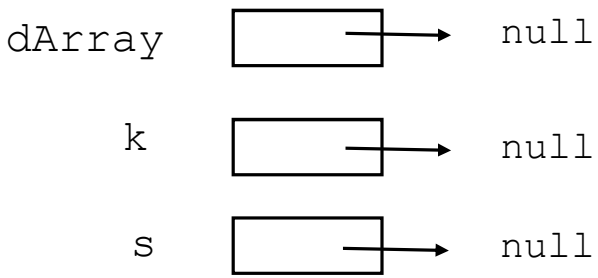
[Correction of above statement (Feb. 4) – see previous slide :] When we define reference variables *as fields in a class*, they are automatically assigned a default value `null`.

```
Point2D   p;
int[]     iArr;
String    s;
Double    x;
Integer   i;
```



Example: no argument constructor

```
class Demo {
   double[]      dArray;
    Integer      k;
    String       s;

    Demo(){ };

    //    :
}
```



The no-argument constructor initializes the reference variables to have value `null`.
If there is no constructor, then the default constructor plays the same role.

"Null pointer exception" (*term pre-dates Java*)

If we try to use a reference variable that has the value `null` in a case where the program expects an object, we will get a runtime error called a `NullPointerException`.

```
int[]  intArray = null;
String    s = null;
Double    x = null;
```

If we just declare the variable but don't initialize, then we get a compiler error.

```
intArray[0] = 3;
char      c = s.charAt(0);
double    y =  x.doubleValue();
```

Here we get the runtime error for all three.

COMP 250

Lecture 7

Objects & Classes 2:

`null`, aliasing,

`static`, variable scope

Jan. 21, 2022

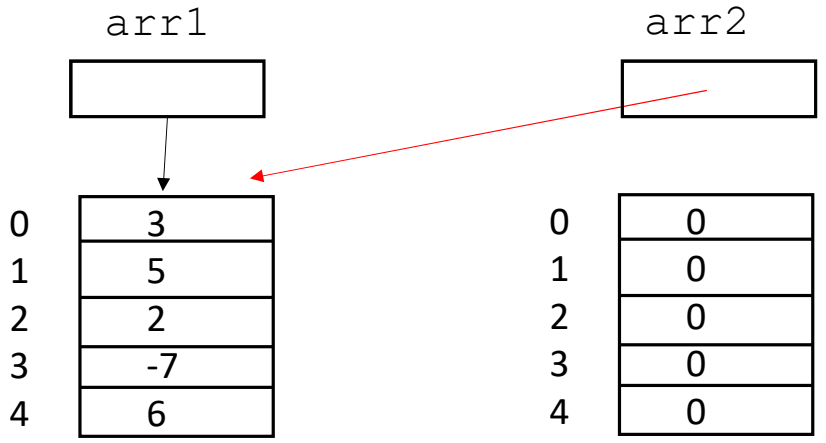
Aliasing

In general, “aliasing” means that we have different names for the same object (e.g. person).

- Prince Rogers Nelson
- Prince
- The Artist Formerly Known as Prince
-  (unpronounceable)

Aliasing: recall example from lecture 5

```
int[]  arr1 = {3, 5, 2, -7, 6};
int[]  arr2 = new int[ arr1.length ];

arr2 = arr1;
```



Now, nothing references this array. It becomes "garbage".

This was an example of aliasing.

Similar example...

```
Point2D  p1 = new  Point2D( 23, 85 );
Point2D  p2 = new  Point2D( 5,  6 ;
```

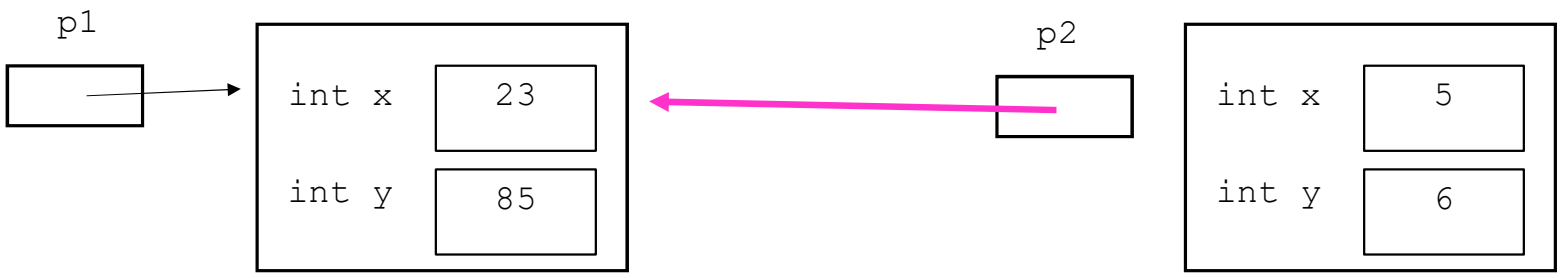

```
p2 = p1;
```

What does this instruction do ?

Similar example...

```
Point2D  p1 = new  Point2D( 23, 85 );
Point2D  p2 = new  Point2D( 5,  6 ;
p2 = p1;
```

The `Point2D` object on the left is “aliased”.



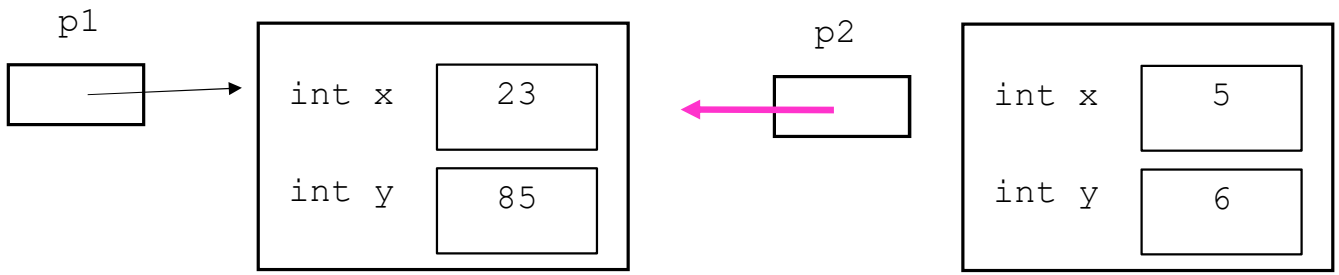
The `Point2D` object on the right will be garbage collected (eventually).

Aliasing and the == operator

For reference types, the == operator checks if its two operands are/reference the same object.

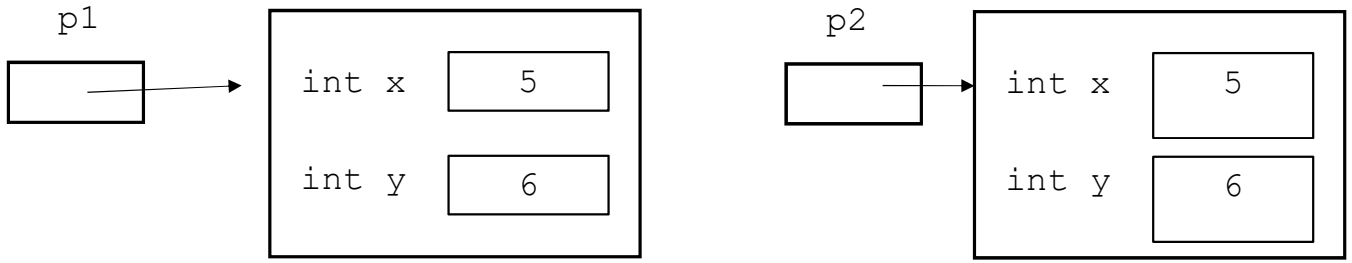
```
Point2D  p1 = new  Point2D( 23, 85 );
Point2D  p2 = new  Point2D( 5,  6  );  // p1 == p2  is false
```

```
p2 = p1;                               // p1 == p2  is true
```



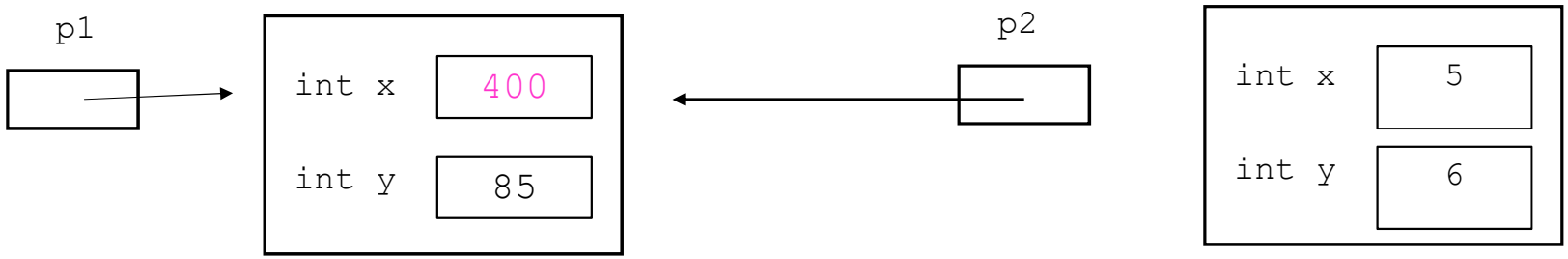
A slightly different example: The two objects have the same `x` and `y` values, but they are *different* objects:

```
Point2D  p1 = new  Point2D( 5, 6 );
Point2D  p2 = new  Point2D( 5, 6 );
```



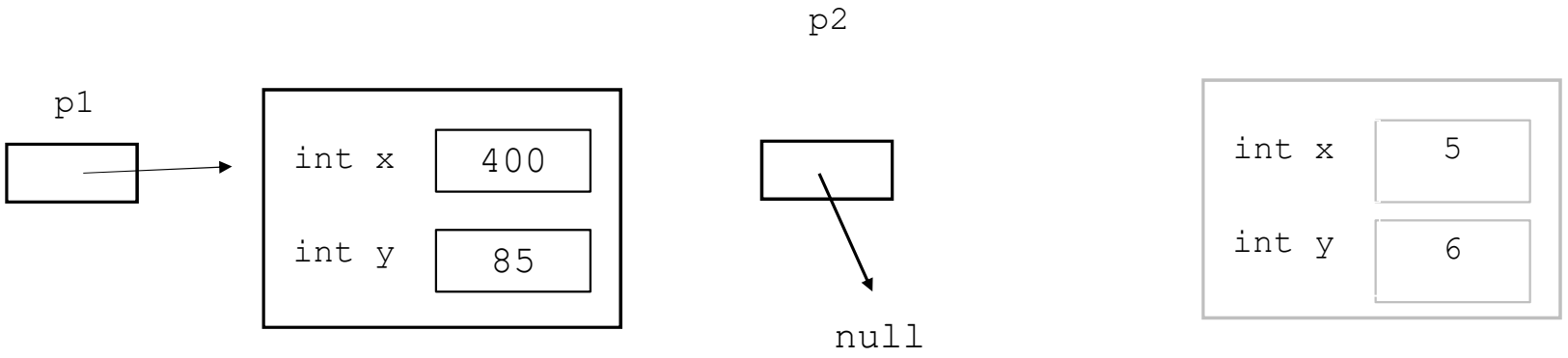
`p2 == p1` is `false`, even though the fields of the two objects have same values.

```
Point2D  p1 = new  Point2D( 23, 85 );
Point2D  p2 = new  Point2D( 5,  6  );
p2 = p1;
p2.x  = 400;
```




```
p1.x == 400   is true.
p2.x == 400   is true.
```

```
Point2D  p1 = new  Point2D( 23, 85 );
Point2D  p2 = new  Point2D( 5,  6  );
p2 = p1;
p2.x  = 400;
p2 = null;
```

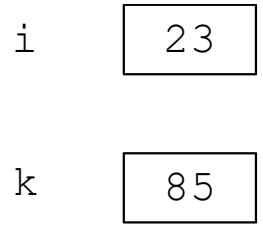


`p1.x == 400` is still true.

`p2.x` is undefined (null pointer exception).

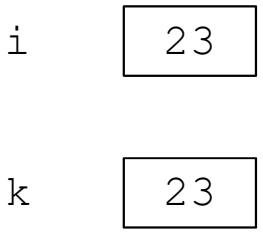
Aliasing does *not* occur with primitive types.

```
int  i = 23;
int  k = 85;
```



The following code copies the `int` value.
There are no references (arrows) here.

```
k = i;
```



COMP 250

Lecture 7

Objects & Classes 2:

`null`, aliasing,

`static`, variable scope

Jan. 21, 2022

`static` modifier

When designing classes, certain fields and methods are naturally associated with objects (instances), whereas other fields and methods are naturally associated with the class itself.

To define the latter, we use the `static` modifier.

For example, suppose some class generates many `Point2D` objects, and stores them at various positions in an array.

```
Point2D  arr = new Point2D[ 1000 ];

arr[23]  = new  Point2D( 23, 85 ) ;
arr[732] = new  Point2D( 5,  6  ) ;
arr[63]  = new  Point2D( 76, 15 ) ;
```

Suppose we want to keep track of how many objects there are.

We define a “class field” (or “class variable” or “static variable”) that counts the number of objects of that class. Such a field/variable is declared with the **`static`** modifier. It is associated with the class, rather than with any particular object.

```
class Point2D {
    int     x;                    //   instance variable (or field)
    int     y;                    //

    static  int  numberOfPoint2D;    //  static variable (class field)

    Point2D(int x, int y){
        this.x = x;
        this.y = y;
        numberOfPoint2D += 1 ;  // increment
    }
}
```

Any field variable (static or not) is initialized to a default value (0, for `int`), unless the constructor initializes it.

We can also define static methods (or “class methods”).

One common **`static`** method is `main()`.

```
class Point2D {
    int     x;
    int     y;

    //   …

    public static void main(String[] args) {

    //  we will discuss what ‘public’ means next lecture
    }
}
```

main() takes as input an optional sequence of `String` arguments. These arguments are stored in an array. You would use this commonly if you were running Java programs from the command line.

Here are two static methods that we might define for the `Point2D` class.

```
static int getNumberOfPoint2D(){
        return( numberOfPoint2D );
}
```

```
static double distanceBetween(Point2D p1, Point2D p2){

        return( Math.sqrt(
                        (p1.x - p2.x) * (p1.x - p2.x)
                      + (p1.y - p2.y) * (p1.y - p2.y) ) );
}
```

These methods is *not* associated with any particular `Point2D` object.

`Math.sqrt()` is a static method. We write the class name `Math` to specify that the `sqrt` method is from that class.

Compare the **`Point2D.distanceBetween()`** method on the last slide with the following instance (non static) method :

```
double distanceTo( Point2D p ){

    return( Math.sqrt(
                (this.x - p.x) * (this.x - p.x)
              + (this.y - p.y) * (this.y - p.y) ) );
}
```

The above method would be called by (or "invoked by") a particular `Point2D` object. It would calculate the distance from `this` point to another `Point2D` object.

Here is an example that combines the above. The two methods compute the same value.

```
class Point2D {
    int     x;              //
    int     y;

    //  put methods defined in previous slides here

    public static void main(String[] args) {

         Point2D  p1 = new  Point2D( 23, 85 );
         Point2D  p2 = new  Point2D( 5,  6 ) ;

         System.out.println(  distanceBetween(p1, p2) );
         System.out.println(  p1.distanceTo( p2 ) );
    }
}
```

In the example below, we call the methods from a different class **`Test.`**
This class'es main method must be written slightly differently: now we have to specify the class `Point2D` that the static method belongs to.

This was unnecessary on the previous slide, because we were calling a method that belonged to the `Point2D` class.

```
class Test {

    public static void main(String[] args) {

        Point2D  p1 = new  Point2D( 23, 85 );
        Point2D  p2 = new  Point2D( 5,  6 ) ;

        System.out.println(  Point2D.distanceBetween(p1, p2) );
        System.out.println(  p1.distanceTo( p2 ) );
    }
}
```

```
public class ExerciseToMiles {
    final static double KM_TO_MILES = 0.6214;

    public static double toMiles(double km){
        return km * KM_TO_MILES;
    }

    public static void main(String[] args) {
        System.out.println(  toMiles(80.0) );
    }

}
```

```
public class Test {

    public static void main(String[] args) {
        System.out.println(  ExerciseToMiles.toMiles(80.0) );
    }

}
```

COMP 250

Lecture 7

Objects & Classes 2:

`null`, aliasing,

`static`, variable scope

Jan. 21, 2022

Scope of a Variable (in Java)

Informal definition: the “scope” of a variable in a class is the part of the code where the name of that variable is well defined.

Different kinds of variables have different scopes :

- instance fields/variables (non-static)
- class fields/variables (static)
- local variables inside a method
- loop variable
- method parameters

An instance variable (field) is visible from any non-static method within the class. The field belongs to instances (objects) of the class.

```
class Demo {
      int k;        //  instance variable (field)

      void  myMethod ( ){

            k = 3;          //  we can instead write  this.k = 3;
      }

      static  void  myStaticMethod(){
            k = 3;                        //  compiler error
      }
}
```

A static variable (field) is visible from any method within the class.

```
class Demo {
      static int k;        //  instance variable (field)

      void  myMethod ( ){

            k = 3;
      }

      static  void  myStaticMethod(){
            k = 3;
      }
}
```

[UPDATED JAN. 28]

We *can* write the instruction in `myMethod` as **`this.k = 3`** although we get a warning. However, if we write the instruction in `myStaticMethod` that way, we get a compiler error. Try it yourself!

We *can* write both of these instructions as **`Demo.k = 3;`**

A local variable is defined *within* a method body.
It's scope is determined by curly brackets, and only below the variable definition.

```
class Demo {
      int k;        //  instance variable (field)

      void  myMethod ( ){

         int m1;  //  m1 scope starts

         {
             int m2;  //  m2 scope starts

         }            //  m2 scope ends

     }             //  m1 scope ends
}
```

A loop variable is defined only with the loop itself.

```
class Demo {

      void  myMethod ( ){

            for (int i = 0; i < 5; i++){
                System.out.println(i);
            }                                   //  scope i ends

            i = 15;           //  compiler error
      }
}
```

A method parameter is visible anywhere within the method.

```
class Demo {
      int k;

      void  myMethod ( int j ){

            System.out.println(j);

            j = 3;

     }
}
```

One *can* (re-)assign values to the parameter variable.
So it behaves like a local variable (next slide).

Examples where the same variable name is defined more than once.

```
class Demo {
      int k;        //  instance variable (field)

      void  myMethod( ){

         int k = 5;      //  local variable

         this.k = 27;

      }
}
```

They are two completely different variables!
When the `myMethod` exits, the local variable **`k`** is no longer defined.
But the `Demo` object's field (**`this.k`**) is still defined and keeps its value (27).

Examples where the same variable name is defined more than once.

```
class Demo {


      void  myMethod( int k ){

         int k = 5;      //  compiler error

         k = 2;
      }
}
```

The reason for the compiler error is that there would be an ambiguity.
The parameter **k** cannot have the same name as a local variable.

A few more examples...

if time permits

(8-10 minutes)

otherwise finish it up next time

Scope of a Variable (in Java)

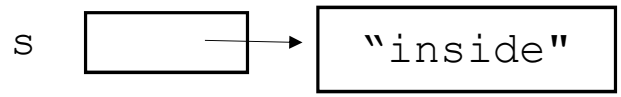
Informal definition: the “scope” of a variable in a class is the part of the code where the name of that variable is well defined.

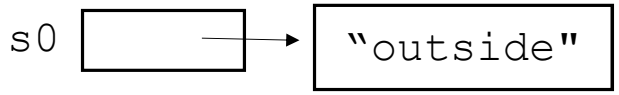
Different kinds of variables have different scopes :

- instance fields/variables (non-static)
- class fields/variables (static)
- local variables inside a method
- loop variable
- method parameters

Tricky Example:

```
class TestMain {

    static void myMethod(String s) {
       s = "inside";
    }

    public static void main(String[] args) {

       String s0 = "outside";
       myMethod(s0);

       System.out.println(s0);
   }
}
```

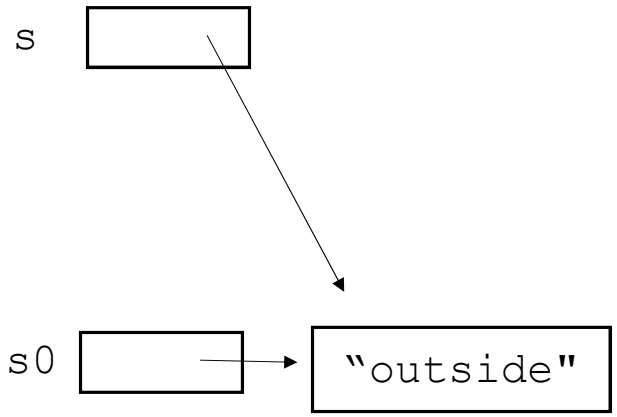




Q: What gets printed ?

Tricky Example:

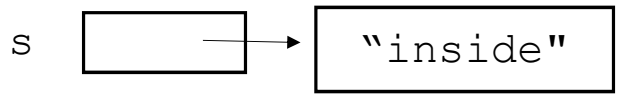
```
class TestMain {

    static void myMethod(String s) {
        s = "inside";
    };

    public static void main(String[] args) {

        String s0 = "outside";
        myMethod(s0);

        System.out.println(s0);
    }
}
```

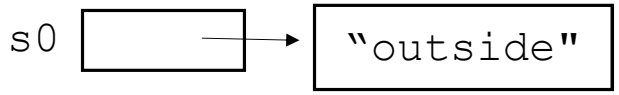


When we first enter **myMethod** and before we assign "inside" s, the method parameter variable s takes the value that is passed to it, namely the argument s0 of the caller method.

Tricky Example:

```
class TestMain {

    static void myMethod(String s) {
       s = "inside";
    };

    public static void main(String[] args) {

       String s0 = "outside";
       myMethod(s0);

       System.out.println(s0);
    }
}
```

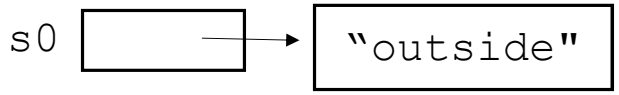




`s` is then reassigned so that it references the string `"inside"`.

Tricky Example:

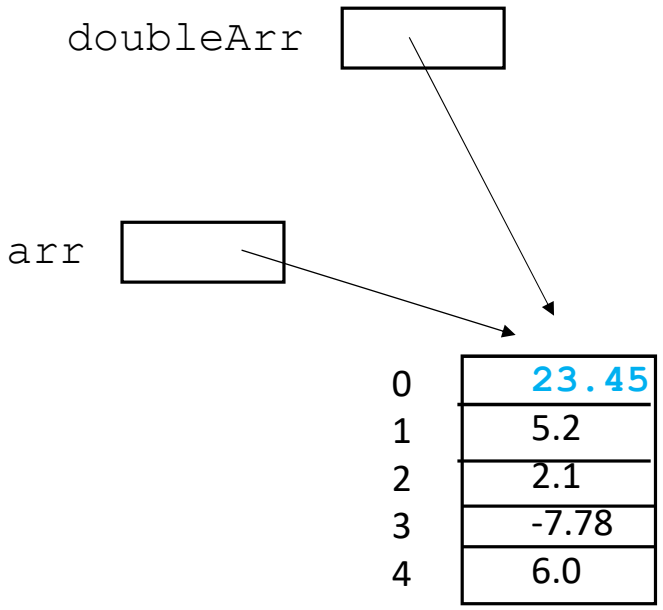
```
class TestMain {

    static void myMethod(String s) {
       s = "inside";
    };

    public static void main(String[] args) {

       String s0 = "outside";
       myMethod(s0);

       System.out.println(s0);
    }
}
```



When the method exits and returns to `main`, `s0` has its initial value. Indeed it never lost that value.) So "outside" is printed.

Recall lecture 5: Passing an array to a method

```
static void  demoPassArray ( double[]  doubleArr   ){
    doubleArr[0] = 23.45;
}
```



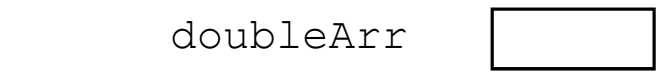
Suppose you call this method in the code below:

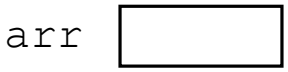
```
double[]  arr = {3.0, 5.2, 2.1, -7.78, 6.0};
demoPassArray( arr );
System.out.print( arr[0] );
```

Note the difference between this example and the previous one: here we aren't creating a new array.

Slight variation (tricky): Passing an array to a method

```
static void  demoPassArray ( double[]  doubleArr   ){
    doubleArr = new double[]{1.0,  -5.2 };
                // yes, that's the syntax needed
}
```





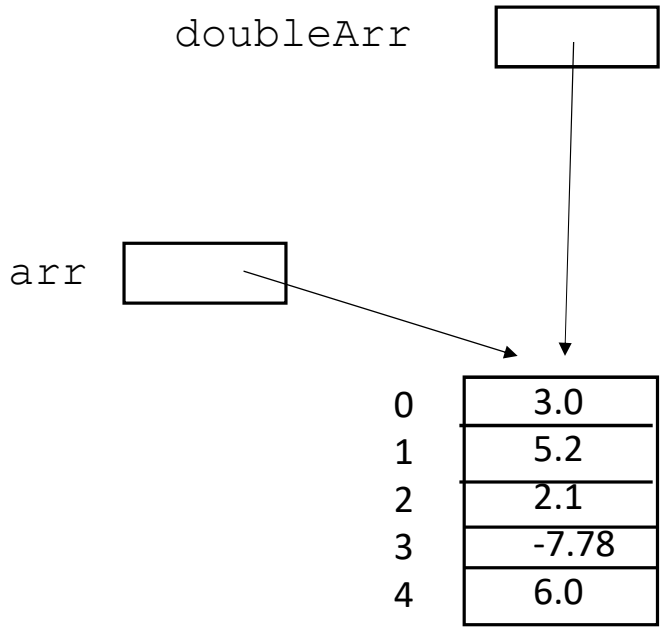
Suppose you call this method in the code below:

```
double[]  arr = {3.0, 5.2, 2.1, -7.78, 6.0};
demoPassArray( arr );
System.out.print( arr[0] );
```

What does it print ?

Slight variation (tricky): Passing an array to a method

```
static void  demoPassArray ( double[]  doubleArr   ){
    doubleArr = new double[]{1.0,  -5.2 };
                // yes, that's the syntax needed
}
```



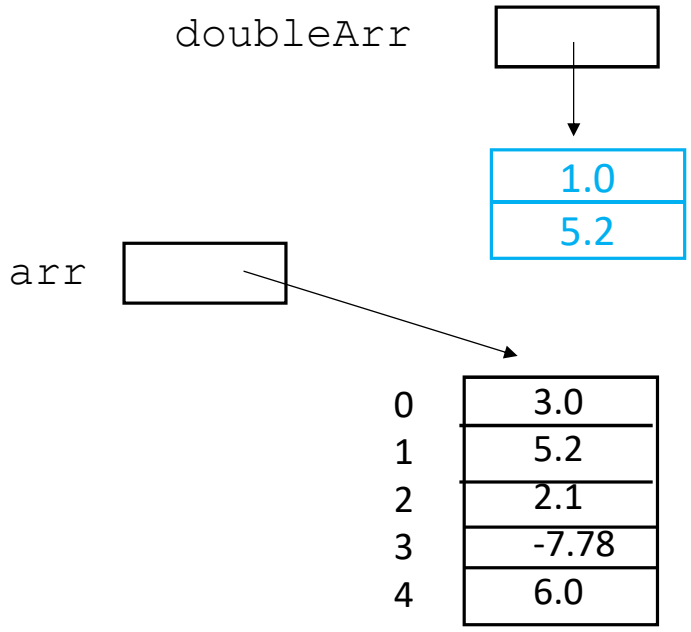
Suppose you call this method in the code below:

```
double[]  arr = {3.0, 5.2, 2.1, -7.78, 6.0};
demoPassArray( arr );
System.out.print( arr[0] );
```

When we first enter **demoPassArray** and before we construct the new array, the method parameter variable `doubleArr` references the one existing array.

Slight variation (tricky): Passing an array to a method

```
static void  demoPassArray ( double[]  doubleArr   ){
    doubleArr = new double[]{1.0,  -5.2 };
                // yes, that's the syntax needed
}
```



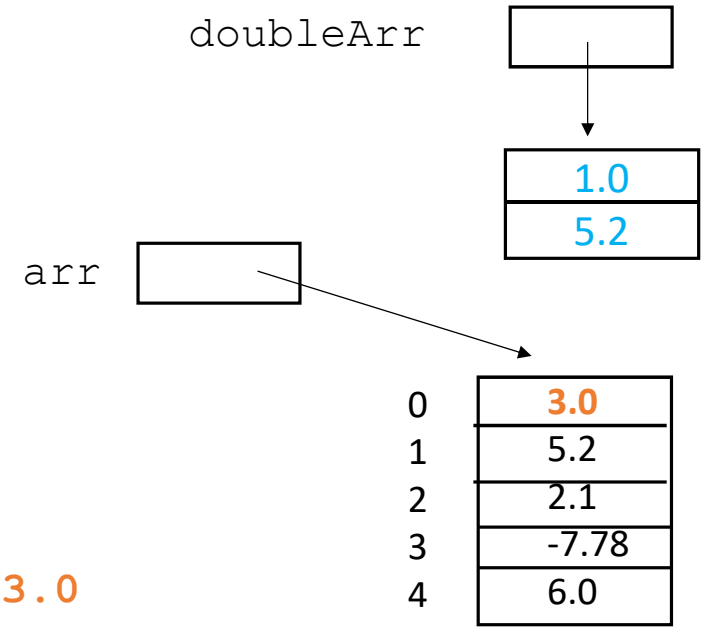
Suppose you call this method in the code below:

```
double[]  arr = {3.0, 5.2, 2.1, -7.78, 6.0};
demoPassArray( arr );
System.out.print( arr[0] );
```

The parameter `doubleArr` behaves like a local variable. This local variable will reference a new array.

Slight variation (tricky): Passing an array to a method

```
static void  demoPassArray ( double[]  doubleArr   ){
    doubleArr = new double[]{1.0,  -5.2 };
                // yes, that's the syntax needed
}
```



Suppose you call this method in the code below:

```
double[]  arr = {3.0, 5.2, 2.1, -7.78, 6.0};
demoPassArray( arr );
System.out.print( arr[0] );       //  print out  3.0
```

When the method exits and returns to `main`, `arr` hasn't changed.

Coming up…

Lectures

Mon. Jan. 24
Visibility modifiers (private & public)

Wed. Jan. 26
ArrayLists

Fri. Jan. 28
Singly Linked Lists

Assessments

Fri. Jan. 28

Quiz 1 (lectures 1-7 i.e. today)
- I will post a practice quiz by Monday

Assignment 1 to be posted
- you will have 2 weeks to do it