# CS 106A, Lecture 17
# More Arrays;
# More Classes and Objects

reading:

*Art & Science of Java*, 12.4

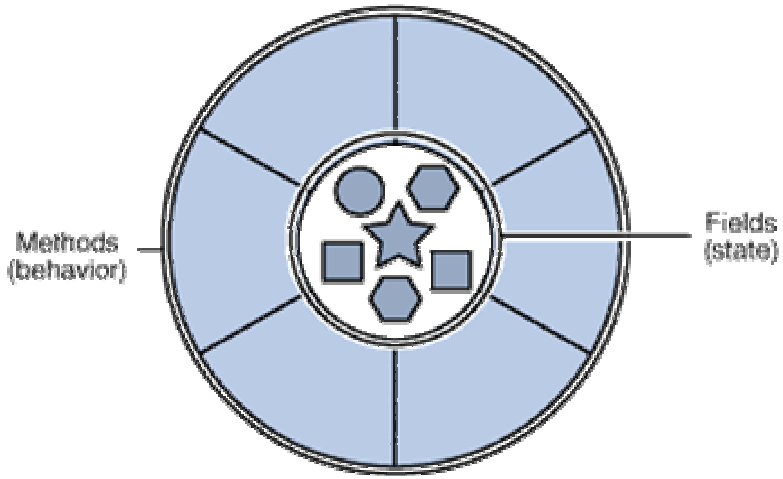
This document is copyright (C) Stanford Computer Science and Marty Stepp, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides created by Keith Schwarz, Mehran Sahami, Eric Roberts, Stuart Reges, and others.

# Lecture Outline

- Today we will learn more about **arrays**.
  - We'll review arrays as **parameters and returns**.
  - We will see an important data concept called **reference semantics**.
  - We will talk about using an array to **tally** data.

- We will also revisit **classes and objects**.

| *index* | *0* | *1* | *2* | *3* | *4* | *5* | *6* |
|---|---|---|---|---|---|---|---|
| *value* | 1 | 7 | 10 | 12 | 8 | 14 | 22 |



# Array as param/return

```
public void name(type[] name) {   // parameter
public type[] name(params)    {   // return
```

– Example:

```
public int sum(int[] a) {
    int result = 0;
    for (int i = 0; i < a.length; i++) {
        result += a[i];
    }
    return result;
}
```

- Call:

```
int[] numbers = {2, -1, 4, 7};
int total = sum(numbers);       // 12
```

# Value semantics

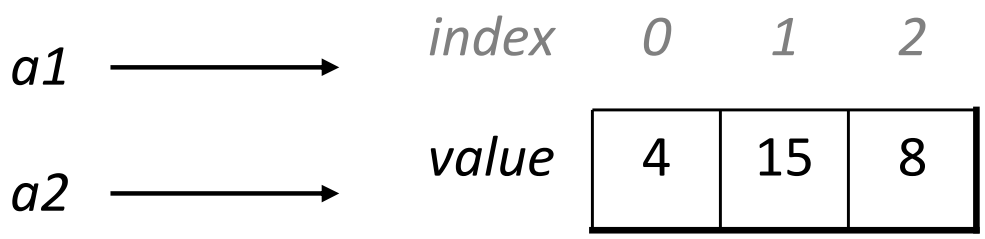
- **value semantics**: Behavior where values are copied when assigned, passed as parameters, or returned. *(used with primitive types)*
  - When a variable is assigned to another, its value is *copied*.
  - Modifying a variable does not affect others.

```
int x = 5;
int y = x;      // x =  5, y =  5   (value of x is copied)

y = 17;         // x =  5, y = 17   (x is not changed)
x = 8;          // x =  8, y = 17   (y is not changed)

x = y;          // x = 17, y = 17   (value of y is copied)
y++;            // x = 17, y = 18   (x is not changed)
x--;            // x = 16, y = 18   (y is not changed)
```

# Reference semantics

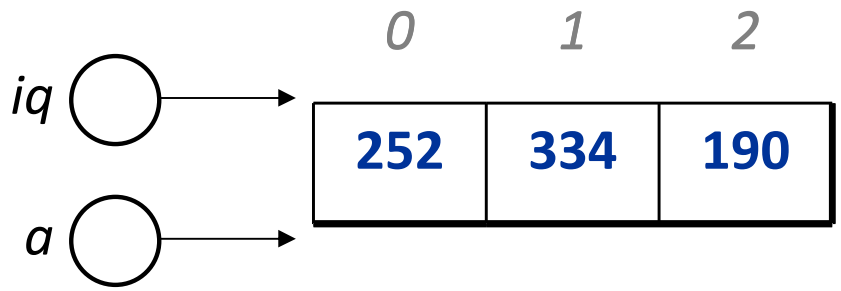
- **reference semantics**: Behavior where variables actually store the address of an object in memory. *(used with arrays/objects)*
  - When a variable is assigned to another, both refer to the *same object*.
  - Modifying the value of one variable ***will*** affect others.

```
int[] a1 = {4, 15, 8};
int[] a2 = a1;                    // refer to same array
a2[0] = 7;                        // (a1/a2 both changed!)
println(Arrays.toString(a1));     // [7, 15, 8]
```



# Pass by reference

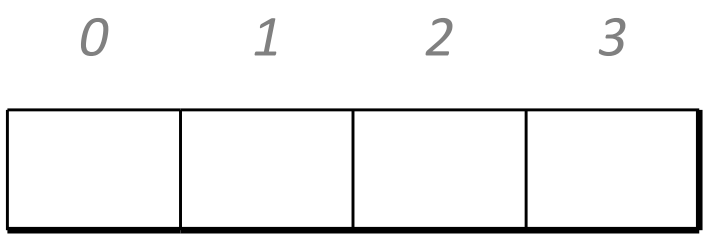
```
// Increases the values in the array by 2x.
public void doubleAll(int[] a) {
    for (int i = 0; i < a.length; i++) {
        a[i] = a[i] * 2;
    }
}

public void run() {
    int[] iq = {126, 167, 95};
    doubleAll(iq);
    println(Arrays.toString(iq));
}   // [252, 334, 190]
```



# Output parameter

- **output parameter**: One passed in by reference to be changed.
  - Its original value is unimportant and will be replaced.

```
// generate a random "password" of digits
public void run() {
    int[] pw = new int[4];
    fillWithRandomDigits(pw);
    println(Arrays.toString(pw));
}

// fills the given array with random values from 0-9
public void fillWithRandomDigits(int[] a) {
    RandomGenerator rg = RandomGenerator.getInstance();
    for (int i = 0; i < a.length; i++) {
        a[i] = rg.nextInt(0, 9);
    }
}
```



# References to objects

- Java objects are stored as **references** (as are arrays).
  - If two variables store the same object, they share that object's data.

```
BankAccount ba1 = new BankAccount("Marty", 1.00);
BankAccount ba2 = new BankAccount("Keith", 3.14);
BankAccount ba3 = ba1;
// ba3 refers to Marty's account

ba3.deposit(4.00);
println(ba1);        // Marty $5.00
println(ba3);        // Marty $5.00
ba1.deposit(2.00);
println(ba1);        // Marty $7.00
println(ba3);        // Marty $7.00
```

ba1 ba3

```
name    = "Marty"
balance = 1.00

deposit(amount) {
    balance += amount;
}
```

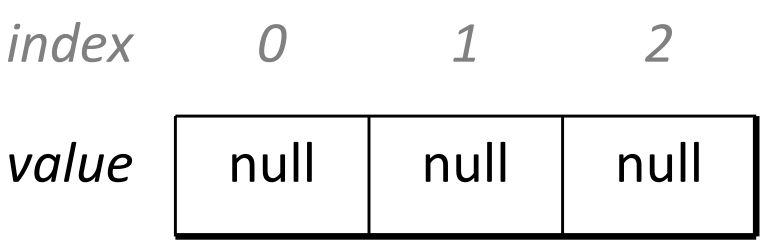
ba2

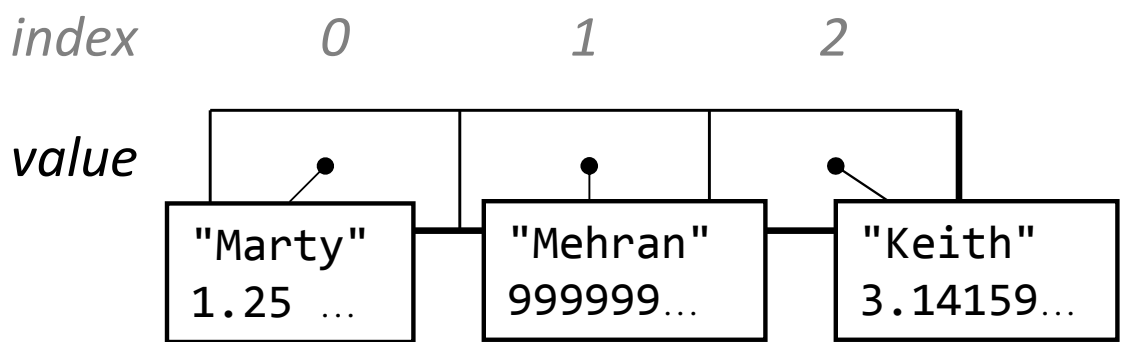
```
name    = "Keith"
balance = 3.14

deposit(amount) {
    balance += amount;
}
```

# Array of objects

- When you make a new array of objects, each element is **`null`**.
  - null is not any object. Can't call methods e.g. `deposit`.
  - Must also initialize each element of the array to store an object.

```
BankAccount[] accounts = new BankAccount[3];
println(Arrays.toString(accounts));  // [null, null, null]

// initialize each account in the array
accounts[0] = new BankAccount("Marty", 1.25);
accounts[1] = new BankAccount("Mehran", 999999.00);
accounts[2] = new BankAccount("Keith", 3.14159);
```



# Array param exercise

switchPairs

- Write a method **`switchPairs`** that accepts an arrays of strings and swaps the values of each neighboring pair of elements.
  - Swap elements 0-1, and 2-3, and 4-5, and ...
  - If the array length is odd, leave the last element untouched.

```
String[] names = {"Ash", "Em", "Nick", "Stu", "Tim"};
switchPairs(names);
println(Arrays.toString(names));

                    //    0      1       2       3        4
                    // {"Em", "Ash", "Stu", "Nick", "Tim"}
```

# Array parameter answer

```
// Swaps the values of neighboring elements.
public void switchPairs(String[] a) {
    for (int i = 0; i < a1.length - 1; i += 2) {
        String temp = a1[i];
        a[i] = a[i + 1];
        a[i + 1] = temp;
    }
}
```

# Array exercise…?



- Write a method **`mostFrequentDigit`** that accepts an integer and returns the digit value from 0-9 that appears most frequently.
  - example:

    **`mostFrequentDigit(340331378)  returns 3`**

# Idea: Array of counters

- For problems like this, where we want to keep count of many things, a *lookup array* (or *tally array*) can be a clever solution.
  - *Idea:* The element at index *i* will store a counter for the digit value *i*.
  - example: count of digits in 340331378:

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

# Classes and Objects, continued

This document is copyright (C) Stanford Computer Science and Marty Stepp, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides created by Keith Schwarz, Mehran Sahami, Eric Roberts, Stuart Reges, and others.

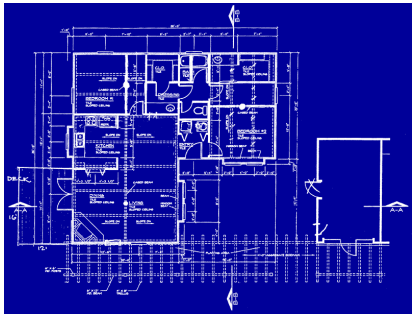
# Classes and objects

- **class**: A template for a new type of objects.
- **object**: An entity that combines state and behavior.

**iPod blueprint (class)**

**state:**
current song
volume
battery life

**behavior:**
power on/off
change station/song
change volume
choose random song



*constructs*

**iPod (object) #1**

**state:**
song = "10000 Miles"
volume = 17
battery life = 2.5 hrs

**behavior:**
power on/off
change station/song
change volume
choose random song

**iPod (object) #2**

**state:**
song = "Letting You"
volume = 9
battery life = 3.41 hrs

**behavior:**
power on/off
change station/song
change volume
choose random song

**iPod (object) #3**

**state:**
song = "Discipline"
volume = 24
battery life = 1.8 hrs

**behavior:**
power on/off
change station/song
change volume
choose random song

# Fields

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

- Declaration syntax:

  `private` ***`type name`***`;`

  - access level is typically `private`, but can be `public` or others

  - Example:

```
public class BankAccount {
    private String name;      // each account object has
    private double balance;   // a name and balance field
    ...
```

# Instance methods

- **instance method** (or **object method**): Exists inside each object of a class, and gives behavior to each object.

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

  - access level is usually `public`, but can be `private` or others

  – Example (in `BankAccount` class):

```
public void deposit(double amount) {
    balance += amount;
}
```

# Using object methods

```
BankAccount ba1 = new BankAccount();
ba1.name = "Marty";
ba1.balance = 1.25;

BankAccount ba2 = new BankAccount();
ba2.name = "Mehran";
ba2.balance = 900000.00;

ba1.deposit(0.20);
ba2.deposit(1000.00);
```

ba1

```
name    = "Marty"
balance = 1.45

deposit(amount) {
    balance += amount;
}
```

ba2

```
name    = "Mehran"
balance = 901000.00

deposit(amount) {
    balance += amount;
}
```

- When you call an object's method:
  - It executes <u>that object's copy</u> of the code from the class.
  - If that code refers to fields, it means <u>that object's copy</u> of those fields.
  - So calling the method on different objects has different effects.

# Constructors

- **constructor**: Initializes the state of new objects as they are created.
  - The constructor runs when the client says new `ClassName(...);`

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

  - <u>no return type</u> is specified; it "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 default values like `0` or `null`.

# Using constructors

```
BankAccount ba1 =
    new BankAccount("Marty", 1.25);
```

ba1

```
name    = "Marty"
balance = 1.25

BankAccount(nm, bal) {
    name = nm;
    balance = bal;
}
```

```
BankAccount ba2 =
    new BankAccount("Mehran", 900000.00);
```

ba2

```
name    = "Mehran"
balance = 900000.00

BankAccount(nm, bal) {
    name = nm;
    balance = bal;
}
```

- When you call a constructor (with new):
  - Java creates a new object of that class.
  - The constructor runs, with that new object as the implicit parameter.
  - The newly created object is returned to your program.

# Array as field

```
private type[] name;        // declare
...
name = new type[length];    // initialize (in constructor)
```

– Example:

```
// Represents a hand in the card game Uno.
public class UnoHand {
    private String[] cards;

    public UnoHand() {
        cards = new String[7];
        ...
    }
}
```

# Printing objects

- By default, Java doesn't know how to print objects.

```
// ba1 is BankAccount@9e8c34
BankAccount ba1 = new BankAccount("Marty", 1.25);
println("ba1 is " + ba1);


// better, but cumbersome to write
// ba1 is Marty with $1.25
println("ba1 is " + ba1.getName() + " with $"
        + ba1.getBalance());


// desired behavior
println("b1 is " + ba1);   // ba1 is Marty with $1.25
```

# The toString method

*tells Java how to convert an object into a string*

```
BankAccount ba1 = new BankAccount("Marty", 1.25);
println("ba1 is " + ba1);

// the above code is really calling the following:
println("ba1 is " + ba1.toString());
```

- Every class has a `toString`, even if it isn't in your code.
  - Default: class's name @ object's memory address (base 16)

```
BankAccount@9e8c34
```

# toString syntax

```
public String toString() {
    code that returns a String
    representing this object;
}
```

– Method name, return, and parameters must match exactly.

– Example:

```
// Returns a String representing this account.
public String toString() {
    return name + " has $" + balance;
}
```

# The keyword this

- **`this`** : A reference to the implicit parameter.
  - *implicit parameter:* object on which a method is called

- Syntax for using `this` :
  - To refer to a field:

    `this.`***`field`***

  - To call a method:

    `this.`***`method`***`(`***`parameters`***`);`

  - To call a constructor from another constructor:

    `this(`***`parameters`***`);`

# Variable names/scope

- Usually illegal to have 2 variables in same scope with same name:

```
public class BankAccount {
    private double balance;
    private String name;
    ...

    public void setName(String newName) {
        name = newName;
    }
}
```

  – The parameter to `setName` is named `newName` to be distinct from the object's field `name` .

# Variable shadowing

- An instance method parameter name can match a field name:

```
public class BankAccount {
    private double balance;
    private String name;
    ...

    public void setName(String name) {
        this.name = name;
    }
}
```

  - Field name is *shadowed* by the parameter with the same name.
  - Any code inside `setName` that refers to name will use the parameter, not the field. To refer to the field, say `this.name` .

# Multiple constructors

- It is legal to have more than one constructor in a class.
  - The constructors must accept different parameters.

```
public class BankAccount {
    private double balance;
    private String name;

    public BankAccount(String name) {
        this.name = name;
        balance = 0.00;
    }

    public BankAccount(String name, double bal) {
        this.name = name;
        balance = bal;
    }
    ...
}
```

# Multiple constructors

- One constructor can call another using `this` :

```
public class BankAccount {
    private double balance;
    private String name;

    public BankAccount(String name) {
        this(name, 0.00); // call other constructor
    }

    public BankAccount(String name, double bal) {
        this.name = name;
        balance = bal;
    }
    ...
}
```

# Overflow (extra) slides

This document is copyright (C) Stanford Computer Science and Marty Stepp, licensed under Creative Commons Attribution 2.5 License. All rights reserved.
Based on slides created by Keith Schwarz, Mehran Sahami, Eric Roberts, Stuart Reges, and others.

# Swapping values

```
public void run() {
    int a = 7;
    int b = 35;

    // swap a with b?
    a = b;
    b = a;

    println(a + " " + b);
}
```

– What is wrong with this code? What is its output?

- The red code should be replaced with:

```
int temp = a;
a = b;
b = temp;
```

# A swap method?

- Does the following swap method work? Why or why not?

```
public void run() {
    int a = 7;
    int b = 35;

    // swap a with b?
    swap(a, b);

    println(a + " " + b);
}

public void swap(int a, int b) {
    int temp = a;
    a = b;
    b = temp;
}
```

# Array param questions

swap
swapAll

- Write a method **swap** that accepts an arrays of integers and two indexes and swaps the elements at those indexes.

```
int[] a1 = {12, 34, 56};
swap(a1, 1, 2);
println(Arrays.toString(a1));  // [12, 56, 34]
```

- Write a method **swapAll** that accepts two arrays of integers as parameters and swaps their entire contents.
  - Assume that the two arrays are the same length.

```
int[] a1 = {12, 34, 56};
int[] a2 = {20, 50, 80};
swapAll(a1, a2);
println(Arrays.toString(a1));  // [20, 50, 80]
println(Arrays.toString(a2));  // [12, 34, 56]
```

# Array parameter answers

```
// Swaps the values at the given two indexes.
public void swap(int[] a, int i, int j) {
    int temp = a[i];
    a[i] = a[j];
    a[j] = temp;
}

// Swaps the entire contents of a1 with those of a2.
public void swapAll(int[] a1, int[] a2) {
    for (int i = 0; i < a1.length; i++) {
        int temp = a1[i];
        a1[i] = a2[i];
        a2[i] = temp;
    }
}
```

# Array return question


- Write a method **`merge`** that accepts two arrays of integers and returns a new array containing all elements of the first array followed by all elements of the second.

```
int[] a1 = {12, 34, 56};
int[] a2 = {7, 8, 9, 10};

int[] a3 = merge(a1, a2);
println(Arrays.toString(a3));
// [12, 34, 56, 7, 8, 9, 10]
```

# Array return answer 1

```
// Returns a new array containing all elements of a1
// followed by all elements of a2.
public int[] merge(int[] a1, int[] a2) {
    int[] result = new int[a1.length + a2.length];

    for (int i = 0; i < a1.length; i++) {
        result[i] = a1[i];
    }
    for (int i = 0; i < a2.length; i++) {
        result[a1.length + i] = a2[i];
    }

    return result;
}
```

# Histogram question



- Given a file of integer exam scores, such as:

```
82
66
79
63
83
```

- Write a **`Histogram`** program that will print a histogram of stars indicating the # of students who earned each unique exam score.

```
85: *****
86: ************
87: ***
88: *
91: ****
```

# Idea: Array of counters

- For problems like this, where we want to keep count of many things, a *lookup array* (or *tally array*) can be a clever solution.
  - *Idea:* The element at index *i* will store a counter for the digit value *i*.
  - Example: myArray[87] stores number of people who scored 87.
  - Example: If myArray[93] stores 4, that means 4 people scored 93.

| *index* | *0* | *1* | *2* | *...* | *91* | *92* | *93* | *94* | *95* | *96* | *97* | *98* | *99* | *100* |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| *value* | 4 | 0 | 0 | ... | 0 | 2 | 4 | 1 | 0 | 3 | 1 | 0 | 7 | 2 |

# Histogram solution 1

```
public class Histogram extends ConsoleProgram {
    public void run() {
        int[] counts = readScores("midterm.txt");
        displayHistogram(counts);
    }

    /* Reads integer scores from the given file and returns
     * an array histogram where index [i] stores how many people
     * got a score of 'i' on the exam. */
    public int[] readScores(String filename) {
        int[] counts = new int[101];     // count scores 0-100
        try {
            Scanner input = new Scanner(new File(filename));
            while (input.hasNextInt()) {     // read into array
                int score = input.nextInt();
                counts[score]++;
            }
        } catch (FileNotFoundException fnfe) {
            println("I/O error: " + fnfe);
        }
        return counts;
    }
    ...
```

# Histogram solution 2

```
    ...

    /* Prints a histogram of stars, one for each student who
     * got a given score on the exam. */
    public void displayHistogram(int[] counts) {
        for (int i = 0; i < counts.length; i++) {
            // print a star for each student who scored 'i'
            if (counts[i] > 0) {
                print(i + ": ");
                for (int j = 0; j < counts[i]; j++) {
                    print("*");
                }
                println();
            }
        }
    }
}
```