Classes and Objects

2020 Spring: AP Computer Science A

January 15th, 2020

Today

- **Classes and objects (General idea)**
- **Strings revisited**
- **Arrays**
- **`null`**
- **Reference Semantics**
- **Abstraction**
- **Writing a Java class**
  - Object state: Fields
  - Object behavior: Methods
  - Object initialization: Constructors
- **Encapsulation**
- **Static methods/fields**

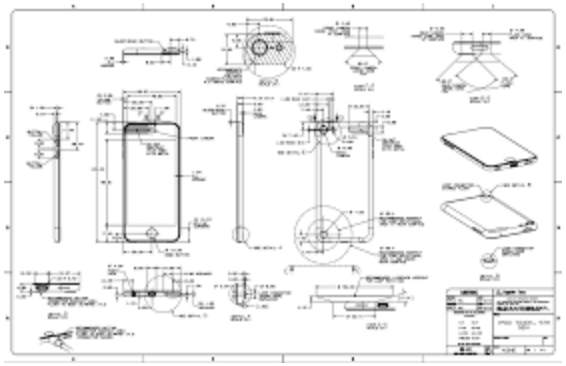
Classes and Objects

- **class: A program entity that represents either**
  - A program / module, or
  - A type of objects

  - A class is a **blueprint** or **template** for constructing objects
    - Java has thousands of built-in classes
    - We can also write our own

- **object: An entity that combines data and behavior**
  - **Object-oriented programming (OOP)**
    - Programs that perform their behavior as interactions between objects

Blueprint Analogy



iPod Blueprint / Factory

- **state**
  - current song
  - volume
  - battery life
- **behavior**
  - turn on
  - turn off
  - change song
  - change volume
  - choose random song

iPod #1

- **state**
  - Into the Unknown
  - 12
  - 35%
- **behavior**
  - turn on
  - turn off
  - change song
  - change volume
  - choose random song

iPod #2

- **state**
  - Show Yourself
  - 12
  - 35%
- **behavior**
  - turn on
  - turn off
  - change song
  - change volume
  - choose random song

Objects

- **object:** An entity that contains data and behavior
  - ***data***: variables inside the object
  - ***behavior***: methods inside the object
  - *You interact with the methods, the data is hidden in the object*

- **An object is an *instance* of a class**

- **Syntax**
  - Constructing (creating) an object
    - `Type objectName = new Type(parameters);`
  - Calling an object's method
    - `objectName.methodName(parameters);`
  - Accessing an object's data
    - `objectName.data;`

Scanner Object

- **Scanner was actually an object!**
  - Declaration with **new** keyword
  - Has methods that we can call on

Scanner Usage

- `Scanner sc = new Scanner(System.in);`

Method	Description
`sc.nextInt()`	Reads an **int** from the user
`sc.nextDouble()`	Reads a **double** from the user
`sc.next()`	Reads a *one-word* **string** from the user

- **Usage**

```
int x = sc.nextInt();        // reads int and stores it into x
double y = sc.nextDouble();  // reads double and stores it into y
String s = sc.next();        // reads string and stores it into s
```

41

Strings

- **Java strings are also objects!**
- **string: An object storing a sequence of text characters**
  - Construction
    - `String name = new String("text");`
    - `String name = "text";`
  - Methods will be shown later

- **Each character of a string are numbered with 0-based indices**
  - `String str = "Hi, Java!";`

index	0	1	2	3	4	5	6	7	8
char	H	i	,		J	a	v	a	!

    - First character's index: 0
    - Last character's index: (string's length) – 1
    - Individual characters are values of type char

String Methods

- https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/lang/String.html

Return Type	methodName(params)	Description
`int`	`indexOf(String)`	Returns the index within this string of the first occurrence of the given String
`int`	`length()`	Returns the length of this string
`String`	`substring(int, int)`	Returns a string that is a substring of this string.
`String`	`substring(int)`	Returns a string that is a substring of this string.
`String`	`toLowerCase()`	Converts all the characters in this String to upper case
`String`	`toUpperCase()`	Converts all the characters in this String to lower case
`boolean`	`equals(Object)`	Compares this string to the specified object.
`boolean`	`equalsIgnoreCase(String)`	Compares this String to another String, ignoring case considerations.
`boolean`	`startsWith(String)`	Tests if this string starts with the specified prefix.
`boolean`	`endsWith(String)`	Tests if this string ends with the specified suffix.
`boolean`	`contains(String)`	Returns true if and only if this string contains the given String.
`char`	`charAt(int)`	Returns the char value at the specified index.

- **And many more …!**

Exercises

- #11654 아스키 코드
- #10809 알파벳 찾기
- #2675 문자열 반복
- #1157 단어 공부 (★)

Programming Problem

- **#4344 평균은 넘겠지**

- **For simplicity, only consider the problem for a single test case.**

  - **Input**

```
5
50
50
70
80
100
```

  - **Output**

```
40.000%
```

- **Try to solve this problem!**

Why is it hard?

- **We need each input value twice**
  - To compute the average
  - To count how many were above average

- **We could read each value into a variable, but we**
  - don't know how many values are needed until the program runs
  - don't know how many variables to declare

- **Need a way to declare many variables in one step.**

Arrays

- **array: An object that stores many values of the same type**
  - *element*: One value in an array
  - *index*: A 0-based integer to access an element from an array

index	0	1	2	3	4	5	6	7	8
value	12	-1	49	0	5	7	-19	128	1

- **Declaration**
  - `type[] name = new type[length];`

- **Example**

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

index	0	1	2	3	4	5	6	7	8	9
value	0	0	0	0	0	0	0	0	0	0

Arrays

- **The array's length can be any integer expression.**

```
int x = 2 * 3 + 1;
int[] data = new int[x % 5 + 2];
```

- **Each element initially gets a "*zero-equivalent*" value**

Type	Default value
`int`	`0`
`double`	`0.0`
`boolean`	`false`
`String` or other objects	`null` (no object)

Arrays

- **Accessing elements**
  - Can be used like a variable

```
name[index]             // access
name[index] = value;    // modify
```

- **Example**

```
arr[0] = 27;
arr[3] = -6;

System.out.println(arr[0]);
if (arr[3] < 0) {
    System.out.println("Element 3 is negative.");
}
```

index	0	1	2	3	4	5	6	7	8	9
value	27	0	0	-6	0	0	0	0	0	0

ArrayIndexOutOfBoundsException

- ***Legal indices*: Between 0 and the array's length – 1**
  - Reading or writing any index outside this range will throw an `ArrayIndexOutOfBoundsException`

- **Example**

```
int[] arr = new int[10];
arr[-1] = -1;
System.out.println(arr[10]);
```

  - Writing to element at index -1 will cause an exception
  - Trying to print (which requires reading) the element at index 10 will cause an exception

Arrays and for Loops

- **It is common to use `for` loops to access array elements**
  - The loop counter is used as an index
  - We can also assign each element a value in a loop

```
for(int i = 0; i < 10; ++i) {
    arr[i] = 2 * i;
}
```

- **Array's `length` field stores its number of elements**
  - Access by `name.length`
  - To traverse an array, change the loop condition as follows

```
for(int i = 0; i < arr.length; ++i) {
    arr[i] = 2 * i;
}
```

for-each Loop

- **This loop is used to iterate over an array or collection**
- **Syntax**
  - Read as "***for each*** *var* ***in*** *arr*"

```
type[] arr = new type[length];
for (type var : arr) {
    statement(s);
}
```

- **Example**

```
for (int e : arr) {
    System.out.println(e);
}
```

- **Notes**
  - This loop *hides the index variable*, so this can't be used when indices are needed
  - *Cannot modify elements* as you traverse the array

Exercise

- **#4344 평균은 넘겠지**
  - Use an integer array to store all the scores
- **#10818 최소, 최대**
  - Use an integer array to store all the values
  - Traverse the array to find the maximum/minimum
- **#2562 최댓값**
- **#2577 숫자의 개수 (★)**

Quick Array Initialization

- **Suppose we want to create this array (no evident pattern)**

index	0	1	2	3	4	5
value	-2	10	-7	0	5	1

  - Normally, we would do...

```
int[] num = new int[6];
num[0] = -2;
num[1] = 10;
num[2] = -7;
num[3] = 0;
num[4] = 5;
num[5] = 1;
```

Quick Array Initialization

- **Suppose we want to create this array (no evident pattern)**

index	0	1	2	3	4	5
value	-2	10	-7	0	5	1

  - Instead, we can use initializer list
    - `type[] name = {value, value, ..., value};`

```
int[] num = {-2, 10, -7, 0, 5, 1};
```

  - Useful when you know what the array's elements will be
  - The compiler figures out the array size by counting the values.

Limitations of Arrays

- **You cannot resize an existing array**

```
int[] a = new int[4];
a.length = 10;              // error
a = new int[10];            // must reassign
```

- **You cannot compare arrays with ==**

```
int[] a1 = {1, 2, 3, 4};
int[] a2 = {1, 2, 3, 4};
if(a1 == a2) { ... }          // false
```

  - Will see why this is false later
  - Comparing arrays – on the next slide

- **An array doesn't know how to print itself**

```
int[] a1 = {1, 2, 3, 4};
System.out.println(a1);   // [I@372f7a8d
```

Arrays Class

- **Class `Arrays` in package `java.util` has useful methods for arrays**
  - https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Arrays.html

Return Type	methodName(params)	Description
`type[]`	`copyOf(type[], int)`	Copies the specified array, truncating or padding with *zero-equivalent value* (if necessary) so the copy has the specified length.
`boolean`	`equals(type[], type[])`	Returns true if the two specified arrays are equal to one another.
`void`	`fill(type[], type)`	Assigns the specified value to each element of the specified array.
`void`	`sort(type[])`	Sorts the specified array into ascending numerical order.
`String`	`toString(type[])`	Returns a string representation of the contents of the specified array.

- **Syntax**
  - `Arrays.methodName(parameters)`
- **Must import `java.util.Arrays`**

Arrays as Parameters

- **Can pass arrays as parameters, can return arrays**
  - `public static type[] methodName(type[] arr);`

- **Solve #12605 단어순서 뒤집기**
  - Use a method that returns an array with elements in reversed order

null

index	0	1	2
value	null	null	null

- **When initializing an array of objects, each element initially gets `null`**
- **`null`: A value that does not refer to any object**

- **You can**
  - Store `null` in an object variable or an object array
    - `String s = null;`
  - Print a `null` reference
    - `System.out.println(s);`
    - `System.out.println(null);`
  - Check if something is `null`
    - `if(s == null) { ... }`
  - Pass `null` as a parameter to a method
    - `someMethod(null);`
  - Return `null` from a method (often to indicate failure)
    - `return null;`

NullPointerException

- ***dereference*: To access data or methods of an object with the dot(.) notation**
  - `str.length(), arr.length, sc.nextInt()` ...
  - It is illegal to dereference `null`
    - `null` is not any object, so it has no methods or data

- **You will get a `NullPointerException` if you dereference `null`**
  - Your code should be fixed!

Reference Semantic (Objects)

- **Compare these two code segments**
  - What is the expected behavior?
  - What is the actual output?

```
int a = 1;
int b = a;
b = 3;
System.out.println(a);

int[] a1 = {4, 0, -7};
int[] a2 = a1;
a2[1] = 3;
System.out.println(Arrays.toString(a1));
```

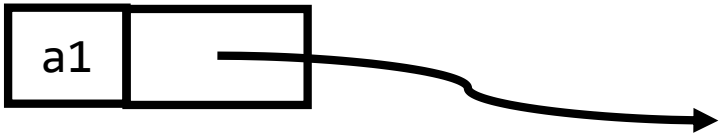
Reference Semantic (Objects)

- ***reference semantics*: Behavior where variables actually store the address of an object in memory**
  - Applies to **objects**! (*Primitive types* use *value semantics*)
  - When one variable is assigned to another, the object is **not copied**, and **both variables *refer* to the same object, reference is copied**
  - Modifying the value of one variable *will affect others*

- ***value semantics*: Behavior where values are copied when assigned, passed as parameters, or returned**
  - When one variable is assigned to another, **its value is copied**
  - Modifying the value of one variable *does not affect others*

In Detail

```
int[] a1 = {4, 0, -7};
int[] a2 = a1;
a2[1] = 3;
System.out.println(Arrays.toString(a1));
```

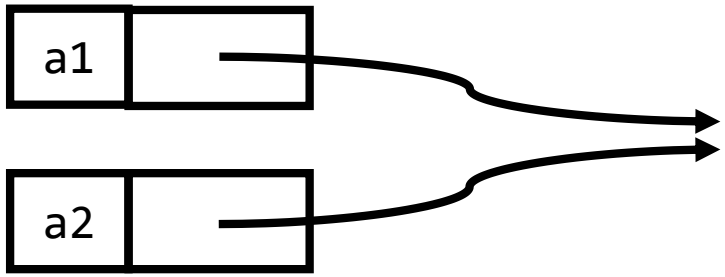


index	0	1	2
value	4	0	-7

- Variable a1 contains the *reference* to the array `{4, 0, -7}`

In Detail

```
int[] a1 = {4, 0, -7};
int[] a2 = a1;
a2[1] = 3;
System.out.println(Arrays.toString(a1));
```

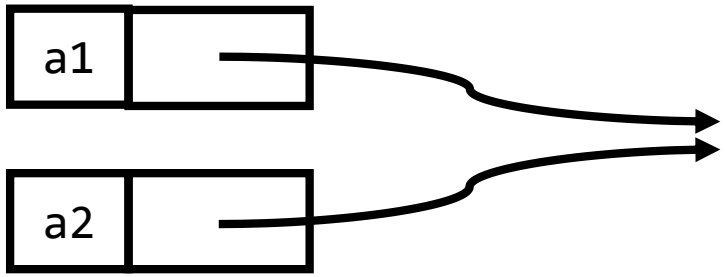


index	0	1	2
value	4	0	-7

- **Variable a2 is assigned the same value as a1**
- **Now a2 also contains the reference to the array `{4, 0, -7}`**
  - Both a1, a2 refer to the same object!

In Detail

```
int[] a1 = {4, 0, -7};
int[] a2 = a1;
a2[1] = 3;
System.out.println(Arrays.toString(a1));
```

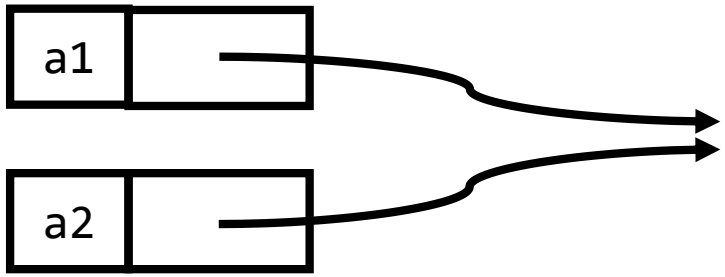


index	0	1	2
value	4	3	-7

- **Change the 1st element of a2 to 3**
  - Now the array has been changed to `{4, 3, -7}`

In Detail

```
int[] a1 = {4, 0, -7};
int[] a2 = a1;
a2[1] = 3;
System.out.println(Arrays.toString(a1));
```



index	0	1	2
value	4	3	-7

- **Print the elements of a1**
  - `[4, 3, -7]`

References and Objects

- **Arrays and objects use *reference semantics***
  - ***Efficiency***: Copying large objects slows down a program
  - ***Sharing***: It's useful to share an object's data among methods

- **When an object is passed as a parameter, the object is not copied, the reference to that object is copied!**
  - Thus the *parameter refers to the same object*
  - If the object (referred by the parameter) is modified, *it will affect the original object*

```
public static void main(String[] args) {
     int[] num = {1, 2, 3};
     add(num);
     System.out.println(Arrays.toString(num));
}

public static void add(int[] a) {
     for(int i = 0; i < a.length; ++i)
          a[i] = a[i] + 100;
}
```

References and Objects

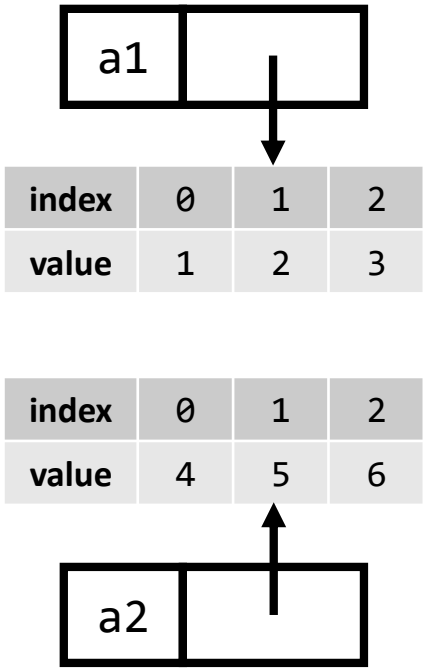
- **Compare these two swap methods**
  - `swapFail` does not swap, while `swap` actually swaps
  - What is the difference?

```
public static void main(String[] args) {
     int[] a1 = {1, 2, 3}, a2 = {4, 5, 6};
     swapFail(a1, a2);
     System.out.println(Arrays.toString(a1));  // [1, 2, 3]
     System.out.println(Arrays.toString(a2));  // [4, 5, 6]
     swap(a1, a2);
     System.out.println(Arrays.toString(a1));  // [4, 5, 6]
     System.out.println(Arrays.toString(a2));  // [1, 2, 3]
}

public static void swapFail(int[] arr1, int[] arr2) {
     int[] tmp = arr2;
     arr2 = arr1; arr1 = tmp;
}

public static void swap(int[] arr1, int[] arr2) {
     for (int i = 0; i < arr1.length; ++i) {
          int tmp = arr2[i];
          arr2[i] = arr1[i]; arr1[i] = tmp;
     }
}
```

swapFail – In Detail

```
public static void main(String[] args) {
     int[] a1 = {1, 2, 3}, a2 = {4, 5, 6};
     swapFail(a1, a2);
     System.out.println(Arrays.toString(a1));  // [1, 2, 3]
     System.out.println(Arrays.toString(a2));  // [4, 5, 6]
}

public static void swapFail(int[] arr1, int[] arr2) {
     int[] tmp = arr2;
     arr2 = arr1;
     arr1 = tmp;
}
```

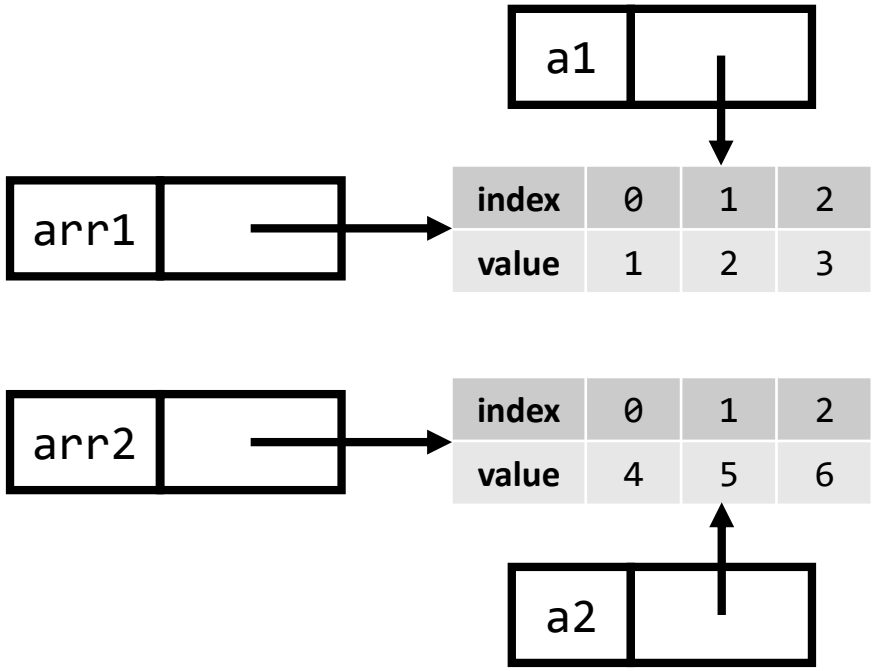
- Initially, a1, a2 refer to the arrays



swapFail – In Detail

```
public static void main(String[] args) {
    int[] a1 = {1, 2, 3}, a2 = {4, 5, 6};
    swapFail(a1, a2);
    System.out.println(Arrays.toString(a1));  // [1, 2, 3]
    System.out.println(Arrays.toString(a2));  // [4, 5, 6]
}

public static void swapFail(int[] arr1, int[] arr2) {
    int[] tmp = arr2;
    arr2 = arr1;
    arr1 = tmp;
}
```

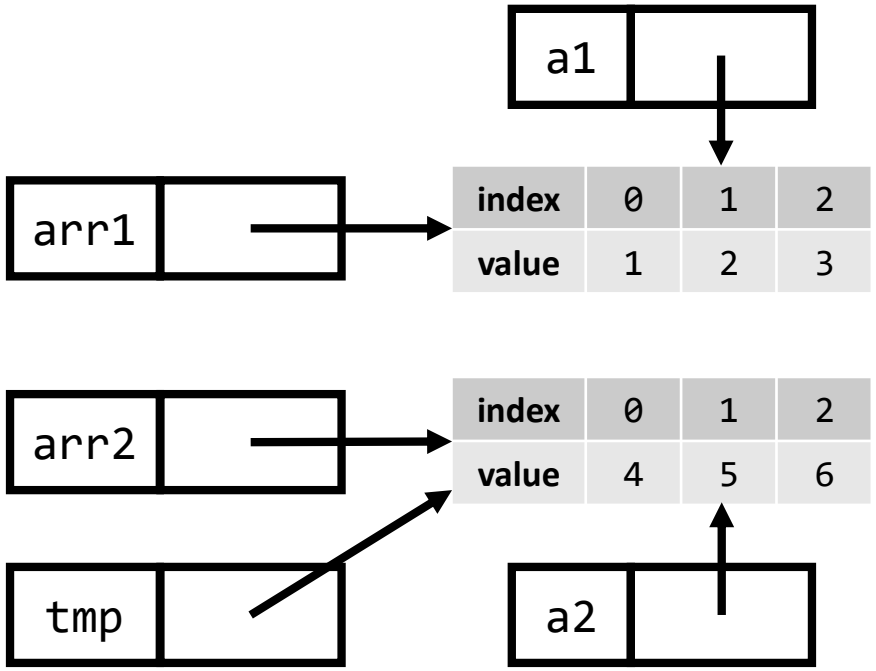
- swapFail is called
- arr1, arr2 also refer to the arrays **(reference is copied)**



swapFail – In Detail

```
public static void main(String[] args) {
    int[] a1 = {1, 2, 3}, a2 = {4, 5, 6};
    swapFail(a1, a2);
    System.out.println(Arrays.toString(a1));  // [1, 2, 3]
    System.out.println(Arrays.toString(a2));  // [4, 5, 6]
}

public static void swapFail(int[] arr1, int[] arr2) {
    int[] tmp = arr2;
    arr2 = arr1;
    arr1 = tmp;
}
```

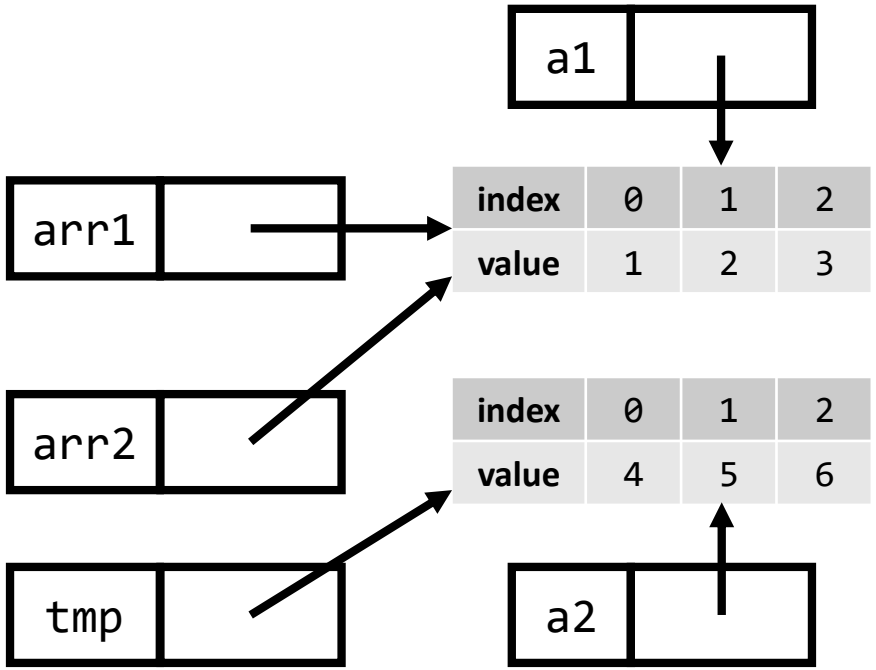
- tmp contains the reference to arr2



swapFail – In Detail

```
public static void main(String[] args) {
    int[] a1 = {1, 2, 3}, a2 = {4, 5, 6};
    swapFail(a1, a2);
    System.out.println(Arrays.toString(a1));  // [1, 2, 3]
    System.out.println(Arrays.toString(a2));  // [4, 5, 6]
}

public static void swapFail(int[] arr1, int[] arr2) {
    int[] tmp = arr2;
    arr2 = arr1;
    arr1 = tmp;
}
```

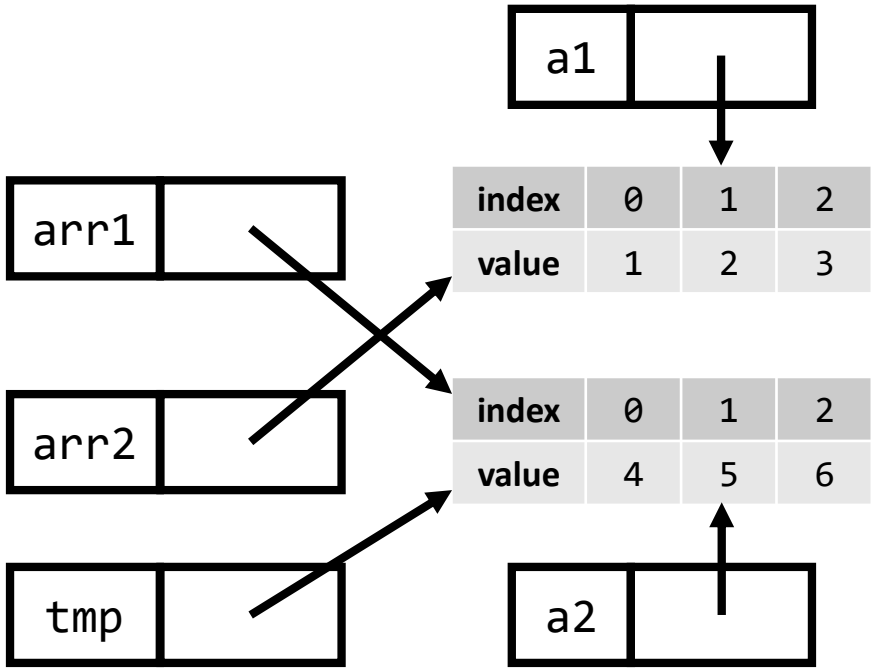
- arr2 contains the reference to arr1



swapFail – In Detail

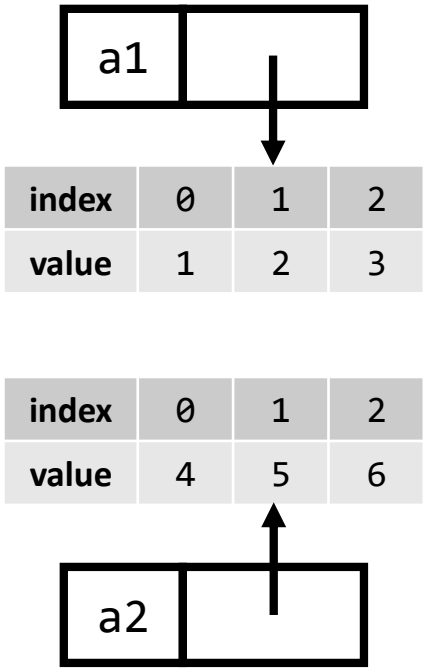
```
public static void main(String[] args) {
    int[] a1 = {1, 2, 3}, a2 = {4, 5, 6};
    swapFail(a1, a2);
    System.out.println(Arrays.toString(a1));  // [1, 2, 3]
    System.out.println(Arrays.toString(a2));  // [4, 5, 6]
}

public static void swapFail(int[] arr1, int[] arr2) {
    int[] tmp = arr2;
    arr2 = arr1;
    arr1 = tmp;
}
```

- arr1 contains the reference to tmp



swapFail – In Detail

```
public static void main(String[] args) {
    int[] a1 = {1, 2, 3}, a2 = {4, 5, 6};
    swapFail(a1, a2);
    System.out.println(Arrays.toString(a1));  // [1, 2, 3]
    System.out.println(Arrays.toString(a2));  // [4, 5, 6]
}

public static void swapFail(int[] arr1, int[] arr2) {
    int[] tmp = arr2;
    arr2 = arr1;
    arr1 = tmp;
}
```

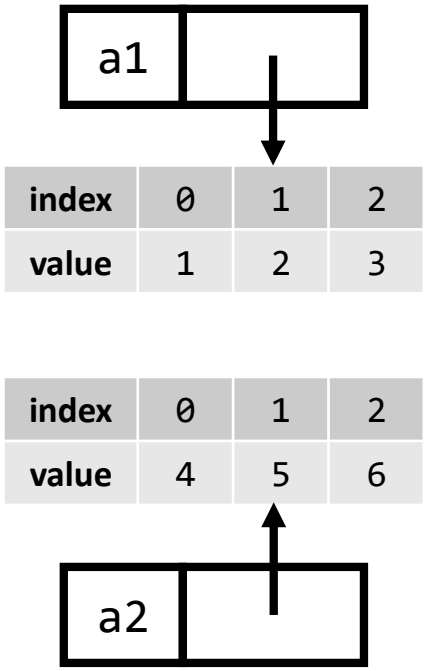


- When `swapFail` is finished, `tmp`, `arr1`, `arr2` all goes away
- **a1, a2 did not change!**

swap – In Detail

```
public static void main(String[] args) {
    int[] a1 = {1, 2, 3}, a2 = {4, 5, 6};
    swap(a1, a2);
    System.out.println(Arrays.toString(a1));  // [4, 5, 6]
    System.out.println(Arrays.toString(a2));  // [1, 2, 3]
}

public static void swap(int[] arr1, int[] arr2) {
    for (int i = 0; i < arr1.length; ++i) {
        int tmp = arr2[i];
        arr2[i] = arr1[i];
        arr1[i] = tmp;
    }
}
```

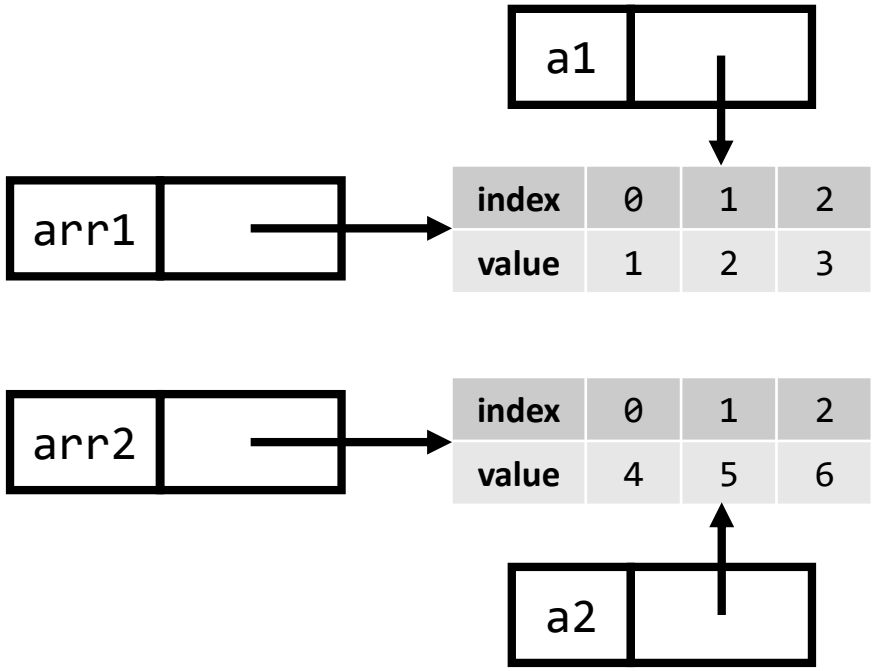
- Initially, a1, a2 refer to the arrays



swap – In Detail

```
public static void main(String[] args) {
    int[] a1 = {1, 2, 3}, a2 = {4, 5, 6};
    swap(a1, a2);
    System.out.println(Arrays.toString(a1));  // [4, 5, 6]
    System.out.println(Arrays.toString(a2));  // [1, 2, 3]
}

public static void swap(int[] arr1, int[] arr2) {
    for (int i = 0; i < arr1.length; ++i) {
        int tmp = arr2[i];
        arr2[i] = arr1[i];
        arr1[i] = tmp;
    }
}
```

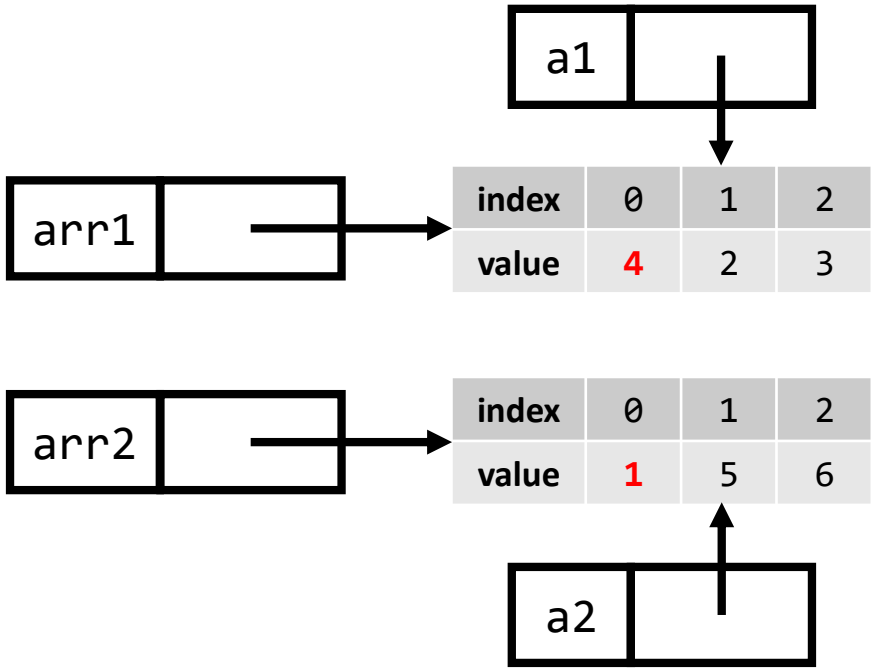
- `swapFail` is called
- `arr1`, `arr2` also refer to the arrays **(reference is copied)**



swap – In Detail

```
public static void main(String[] args) {
    int[] a1 = {1, 2, 3}, a2 = {4, 5, 6};
    swap(a1, a2);
    System.out.println(Arrays.toString(a1));  // [4, 5, 6]
    System.out.println(Arrays.toString(a2));  // [1, 2, 3]
}

public static void swap(int[] arr1, int[] arr2) {
    for (int i = 0; i < arr1.length; ++i) {
        int tmp = arr2[i];
        arr2[i] = arr1[i];
        arr1[i] = tmp;
    }
}
```

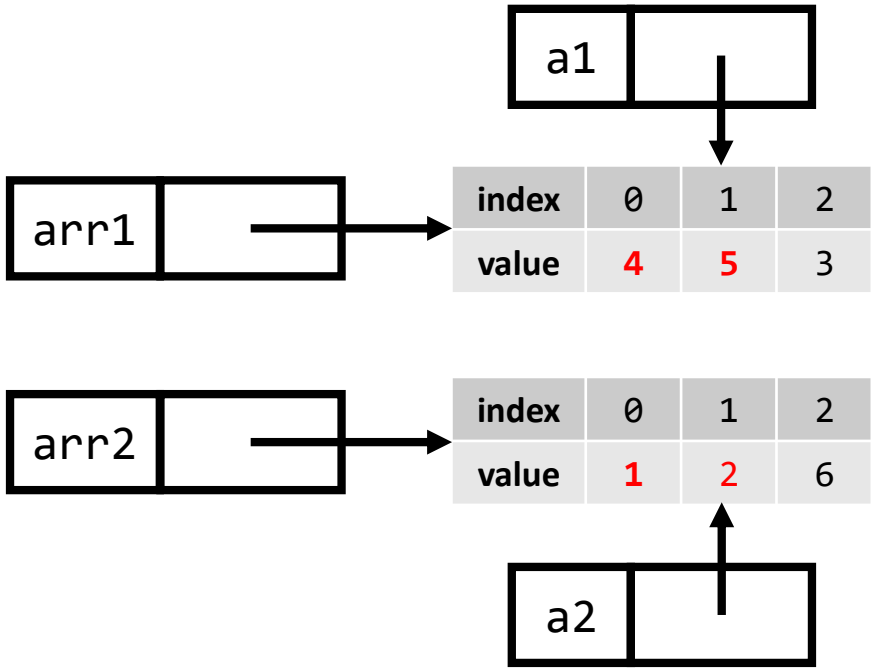
- `for loop (i = 0)`
- Swap 0th element of `arr1` and 0th element of `arr2`



swap – In Detail

```
public static void main(String[] args) {
    int[] a1 = {1, 2, 3}, a2 = {4, 5, 6};
    swap(a1, a2);
    System.out.println(Arrays.toString(a1));  // [4, 5, 6]
    System.out.println(Arrays.toString(a2));  // [1, 2, 3]
}

public static void swap(int[] arr1, int[] arr2) {
    for (int i = 0; i < arr1.length; ++i) {
        int tmp = arr2[i];
        arr2[i] = arr1[i];
        arr1[i] = tmp;
    }
}
```

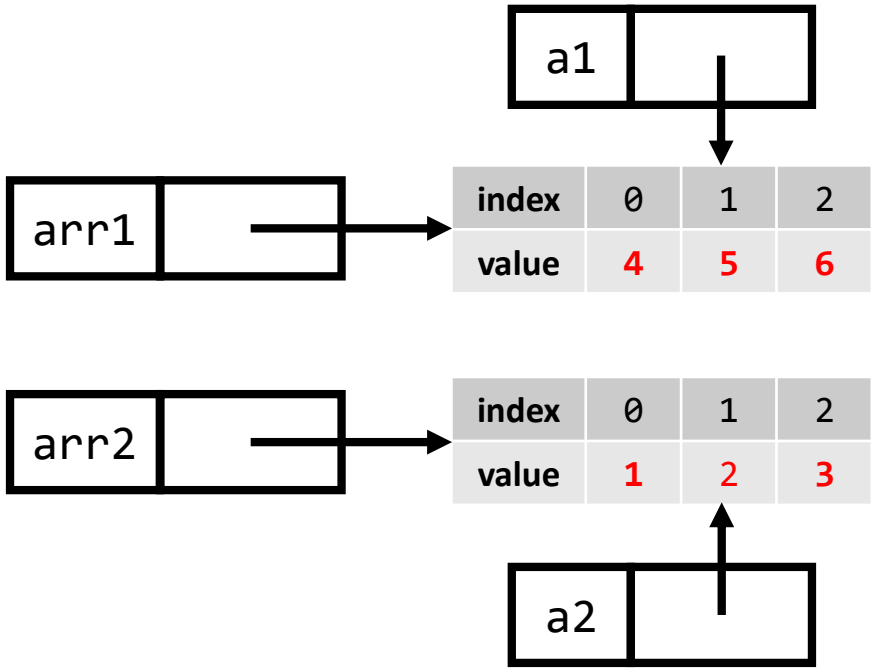
- `for loop (i = 1)`
- Swap 1st element of `arr1` and 1st element of `arr2`



swap – In Detail

```
public static void main(String[] args) {
    int[] a1 = {1, 2, 3}, a2 = {4, 5, 6};
    swap(a1, a2);
    System.out.println(Arrays.toString(a1));  // [4, 5, 6]
    System.out.println(Arrays.toString(a2));  // [1, 2, 3]
}

public static void swap(int[] arr1, int[] arr2) {
    for (int i = 0; i < arr1.length; ++i) {
        int tmp = arr2[i];
        arr2[i] = arr1[i];
        arr1[i] = tmp;
    }
}
```

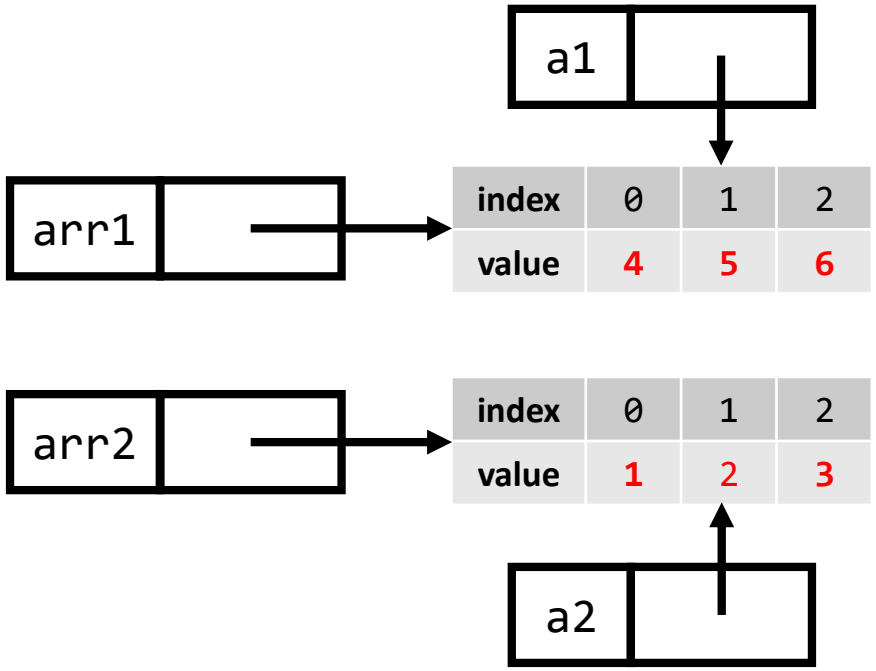
- `for loop (i = 2)`
- Swap 2nd element of `arr1` and 2nd element of `arr2`



swap – In Detail

```
public static void main(String[] args) {
    int[] a1 = {1, 2, 3}, a2 = {4, 5, 6};
    swap(a1, a2);
    System.out.println(Arrays.toString(a1));  // [4, 5, 6]
    System.out.println(Arrays.toString(a2));  // [1, 2, 3]
}

public static void swap(int[] arr1, int[] arr2) {
    for (int i = 0; i < arr1.length; ++i) {
        int tmp = arr2[i];
        arr2[i] = arr1[i];
        arr1[i] = tmp;
    }
}
```

- When swap is finished, `arr1`, `arr2` all goes away
- **a1, a2 changed!**



Points Inside Circle

- You are given coordinates (x, y) of n points on a plane. Also, you are given a circle, by its center (a, b) and its radius r. Print all the points that are inside the given circle (including the border), in given order

- **Input**

```
5               // number of coordinates
0 0             // coordinates of cities
-3 1
3 -5
1 2
-2 -2
3 2             // center of circle
4               // radius of circle
```

- **Output**

```
(0, 0)
(1, 2)
```

Points Inside Circle – Bad Solution

```
public static void main(String[] args) {
     Scanner sc = new Scanner(System.in);
     int n = sc.nextInt();
     int[] x = new int[n];
     int[] y = new int[n];
     for (int i = 0; i < n; ++i) {
          x[i] = sc.nextInt();
          y[i] = sc.nextInt();
     }
     int a = sc.nextInt(), b = sc.nextInt(), r = sc.nextInt();
     for (int i = 0; i < n; ++i) {
          double dist2 = (a - x[i]) * (a - x[i]) + (b - y[i]) * (b - y[i]);
          if (Math.sqrt(dist2) <= r) {
               System.out.println("(" + x[i] + ", " + y[i] + ")");
          }
     }
}
```

- **Need 2 arrays with related data at same indices**
- **Simple, but dirty code...**

Observations

- **The data in this problem is a set of points**
- **Apply OOP!**
  - Want to solve the problem through interaction between objects!
- ***It would be better to store the data as*** `Point` ***objects!***

- ***What are its states/data and behavior?***

Abstraction

- ***abstraction*: A distancing between ideas and details**
  - We can use objects *without knowing how they work*

- **Abstraction in an iPod**
  - You understand its external behavior (buttons, screens, touch)
  - You don't understand its inner details, you don't need to
    - Inner details: circuits, software, etc.

- **We create objects through *abstraction***
  - *Understand the object's state/data and behavior*
  - But *don't need to understand the implementation details*

`Point` Object through Abstraction

- **Our Point class should look like this**
  - **State / Data**
    - A `Point` would store x, y coordinates
  - **Behavior**
    - We can compare distances between `Points` to see if the point is inside the given circle
    - Each `Point` should know how to print itself
- Ignore the implementation details for now. **(abstraction)**
  - *We assume the methods are implemented correctly*
  - *Use the methods as building blocks for bigger programs*

`Point` Object through Abstraction

- **Point object should have x, y as its data**

- **Methods (behavior) should look like this:**

Return Type	methodName(params)	Description
`void`	`setLocation(int x, int y)`	Set the point's x, y to the given values
`void`	`translate(int dx, int dy)`	Adjust the point's x and y by the given amounts
`double`	`distance(Point p)`	Returns the distance from another `Point p`
`void`	`print()`	Prints the point

Object State: Fields

- **field: A variable inside an object *that is part of its state***
  - Each object has *its own copy* of each field

- **Declaration syntax**
  - `type name;`
  - Example: In `Point.java`

```
public class Point {
    int x;
    int y;
}
```

- **The above creates a *new type* named `Point`**
  - Each Point object contains two pieces of data
    - Two integers, x and y
  - The object do not contain any behavior for now

Object State: Fields

- **Other classes can access/modify an object's fields.**
  - *Access*: **`variable.field`**
  - *Modify*: **`variable.field = value;`**

- **Example: In PointMain.java**

```
public class PointMain {
    public static void main(String[] args) {
        Point p1 = new Point();
        System.out.println(p1.x);    // access
        p1.y = 13;                   // modify
        System.out.println(p1.y);
    }
}
```

- **`Point.java` will not have a `main` method, so it's not a runnable program**
  - It will be used by other **client programs**
    - `PointMain.java`, in our case

Modify the Solution

```
public class PointSolution {
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        int n = sc.nextInt();
        Point[] p = new Point[n];
        for (int i = 0; i < n; ++i) {
            p[i] = new Point();
            p[i].x = sc.nextInt();
            p[i].y = sc.nextInt();
        }
        int a = sc.nextInt(), b = sc.nextInt(), r = sc.nextInt();
        for (int i = 0; i < n; ++i) {
            double dist2 = (a - p[i].x) * (a - p[i].x) + (b - p[i].y) * (b - p[i].y);
            if (Math.sqrt(dist2) <= r) {
                System.out.println("(" + p[i].x + ", " + p[i].y + ")");
            }
        }
    }
}
```

- **Declare array of `Points`, initialize and set `x`, `y` fields**
  - Access and use them later in computation

Object Behavior: Methods

- **Our client program wants to print `Point` objects**
- **To print it in other places, the code must be repeated**
  - We should remove the *redundancy* by using a method

- **But if we use a `static` method:**
  - Every program that prints a `Point` object would need this method
  - The syntax wouldn't match, how we're used to using objects
    - `print(p[i]);  // static (bad)`
  - The point of classes is ***to combine state and behavior***
    - The print behavior is closely related to a `Point`'s data
    - The method should belong inside each `Point` object
    - `p[i].print()  // inside object (better)`

Object Behavior: Methods

- ***instance method*** **(*object method*): A method that exists inside each object of a class and gives behavior to each object**
  - Same syntax as static methods, but without `static` keyword

- **Declaration Syntax**

```
public type name(parameters) {
    statements;
}
```

  - Example: In `Point.java`

```
public class Point {
    int x;
    int y;

    public void print() {
        // ?
    }
}
```

Object Behavior: Methods

- **The `print` method no longer has a `Point p` parameter**
  - *How will the method know which point to print?*
    - *How will the method access that point's `x, y` data?*

- **Each `Point` object has its own copy of the print method**
  - It can operate on that object's state!

- ***implicit parameter*: The object on which an instance method is called**
  - In the call `p1.print()`, `p1` is the implicit parameter
  - The instance method *can refer to that object's fields*
    - We say that it executes *in the context of* a particular object
    - `print` can refer to the `x` and `y` of the object it was called on

Object Behavior: Methods

- Now our Point class should look like this

```
public class Point {
    int x;
    int y;

    public void print() {
        System.out.println("(" + x + ", " + y + ")");
    }
}
```

- Now each `Point` object contains a print method that prints the points current position

Kinds of Methods

- ***accessor*: A method that lets clients *examine* object state**
  - Example: `distance`
  - Often has non-void return type

- ***mutator*: A method that *modifies* an object's state**
  - Example: `setLocation`, `translate`

- **Add the methods!**
  - Thanks to *abstraction*, we can only focus on the method itself

Printing Objects

- **By default, Java does not know how to print objects**

```
Point p = new Point();
System.out.println(p);    // Point@2f92e0f4
```

  - So we defined a print method, but `System.out.println(p)` is a lot more *coherent and easier to use* than `p.print()`

- **Every class has a `toString` method, even if it isn't in your code**
  - *Default: class's name @ object's memory address (base 16)*
  - This method is ***automatically called*** when
    - Passed to `System.out.println()`
    - Concatenated with another string

Printing Objects

- **`toString()` syntax**

```
public String toString() {
    // return a String representing the object
}
```

- **Method name, return type, parameters must match exactly**

- **Example:**

```
public String toString() {
    return "(" + x + ", " + y + ")";
}
```

  - Now we can use
    - `System.out.println(p);`

Current Point Class

```
public class Point {
     int x;
     int y;

     public void setLocation(int newX, int newY) {
          x = newX;
          y = newY;
     }

     public void translate(int dx, int dy) {
          x += dx;
          y += dy;
     }

     public double distance(Point p) {
          int dx = x - p.x, dy = y - p.y;
          return Math.sqrt(dx * dx + dy * dy);
     }

     public String toString() {
          return "(" + x + ", " + y + ")";
     }
}
```

Modify the Solution

```
public class PointSolution {
     public static void main(String[] args) {
          Scanner sc = new Scanner(System.in);
          int n = sc.nextInt();
          Point[] p = new Point[n];
          for (int i = 0; i < n; ++i) {
               p[i] = new Point();
               int x = sc.nextInt(), y = sc.nextInt();
               p[i].setLocation(x, y);
          }
          int a = sc.nextInt(), b = sc.nextInt(), r = sc.nextInt();
          Point o = new Point();
          o.setLocation(a, b);
          for (int i = 0; i < n; ++i) {
               if (p[i].distance(o) <= r)
                    System.out.println(p[i]);
          }
     }
}
```

- **A lot cleaner and intuitive!**

Object Initialization: Constructors

- **Currently creating a `Point` and initializing it takes**
  - 3 lines

```
Point p = new Point();
p.x = 3;
p.y = 8;
```

  - 2 lines (thanks to `setLocation` method)

```
Point p = new Point();
p.setLocation(3, 8);
```

- **Can we do better? Why not specify the initial values at the start?**

Object Initialization: Constructors

- ***constructor*: Initializes the state of new objects**
  - Runs when the client uses the **new** keyword
  - No return type is specified
    - It *implicitly* "returns" the new object being created
  - If a class has no constructor, Java gives it a *default constructor*, with no parameters that *sets all fields to "zero-equivalent" value*
  - You can have multiple constructors
    - Each constructor must have a *unique set of parameters*
- **Syntax**

```
public type(parameters) {
    statements;
}
```

Object Initialization: Constructors

- **`Point` class constructor**

```
public class Point {
     int x;
     int y;

     public Point() {   // default constructor
          x = 0;
          y = 0;
     }

     public Point(int initX, int initY) {
          x = initX;
          y = initY;
     }

     // omitted
}
```

Final Solution

```
public class PointSolution {
     public static void main(String[] args) {
          Scanner sc = new Scanner(System.in);
          int n = sc.nextInt();
          Point[] p = new Point[n];
          for (int i = 0; i < n; ++i) {
               int x = sc.nextInt(), y = sc.nextInt();
               p[i] = new Point(x, y);
          }
          int a = sc.nextInt(), b = sc.nextInt(), r = sc.nextInt();
          Point o = new Point(a, b);
          for (int i = 0; i < n; ++i) {
               if (p[i].distance(o) <= r)
                    System.out.println(p[i]);
          }
     }
}
```

Encapsulation

- ***encapsulation*: Hiding implementation details from clients**
  - Encapsulation forces *abstraction*
  - Separates external view (behavior) from internal view
  - Protects the *integrity* of an object's data

- **Benefits**
  - Protects objects from unwanted access
  - Can change the class implementation later
    - `Point` could be rewritten in polar coordinates with the same methods

- This term also means: Combining an object's data and methods into a class

Private Fields

- **A field that *cannot be accessed from outside the class***
  - Used for information hiding
  - Private fields can be accessed *within the class*
- **Declaration**
  - `private type name;`
- **Client code won't compile if it accesses private fields**

```
public class Point {
    private int x;
    private int y;
    ...
}

System.out.println(p1.x);
```

```
Unresolved compilation problem:
    The field Point.x is not visible
```

Acessing Private Fields

- **We use getter/setter to access/modify private variables**
  - These methods can be accessed elsewhere

```
public int getX() {
    return x;
}

public void setX(int newX) {
    x = newX;
}
```

Acessing Private Fields

- **Why do we do such complicated things?**
  - https://stackoverflow.com/a/1568230
  - Encapsulation of behavior associated with getting or setting the property allows additional functionality (like validation) to be added more easily later
  - Hiding the internal representation of the property while exposing a property using an alternative representation.
  - Allowing the public interface to remain constant while the implementation changes without affecting existing consumers.
  - Providing a debugging interception point for when a property changes at runtime - debugging when and where a property changed to a particular value can be quite difficult without this in some languages.
  - Getters and setters can allow different access levels - for example, the get may be public, but the set could be protected.

`this` Keyword

- **`this`: Refers to the implicit parameter *inside your class***
  - Refer to a field: **`this`**`.field`
  - Call a method: **`this`**`.method(parameters)`
  - Call another constructor: **`this`**`(parameters)`

- ***shadowing*: 2 variables with same name in same scope**
  - Normally illegal, except when one variable is a field

```
public class Point {
    private int x;
    private int y;

    public void setLocation(int x, int y) {
        // ...
    }
}
```

  - In other parts of the code, x and y refer to the fields
  - In `setLocation`, x and y refer to the method parameters

this Keyword

- But since x, y are parameters you cannot access the class fields inside `setLocation`

```
public class Point {
    private int x;
    private int y;

    public void setLocation(int x, int y) {
        // cannot access field!
    }
}
```

- Use `this` keyword to refer to the class fields!

```
public void setLocation(int x, int y) {
    this.x = x;
    this.y = y;
}
```

`this` Keyword

- **Can also call another constructor**

```
public Point() {
    this(0, 0);
}

public Point(int x, int y) {
    this.x = x;
    this.y = y;
}
```

  - Avoid redundancy between constructors
  - Only a *constructor can call another constructor*

Static Members

- ***static*: Part of a class, rather than part of an object**
  - Object classes can have static *methods and fields*
  - Not copied into each object, *shared by all objects* of that class

- **Declaration syntax**
  - `static type name;`

- **static field: A field stored in the class, instead of each object**
  - A "*shared*" global field that all objects can access and modify
  - Like a class constant, except that its value can be changed

  - Access by **`ClassName.fieldName`**

Static Fields

- Can count how many objects of that class are created

```
public class Foo {
    private static int count = 0;

    public Foo() {
        count++;
    }

    public int getCount() {
        return count;
    }
}
```

```
public static void main(String[] args) {
    Foo f = new Foo();
    Foo g = new Foo();
    System.out.println(f.getCount());    // 2
    Foo h = new Foo();
    System.out.println(h.getCount());    // 3
}
```

- Can also be used as constants
  - PI, E is a static field in Java's Math class

```
System.out.println(Math.PI);     // 3.141592653589793
System.out.println(Math.E); // 2.718281828459045
```

Static Methods

- ***static method*: A method stored in a class, not in an object**
  - Shared by all objects of the class, not replicated
  - Does not have any *implicit parameter* (this)
    - Cannot access any *particular* object's fields
  - Access by `ClassName.methodName(parameters)`

- **Examples**
  - `Integer.parseInt(String str);`
  - `Arrays.sort(T[] arr);`