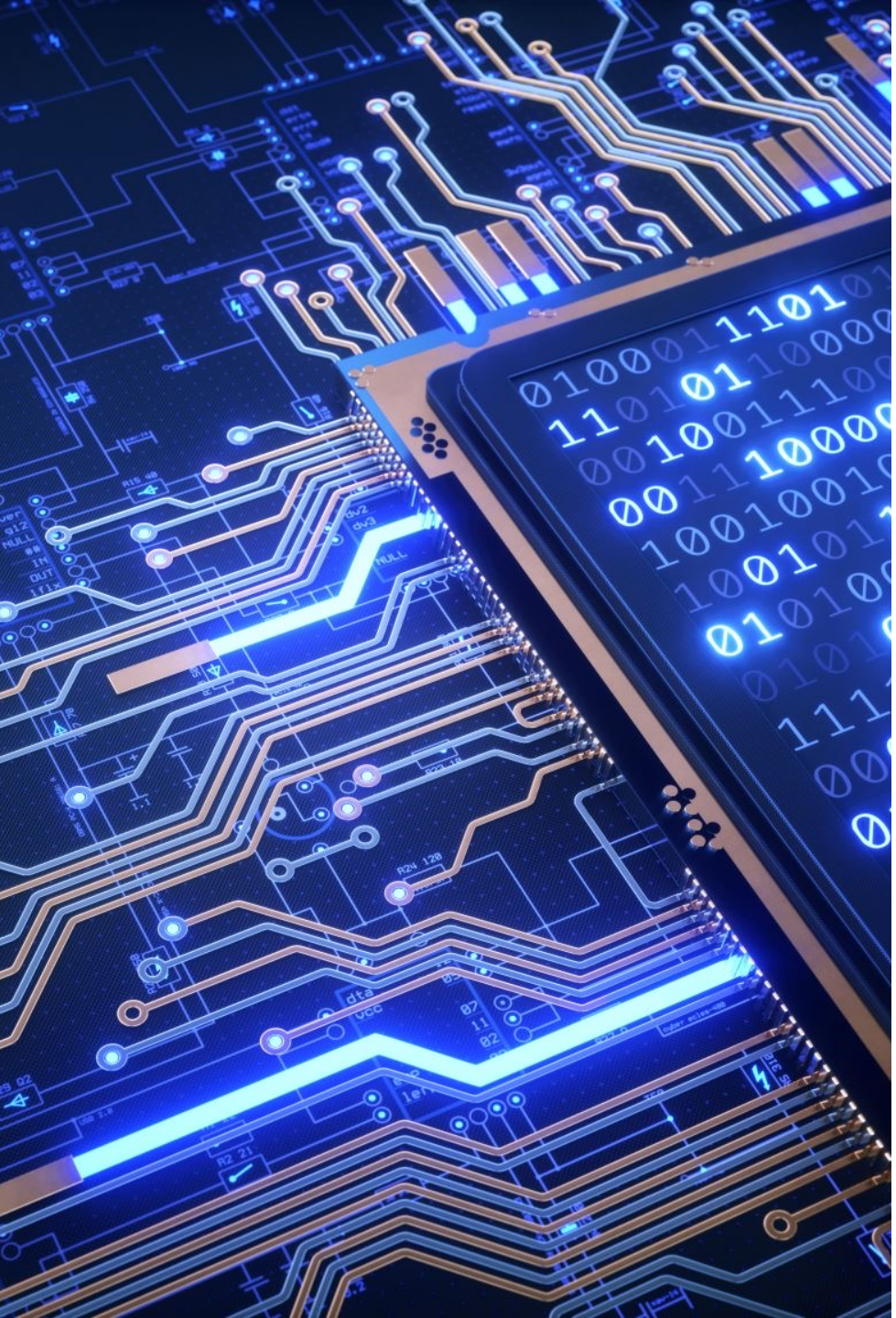
CMPS 200: Introduction to Programming Using JAVA

LECTURE 11 – Classes and Objects

Maurice J. KHABBAZ, Ph.D.



Last Time

`Scanner` as an Iterator:

`String` Processing.

`StringTokenizer`

Input / Output.

Exceptions.

Reading From Files.

Writing to Files.

File Processing.

Today

Object-Oriented Programming

- Recapitulation on Objects.
- Advantages of OOP.
- Class-Object Relationships.
- Anatomy of a Class.
- Instantiation of Objects.
- Client/Driver Programs.

Encapsulation

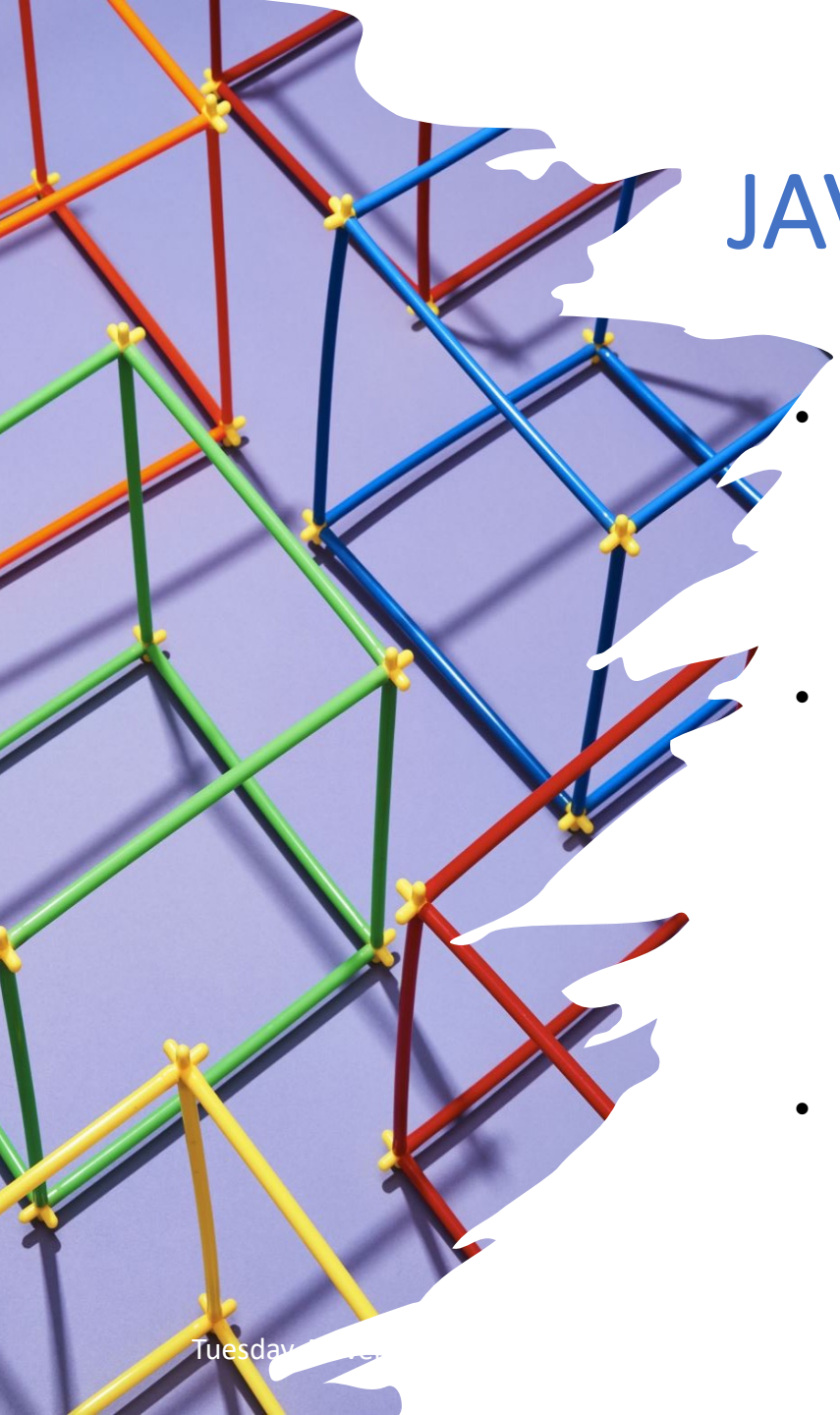
- Accessor/Mutator and Custom-Built Methods.
- Print Representation of an Object.
- Method Overriding and Overloading.
- Variable Shadowing.

Arrays of Objects (Revisited)

- `null` Pointer.
- Dereferencing and `NullPointerException`.

`static` Members of a Class

Classes as Modules



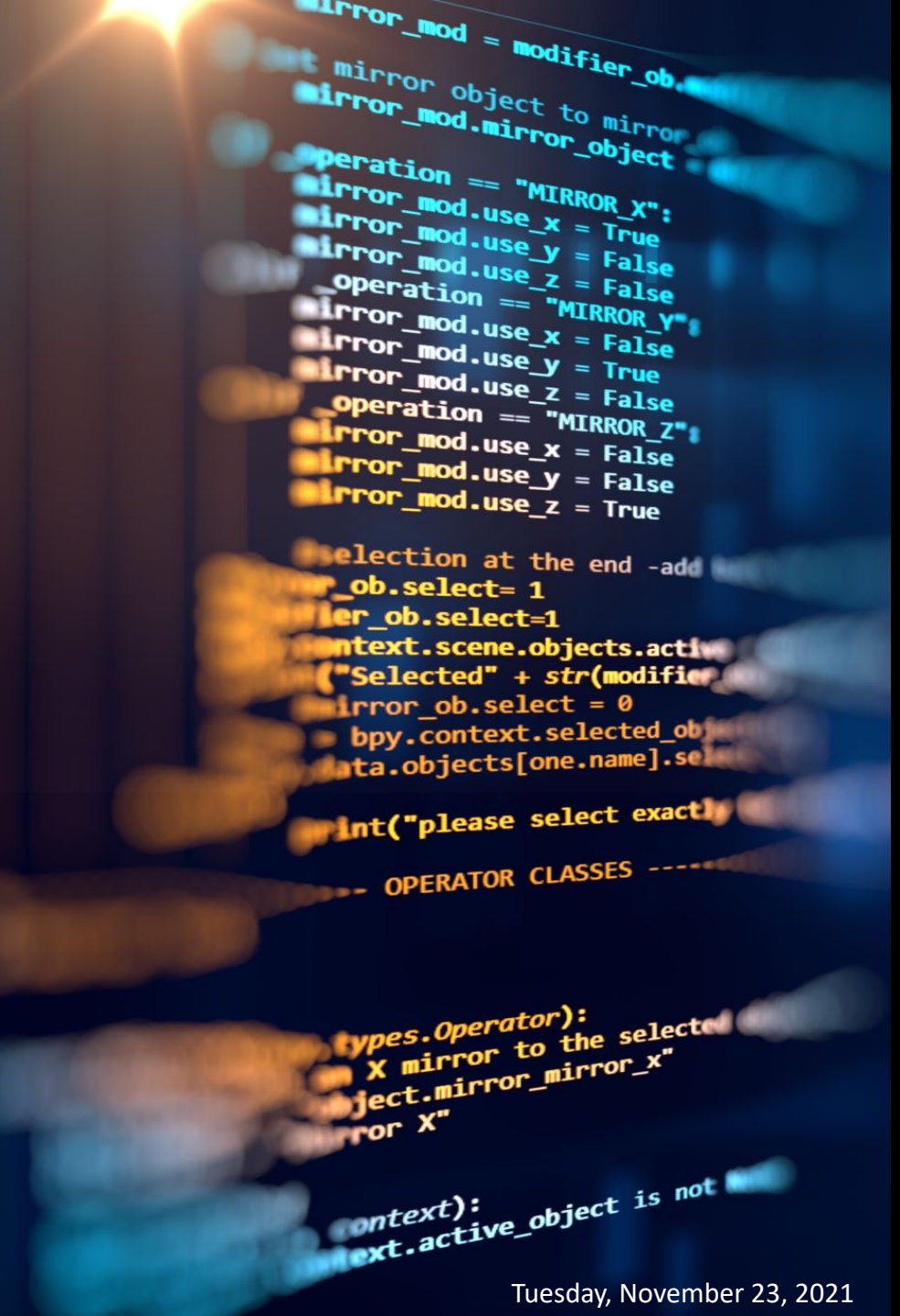
JAVA Is About Objects and Their Types

- So far, known data types: `int`, `float`, `String`, `arrays`, `ArrayList...`:
  - **Examples** of **objects** of these types:

    `1234` `2.1` `"hi"` `[1, 2]` `[1, 2, 3]`

- Each object has:
  - A **type** established through a defined **class**.
  - Possibly (i.e. if non-scalar) an internal **data representation** (primitive/composite):
    - Hidden for the sake of **abstraction**.
  - A set of procedures/methods/functions for **interaction** with the (other) object(s).

- An object is an **instance** of a type/class:
  - `1234` is an instance of an `int`.
  - `"hi"` is an instance of a `String`.



Object-Oriented Programming

- Thus far, only pre-defined classes have been used in programs:
  - **Examples:** `String`, `Math`, `Scanner`, `File...`
- Such classes have been pre-defined in the JAVA Standard Class Library:
  - **Examples:** `java.lang`, `java.util`, `java.io...`
- It's about time to learn how to write **custom-built classes**:
  - To define custom object types → **Essence of OOP:**
    - Define how to instantiate an object from a custom-built type.
    - Define how to interact with such an object.
    - Manipulate objects.
    - Delete/destroy objects:
      - **Explicitly** nulling them (assigning `null` to their variables).
      - Just "forget" about them.
      - Destroyed/inaccessible are "garbage":
        - Reclaimed through "garbage collection".
  - Need to suite **specific requirements**.

Advantages of OOP

Bundling/Packaging Data

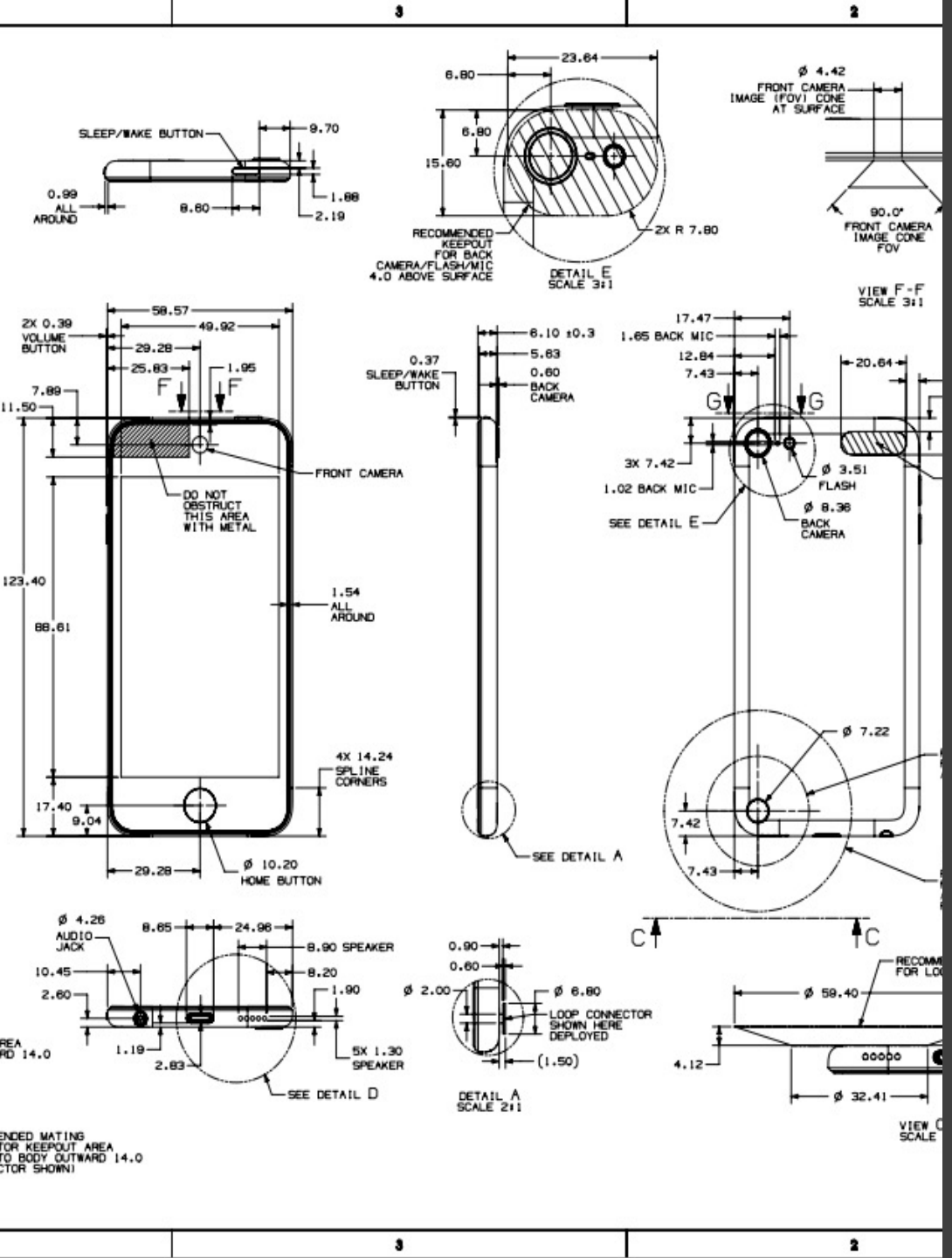
- Values + methods to work on them through well-defined interfaces.

Divide-and-Conquer Development:

- Implement and test behavior of each **class/object** separately.
- Increased modularity reduces complexity.

Ease of Code Reuse

- Each class has a **separate environment** (no collision).
- **Inheritance** allows subclasses:
  - To **re-define/extend** a selected subset of a **superclass'** behavior.
  - More on that later …



Relationship Between an Object and a Class

CLASS

- **Blueprint**/**template** used to create specific objects:
  - Defines object **properties**/**attributes**:
    - **Descriptive** attributes → **vars**/**fields** declared within class.
      - Indicate the **state**/**status** of instantiated objects using this class.
      - Each instantiated object will have **its own copy** of these fields.
    - **Behavioral** attributes → **methods** defined within class:
      - **Interface** for user-to-object and object-to-object interaction.
      - Hide the implementation.
- Formal representation of a certain **object concept**.

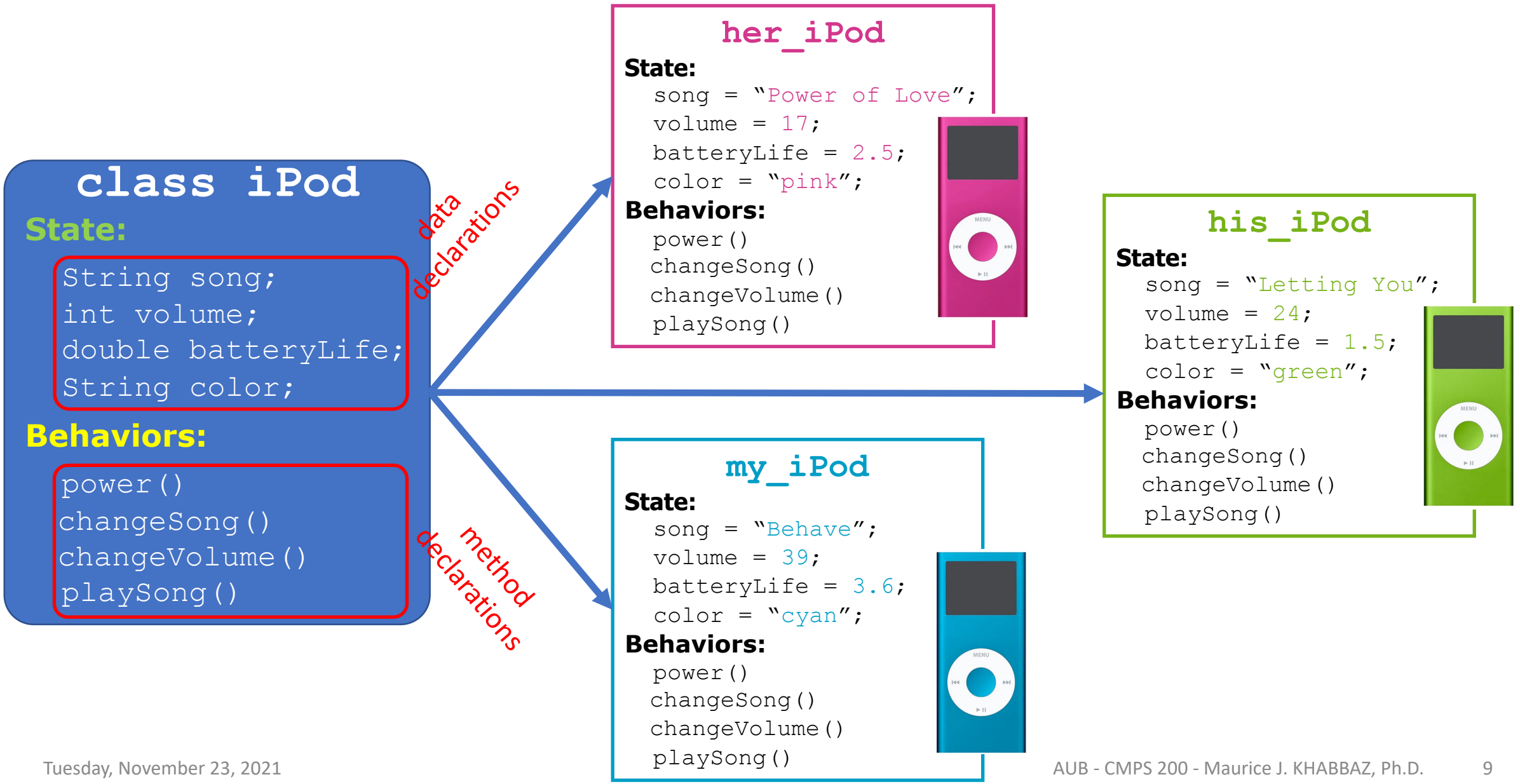
OBJECT

- An **instance** created using a certain class.
- **Abstract realization** of the concept defined by the class.
- **State:** defined by values assigned to descriptive attributes.

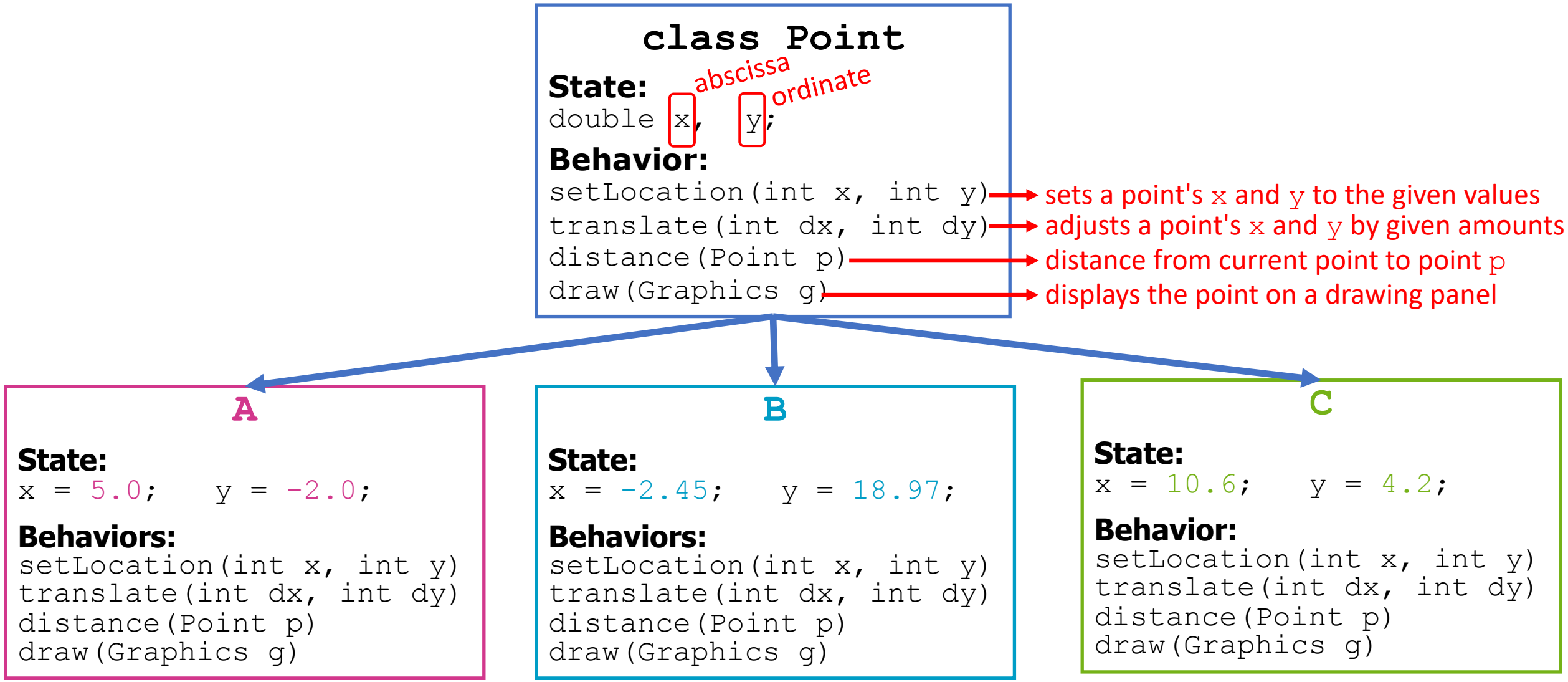
Examples: Classes, Attributes and Behaviors

Class Name	Attributes	Behaviors / Operations
Rectangle	length width color	setLength() setWidth() setColor
Flight	airline flightNumber originCity destinationCity	setAirline() setFlightNumber() setOrigin() setDestination()
Employee	name company salary	setName() setCompany() computeBonus() computeTaxes()

Example 1: `iPod` Class And Its Object Instances



Example 2: `Point` Class And Its Object Instances



Important to Distinguish Between

- **Creating a class** → Defining the class' name and attributes (**no behaviors yet**):

class visibility — keyword for class definition

```
public class <c_name> {
    <visibility> <type1> <v1>;
    ...
}
```

- if omitted or `public`, allows direct access
- other visibility modifiers later …

At class definition point attribute variables are only declared with no memory reservation

save this code in a file that **must be called** `<c_name>.java`

- **Instantiating Objects** → Creating new **instances of objects** of this type/class:

```
<c_name> <o_name> = new <c_name>(<params>)
```

- **Operating** on the newly created instances:

```
System.out.println(<o_name>.<var>); // Direct access
<o_name>.<var> = <val>;             // Direct modification
```

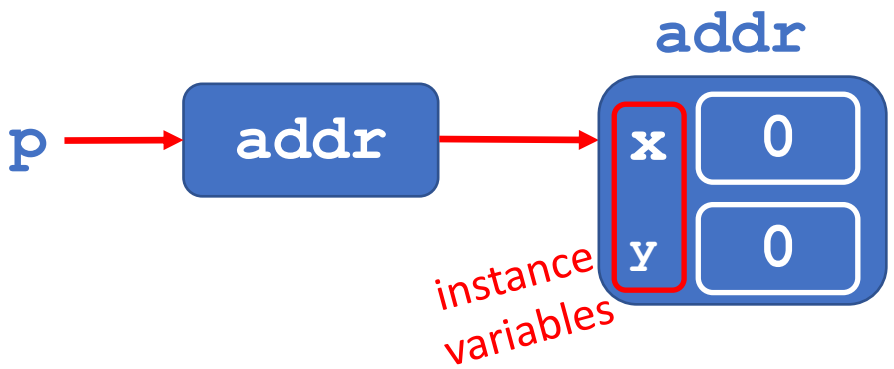
Example 1: A Class and Its Client/Driver Program

Point.java
(new custom class of objects)

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

attribute variables declared only

Creates a new object p of type Point
Initializes x and y to default values (0, 0)



PointClient.java
(Client program for Point class)

```
public class PointClient {
  public static void main(String args[])
  {
    Point p = new Point();
    System.out.println("x = " + p.x);
    System.out.println("y = " + p.y);
    p.x = 7; p.y = 2;
    System.out.println("x = " + p.x);
    System.out.println("y = " + p.y);
  }
}
```

Example 1: A Class and Its Client/Driver Program

Point.java
(new custom class of objects)

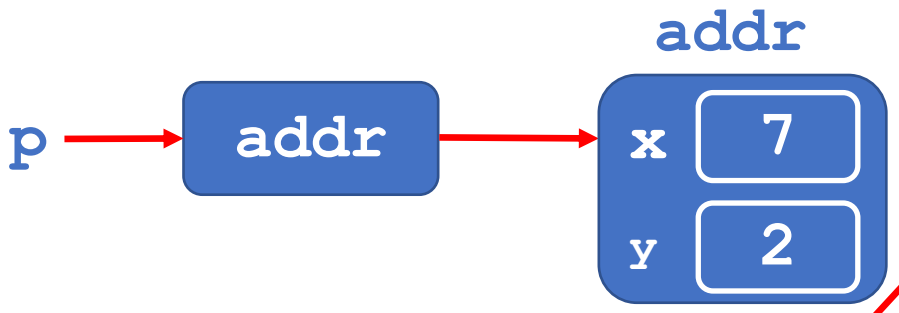
```
public class Point {
    int x;
    int y;
}
```

PointClient.java
(Client program for Point class)

```
public class PointClient {
  public static void main(String args[])
  {
    Point p = new Point();
    System.out.println("x = " + p.x);
    System.out.println("y = " + p.y);
    p.x = 7; p.y = 2;
    System.out.println("x = " + p.x);
    System.out.println("y = " + p.y);
  }
}
```

Accessing (printing out) initial attributes

Modifying (assign new values) attributes



Printing out modified attributes

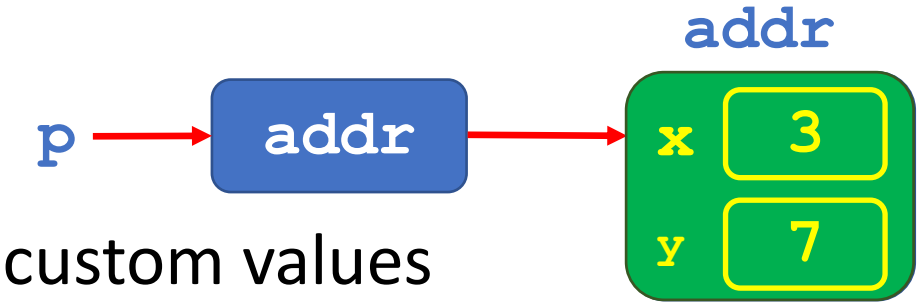
Object Instantiation Procedure

- **Q:** Given a class, how is a new object instance of that class created?
- **Analogy:** Given the blueprint of a building, who creates the building?
- **A:** The **constructor**!
- **In JAVA:**
  - Every class must contain a special method called the **constructor method**:
    - Runs whenever a client program attempts to **create an object instance** of a class.
    - Has the **exact same name** of the class with **`public`** visibility.
    - Has **no specified return type**.
    - It implicitly creates and returns the newly created object:
      - **Reserves appropriate memory space** to fit all the object's state variables.
      - **Creates copies of these state variables and initializes them** to given/default values.
  - If a class has no constructor method, JAVA implicitly creates one for that class:
    - Initializes all state variables/fields to appropriate **default values**.

Example 2: Constructor

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

- Recall the `Point` class:
  - Has no explicitly defined constructor.
  - JAVA creates a default constructor:
    - All instance variables initialized to default values.
  - **Client:** `Point p = new Point();`



- **Requirement:** initialize instance variables to custom values
  - Modify class to integrate an explicit constructor.

```
public class Point {
    int x; // Attribute variable
    int y; // Attribute variable
    // Class Constructor Method:
    public Point(int init_x, int init_y) {
        x = init_x; y = init_y;
    }
}
```

Client: `Point p = new Point(3, 7);`

Common Constructor Bugs

- **Shadowing:** re-declaring fields as **local variables**:

```
public Point(int x_init, int y_init) {
    int x = x_init;
    int y = y_init;
}
```

  - This **declares local variables with the same name as the fields**, rather than storing values into the fields.
  - **The fields' values remain 0.**

- **Accidental Association Of Return Type:**

```
public void Point(int x_init, int y_init) {
    x = x_init;
    y = y_init;
}
```

  - This is actually **not a constructor**, but a method named `Point`.

Encapsultation to Enforce Privacy and Abstraction

- Encapsulation aims at:
  - **Hiding implementation** details from clients.
  - Forcing **abstraction** by separating an object's:
    - **External view** (*i.e.* behavior):
      - Services an object provides / Interaction with System.
    - **Internal view** (*i.e.* state):
      - Details of class variables and methods.
  - Protecting **integrity** of object's data & avoid unwanted access:
    - Promote an object's **self-governance**.
    - Associate `private` visibility to state vars (referenced only in class).
    - No direct access (using " . " op.) is allowed.
    - Accessing/Modifying state vars require:
      - Additional `public` interface methods (references in/out of class):
        - **Accessor Methods.**
        - **Mutator Methods.**

Example 3: Errors Due To Encapsulation

```
public class Point {
    private int x; // Attribute variable
    private int y; // Attribute variable
    // Class Constructor Method:
    public Point(int init_x, int init_y) {
        x = init_x; y = init_y;
    }
}
```

```
public class PointClient {
  public static void main(String args[]) {
    Point p = new Point(5, 8);
    System.out.println("x = " + p.x + "y = " + p.y);
    p.x = 7; p.y = 2;
  }
}
```

```
PointClient.java: 4: x has private access in Point
System.out.println("x = " + p.x + "y = " + p.y);
                                    ^
```

Accessor / Getter Methods

- Interface methods with **`public`** visibility and without **`static`** keyword:
  - Used to request the permission to **access**/**read**/**get** an object's internal state variables.
  - Typically implement one accessor method per state variable.
  - Such methods will be applicable to every object instance of a certain class.
- **Header Syntax:** `public <type> <m_name>()`
- **Example:**

```
public class Point {
    private int x; // Attribute variable
    private int y; // Attribute variable
    // Class Constructor Method:
    public Point(int init_x, int init_y) {
       x = init_x; y = init_y;
    }
    // Accessor Methods for x and y:
    public int getX() { return x; }
    public int getY() { return y; }
}
```

Mutator / Setter Methods

- Interface methods with **`public`** visibility and without **`static`** keyword:
  - Used to request the permission to **modify**/**set** an object's internal state variables.
  - Typically implement one mutator method per state variable.
  - Such methods will be applicable to every object instance of a certain class.
- **Header Syntax:** `public void <m_name>(<v_type> <v_name>)`
- **Example:**

```
public class Point {
    private int x; private int y;           // Attr. variables
    public Point(int init_x, int init_y) { // Constructor
       x = init_x; y = init_y;
    }
    // Accessor Methods for x and y:
    public int getX() { return x; }
    public int getY() { return y; }
    // Mutator Methods for x and y:
    public void setX(int x_new) { x = x_new; }
    public void setY(int y_new) { y = y_new; }
}
```

Example 3 – Fixed: No Encapsulation Errors

```
public class Point {
    private int x; private int y;           // Attribute variables
    public Point(int init_x, int init_y) { // Constructor
       x = init_x; y = init_y;
    }
    // Accessor Methods for x and y:
    public int getX() { return x; }
    public int getY() { return y; }
    // Mutator Methods for x and y:
    public void setX(int x_new) { x = x_new; }
    public void setY(int y_new) { y = y_new; }
}
```

```
public class PointClient {
  public static void main(String args[]) {
    Point p = new Point(5, 8);
    System.out.println(p.getX() + ", " + p.getY());
    p.setX(7); p.setY(2);
    System.out.println(p.getX() + ", " + p.getY());
}} // due to space limitation.
```

Output? 5, 8

Output? 7, 2

Including More Custom-Built Methods

- Similar to accessor/mutator methods, other methods can be added.
  - These methods extend the interface and need to be public.
  - They will also be applicable for every object instance of a class.
- **Header Syntax:** `public <type> <m_name>(<params>)`
- **Example:**
  - Write a method `distFO()` to compute the distance of a point from the origin.

```
public class Point {
    private int x; private int y;
    public Point(int init_x, int init_y) {
       x = init_x; y = init_y;
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public void setX(int x_new) { x = x_new; }
    public void setY(int y_new) { y = y_new; }
    public double distFO() { return Math.sqrt(x * x + y * y); }
}
```

Exercise 1.1: Distance Between Two Points

Augment the `Point` class with a method called `distance()` that takes another point, say `p`, as a parameter and computes the distance between the current point and `p`.

Solution:

```
public class Point {
    private int x; private int y;
    public Point(int init_x, int init_y) {
       x = init_x; y = init_y;
    }
    public int getX() { return x; }
    public int getY() { return y; }
    public void setX(int x_new) { x = x_new; }
    public void setY(int y_new) { y = y_new; }
    public double distFO() { return Math.sqrt(x * x + y * y); }
    public double distance(Point p) {
       return Math.sqrt(Math.pow(x - p.getX(), 2) +
                        Math.pow(y - p.getY(), 2) );
    }
}
```

Exercise 1.2: Distance Between Two Points

Now, modify the `PointClient.java` program to use the earlier implemented `distance()` method.

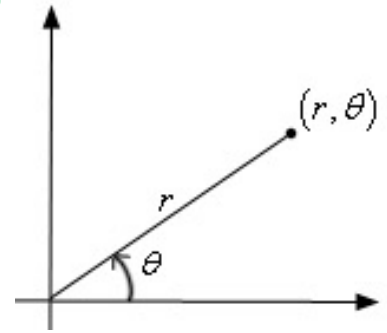
Solution:

```
public class PointClient {
  public static void main(String args[]) {
    Point p1 = new Point(5, 8);
    Point p2 = new Point(7, 3);
    double d = p1.distance(p2);
    System.out.println("P1(" + p1.getX() + ", " + p1.getY() + ")");
    System.out.println("P2(" + p2.getX() + ", " + p2.getY() + ")");
    System.out.println("Distance: " + d);
  }
}
```

Exercise 2: Polar Coordinates of a Point

Augment the `Point` class to include two additional attributes `r` and `theta` (degrees) representing the polar coordinates of a `Point`. Then write a method called `polar()` that computes and returns these parameters in the form of a `String` formatted as follows: `(r, theta)` both formatted to two digits after the decimal point.

```
import java.text.DecimalFormat;
public class Point {
    private int x; private int y; private double r; private double theta;
    public Point(int init_x, int init_y) { x = init_x; y = init_y;}
    public int getX() { return x; }
    public int getY() { return y; }
    public void setX(int x_new) { x = x_new; }
    public void setY(int y_new) { y = y_new; }
    public double distFO() { return Math.sqrt(x * x + y * y); }
    public double distance(Point p) {
       return Math.sqrt(Math.pow(x - p.getX(), 2) + Math.pow(y - p.getY(), 2) );
    }
    public String polar() {
       DecimalFormat fmt = new DecimalFormat("0.##");
       r = distFO(); theta = Math.atan((double) y / x);
       return "(" + fmt.format(r) + ", " + fmt.format(theta) + ")";
    }
}
```



Exercise 3: Point Geometric Translation

Augment the `Point` class with a method called `translate()` that moves a point to a new location within the 2D plane, vertically by `dx` and horizontally by `dy`. For more correctness, x and y must become floating points from now on.

```
import java.text.DecimalFormat;
public class Point {
    private double x; private double y; private double r; private double theta;
    public Point(double init_x, double init_y) { x = init_x; y = init_y;}
    public double getX() { return x; }
    public double getY() { return y; }
    public void setX(double x_new) { x = x_new; }
    public void setY(double y_new) { y = y_new; }
    public double distFO() { return Math.sqrt(x * x + y * y); }
    public double distance(Point p) {
       return Math.sqrt(Math.pow(x - p.getX(), 2) + Math.pow(y - p.getY(), 2) );
    }
    public String polar() {
       DecimalFormat fmt = new DecimalFormat("0.##");
       r = distFO(); theta = Math.atan((double) y / x);
       return "(" + fmt.format(r) + ", " + fmt.format(theta) + ")";
    }
    public void translate(double dx, double dy) {setX(x + dx); setY(y + dy)};
}
```

Print Representation Of An Object

```
Point c = new Point(3,4);
System.out.println("c is " +
c);

Output: c is Point@28a418fc
```

- Quite **uninformative** default print representation:
  - JAVA knows not how to print an object.
- **Desired output** `c is (3, 4)` can be achieved by using (also too cumbersome):

```
System.out.println("(" + c.getX() + ", " + c.getY() + ")");
```

- **Requirement:** print useful information about an object using `System.out`
- Define a new **`toString()` method** for a class:
  - JAVA calls this method when attempting to print object using `System.out` methods.
- **Example Usecase:**

```
System.out.println(c);
```

Output: `(3, 4)`

toString() Method

- Tells JAVA how to convert an object into a `String`.
- Included into all classes **even if not explicitly defined by programmer**:
- **Default Outcome:** `<c_name>@<o_addr>` (address is Hex, *i.e.* base 16).
- **Header Syntax:** `public String toString()`
- Programmer can **override** this method by customly defining it in the class.
- **Example:** for appropriately printing a Point object

```
public String toString() {
        return "(" + x + ", " + y + ")";
}
System.out.print(c);  // from main() → (3, 4)
```

Complete **Point** Class

```
import java.text.DecimalFormat;
public class Point {
    private double x; private double y; private double r; private double theta;
    public Point(double init_x, double init_y) { x = init_x; y = init_y;}
    public double getX() { return x; }
    public double getY() { return y; }
    public void setX(double x_new) { x = x_new; }
    public void setY(double y_new) { y = y_new; }
    public double distFO() { return Math.sqrt(x * x + y * y); }
    public double distance(Point p) {
       return Math.sqrt(Math.pow(x - p.getX(), 2) + Math.pow(y - p.getY(), 2) );
    }
    public String polar() {
       DecimalFormat fmt = new DecimalFormat("0.##");
       r = distFO(); theta = Math.atan((double) y / x);
       return "(" + fmt.format(r) + ", " + fmt.format(theta) + ")";
    }
    public void translate(double dx, double dy) {setX(x + dx); setY(y + dy)};
    public String toString() { return "(" + x + ", " + y + ")"; }
}
```

Method Overloading

- It is possible to define multiple versions of a method having:
  - The same name.
  - Different header signatures (*i.e.* different return type, different parameters…):
    - The order of parameters is important → **different order implies a different signature!!**
  - The signature of each overloading method must be unique.
- Depending on the call, JAVA will match and execute the rigth version.
- **Example:** Defining multiple constructor methods for `Point` class.

```
public Point() { x = 0; y = 0; }
public Point(double x_init, y_init) {
      x = x_init; y = y_init
}
```

Another Example: Method Overloading

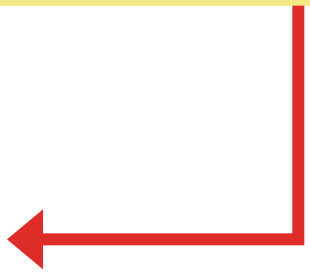
- Compiler determines which method is being invoked by analyzing the parameters

```
float tryMe(int x)
{
    return x + .375;
}

float tryMe(int x, float y)
{
    return x*y;
}
```

Invocation

```
result = tryMe(25, 4.32)
```



Another Example: Method Overloading

- The `println()` method is overloaded:

```
println (String s)
println (int i)
println (double d)
...
```

- The following lines invoke different versions of the `println()` method:

```
System.out.println ("The total is:");
System.out.println (total);
```

Practice Exercise: `RationalNumbers` Class

- **Definition:**
  - A number represented as a ratio of two integers:
    - Numerator, `N`.
    - Denominator, `D`.
- **Required Operations:**
  - Addition, subtraction, multiplication, division, inversion, check equality, inequality …
  - Printing a rational number in a fractional format (*i.e.* `N/D`).
- **Tasks:**
  - Write a JAVA class to define the `RationalNumber` type.
  - Write a Driver/Client program to:
    - Instantiate objects of this type.
    - Test the interaction among actual `RationalNumber` objects.

Practice Exercise: `Die` Class

- A **die** (singular of dice) is a six-sided squared box:
  - Its state can be defined in terms of the shown face (*i.e.* integer on top side of the die).
  - Its primary behavior is that it can be rolled.
- **Task:**
  - Write a JAVA class called `Die` having:
    - One class variable `faceValue` to represent a die's face:
      - This is initialized as a random integer between `1` and `6` (both inclusive).
    - Appropriate accessor and mutator methods for that state variable.
    - A method called `rolls()` that mimics the rolling of the die (*i.e.* `faceValue` varies again randomly).
    - A `toString()` method that informatively returns a die's faceValue as a String

The `this` Keyword

- Refers to a variable that **points to the object** on which a method is called:
  - In other words it **implicitly refers to the instance variables** inside that given object.
- **Syntax:**
  - **Refer to an attribute / field:**

```
this.<v_name>
```

  - **Invoke a method:**

```
this.<m_name>(<params>);
```

  - **Constructor calling an overloaded one of its versions:**

```
this(<params>);
```

Variable Shadowing - Revisited

- This occurs whenever **multiple variables within the same scope have the same name**.
  - Normally this is illegal, except when one variable is a field.
- **Example:**

```
public class Point {
    private double x; private double y;
    // The below constructor is legal but bogous
    public Point (double x, double y) {
        x = x; y = y;
    }
```

local params

- Assigning local variables their own values.
- Field values will remain 0.

  - In most of the class, `x` and `y` refer to the fields.
  - In `setLocation`, `x` and `y` refer to the method's parameters.

Variable Shadowing - Revisited

- Fix the variable shadowing problem using `this` keyword:
- **Example:**

```
public class Point {
    private double x; private double y;
    public Point (double x, double y) {
        this.x = x; this.y = y;
    }
```

Reads as:
the **instance field/variable `x` / `y`** is assigned the **value of the local method parameter `x` / `y`**.

Calling Another Constructor

- **Example:** Consider the below JAVA class Snippet:

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

        public Point() { this(0, 0); }     // calls (x, y) constructor
        public Point(double x, double y) {
            this.x = x;
            this.y = y;
        }
        ...
  }
```

  - **Avoids redundancy** between constructors
  - **Only a constructor** (not a method) **can call another constructor** this way.

Revised Complete Point Class

```
import java.text.DecimalFormat;
public class Point {
    private double x; private double y; private double r; private double theta;
    public Point(double x, double y) { this.x = x; this.y = y;}
    public double getX() { return x; }
    public double getY() { return y; }
    public void setX(double x) { this.x = x; }
    public void setY(double y) { this.y = y; }
    public double distFO() { return Math.sqrt(x * x + y * y); }
    public double distance(Point p) {
       return Math.sqrt(Math.pow(x - p.getX(), 2) + Math.pow(y - p.getY(), 2) );
    }
    public String polar() {
       DecimalFormat fmt = new DecimalFormat("0.##");
       r = distFO(); theta = Math.atan((double) y / x);
       return "(" + fmt.format(r) + ", " + fmt.format(theta) + ")";
    }
    public void translate(double dx, double dy) {setX(x + dx); setY(y + dy)};
    public String toString() { return "(" + x + ", " + y + ")"; }
}
```

Arrays of Objects

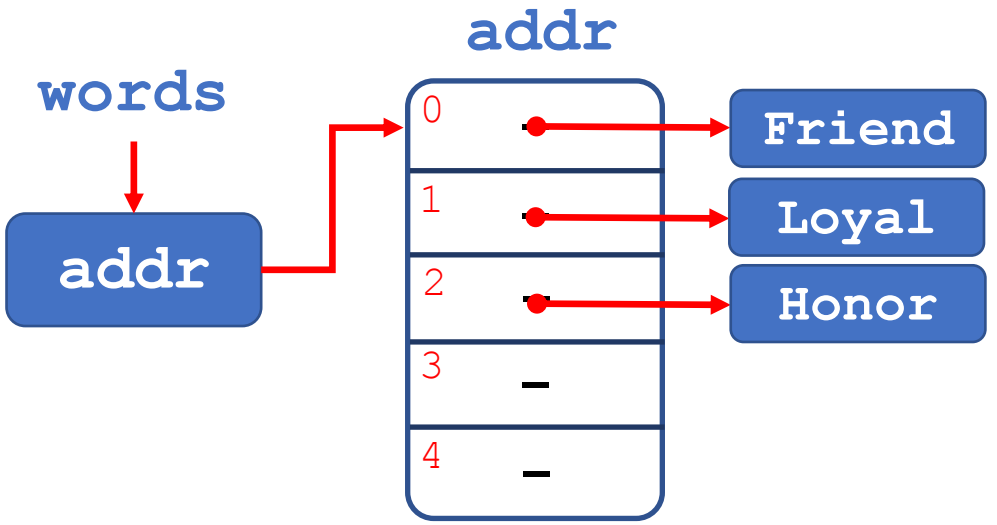
- The elements of an array can be references to non-scalar objects.
- **Example:**

```
String[] words = new String[5];
```

Reserves space to store five references to `String` objects

- Initially, an array of objects holds `null` (–) references.
- Each object stored in an array:
  - must be instantiated separately.
- **Example:**

```
words[0] = "Friend";
words[1] = "Loyal";
words[2] = "Honor";
```



Example: Array Of `Point` Objects

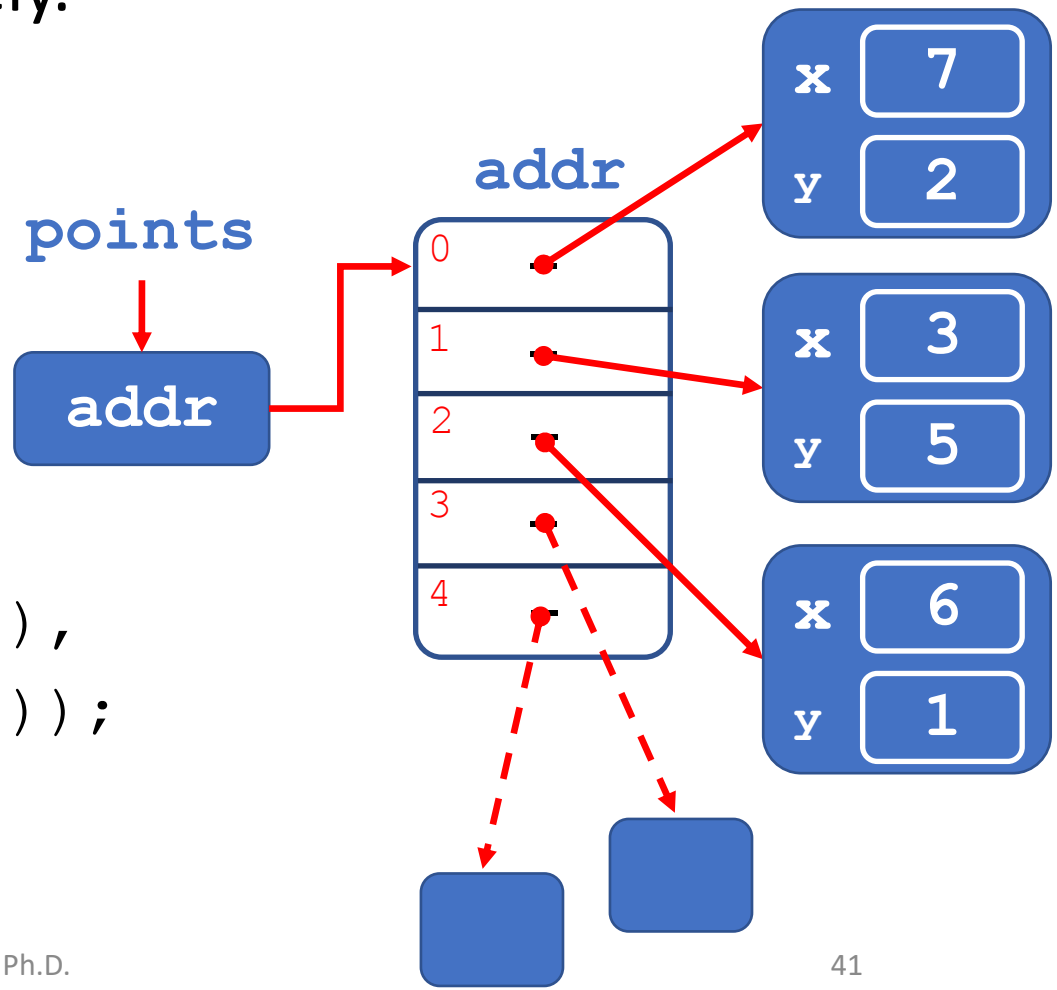
- Such an array is, first, initialized to hold `null` (–) references.

```
Point[] points = new Point[5];
```

Reserves space to store five references to `Point` objects

- Then each `Point` object is initialized separately.
- **Example:**

```
Scanner k = new Scanner(System.in);
for (int i = 0; i < 5; i++) {
  System.out.print("Enter x" + i +
                   " and y" + i +
                   ": ");
  points[i]= new point(k.nextDouble(),
                       k.nextDouble());
}
```



`null` Usecases

- **Store `null` in a variable or an array element:**

```
String s = null;
words[2] = null;
```

- **Print a `null` reference:**

```
System.out.println(s);          // Output: null
```

- **Check whether a variable or array element is `null`:**

```
if (words[2] == null) { ... }
```

- **Pass `null` as a parameter to a method:**

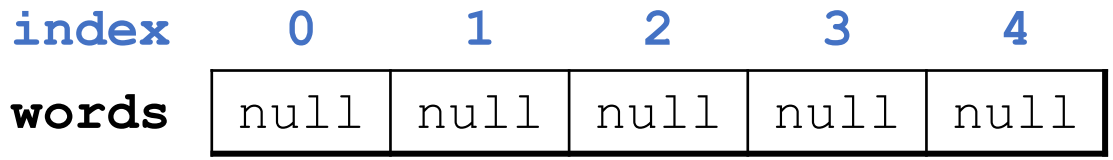
```
System.out.println(null);       // Output: null
```

- **Return `null` as a method's result (often to indicate failure):**

```
return null;
```

NullPointerException

- **Dereferencing**:
  - Pursual of memory address in a reference to location in memory where actual object is stored.
  - It is **not allowed to dereference `null`** (causes an exception).
  - `null` is **not an object**, so it **has no methods or data**.
- **Example:**




```
String[] words = new String[5];
System.out.println("word is: " + words[0]);
words[0] = words[0].toUpperCase();
```

Output:

```
word is: null
Exception in thread "main" java.lang.NullPointerException
        at Example.main(Example.java:8)
```

Look Before Leaping

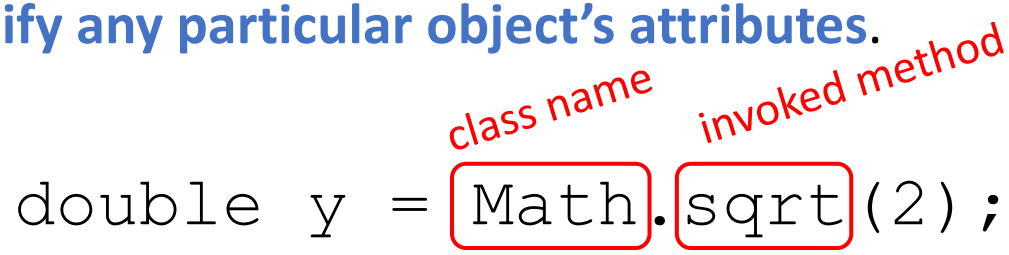
- **Check for `null` before invoking an object's methods:**

```
String[] words = new String[5];
words[0] = "hello";
words[2] = "goodbye";    // words[1], [3], [4] are null

for (int i = 0; i < words.length; i++) {
    if (words[i] != null) {
        words[i] = words[i].toUpperCase();
    }
}
```

Class `static` Members

- In a class both variables and method can be declared as `static`.
- `static` methods are those invoked using the class's name.
  - Are stored in a class not in an object and has no implicit parameter `this`.
  - **Cannot access/modify any particular object's attributes**.
- **Example:**



- Opposite to instance variables, `static` variables in a class:
  - Are referred to as **class variables**.
  - Memory space for such variables is **reserved upon the class's first referencing**.
  - Are **shared among all object instances** of the class (one copy for all objects).
  - Once changed by one of the objects, the change is reflected to all other objects.
- **Example:**

```
private static int numPoints;
```

Example: `static` Members

```
public class Slogan {
    private String phrase; static int count = 0;
    public Slogan(String phrase) { this.phrase = phrase; }
    public static getCount() { return count; }
    public String toString() { return phrase;}
}
public class SloganClient {
    public static void main(String[] args) {
        Slogan s1 = new Slogan("Live free or die!");
        Slogan s2 = new Slogan("Talk is cheap.");
        Slogan s3 = new Slogan("Think before you ink.");
        System.out.println(s1 + "\n" + s2 + "\n" + s3);
        System.out.println("# of slogans: " + Slogan.getCount());
    }
}
```

Remarks on `static` Members

1. `static` methods are allowed to use `static` variables.
2. `static` methods are not allowed to use non-`static` variables.
3. Non-`static` methods are allowed to use `static` variables.
4. Non-`static` methods are allowed to use non-`static` variables.
5. `static` variables are not referenced in the context of an object instance.
6. Non-`static` variables are referenced in the context of an object instance.

Practice Exercise: `BankAccount` Class

- **Tasks:**
  - Create a class called `BankAccount` according to the below specifications:
    - Any object of type BankAccount will have the following attributes:
      - A unique integer ID.
      - The name of the account's owner.
      - The account's type (*i.e.* checking / savings account)
      - The account's balance.
      - The interest rate (in case of a savings account.
    - The class must define:
      - Appropriate accessor/mutator methods for each attribute.
      - Methods that allow to withdraw, deposit, compute and accumulate interest amounts.
      - A method to print out useful information about an account.
    - In addition, one must, at any time, be able to retrieve the number of accounts created.
  - Create an appropriate client program to test the above-elaborated class.

Classes As Modules

- A module is a **partial** program (**reusable** software) stored in a class:
  - It **does not have a `main()` method**.
  - It **cannot be directly executed**.
  - Is meant to be **utilized by other client classes**.
  - It is composed of `static` members.
  - **Examples of Module Classes:** `Math`, `Arrays`, `System`
- **Syntax:** **`<c_name>.<m_name>(parameters);`**
- **Example:**

```
int num = -24;
int absNum = Math.abs(num);
```

Example F.1: A Custom-Built Module **Factors**

```
// This module contains useful methods related to factors and primes
public class Factors {
    // Returns the number of factors of the given integer.
    public static int countFactors(int number) {
        int count = 0;
        for (int i = 1; i <= number; i++)
            if (number % i == 0) count++; // i is a factor of number
        return count;
    }
    // Returns true if the given number is prime.
    public static boolean isPrime(int number) {
        return countFactors(number) == 2;
    }
}
```

Example F.2: Using the `Factors` Module

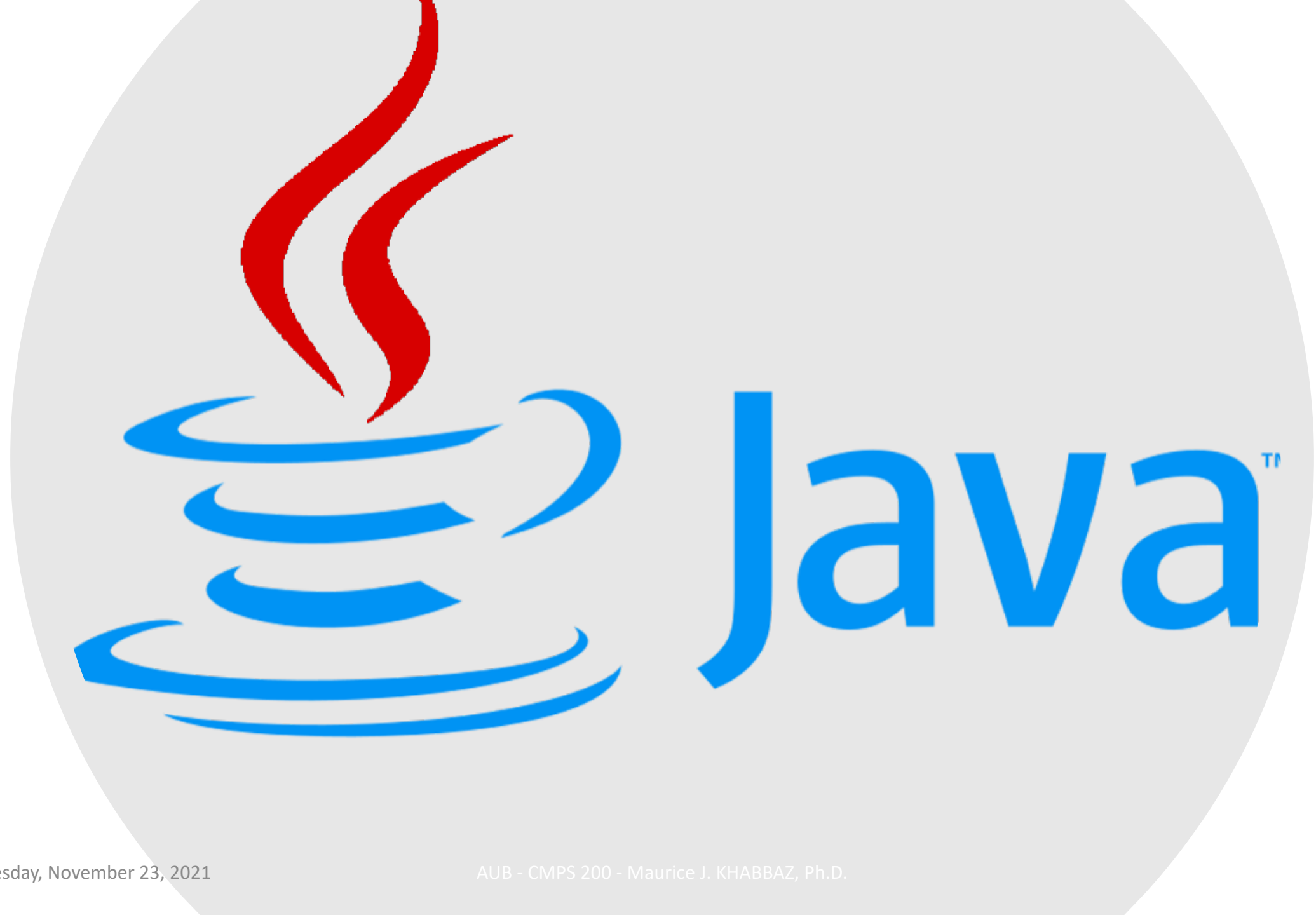
```
// This program sees whether some interesting numbers are prime.
public class FirstPrimes {
    public static void main(String[] args) {
        int[] nums = {1234517, 859501, 53, 142};
        for (int i = 0; i < nums.length; i++)
            if (Factors.isPrime(nums[i]))
                System.out.println(nums[i] + " is prime");
    }
}
// This program prints all prime numbers up to a given maximum.
public class SecondPrimes {
    public static void main(String[] args) {
        Scanner k = new Scanner(System.in);
        System.out.print("Max number? "); int max = k.nextInt();
        for (int i = 2; i <= max; i++)
            if (Factors.isPrime(i)) System.out.print(i + " ");
           System.out.println();
    }
}
```

An Abused Example of Modules in Java Libraries

```
// JAVA's built in Math class is a module
public class Math {
    public static final double PI = 3.14159265358979323846;
    ...
    public static int abs(int a) {
        if (a >= 0) return a;
        else return -a;
    }
    public static double toDegrees(double radians) {
        return radians * 180 / PI;
    }
}
```

The Power Of OOP

- **Bundle together objects** that share:
  - Common attributes.
  - Procedures that operate on those attributes.
- Use **abstraction** to make a distinction between:
  - Object implementation.
  - Object usage.
- Build **layers** of object abstractions that inherit behaviors from other classes/objects.
- Create **custom-built object classes** on top of JAVA's basic classes.


Java