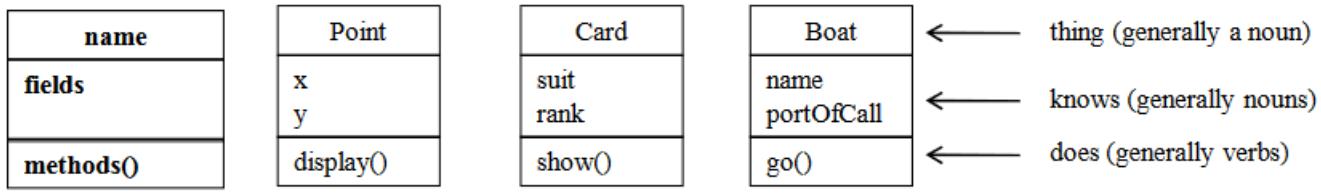
WEEK TEN

Acknowledgements: Slides created based off material provided by Dr. Travis Doom

OBJECT-ORIENTED PROGRAMMING (OOP)

- In contrast to procedural programming (using methods), OOP uses objects to decompose complexity
- An object often represents a real-world "thing"
- It is made up of:
  - *Fields:* instance variables; things the object has or knows
  - *Methods:* things the object can do



UML (Unified modeling language) diagrams

CREATING/USING AN OBJECT

- Generally, create a second file to create the new class in
  - Therefore, we have a class that creates/defines the object
  - And one with our main method to test the object
- Remember, object creation
  - `Scanner scnr = new Scanner(System.in);`
  - We will be able to do this for our own objects too
- Method access through an object
  - `scnr.next();`
- Field access through an object
  - `int[] nums = new int[5];`
  - `nums.length;`

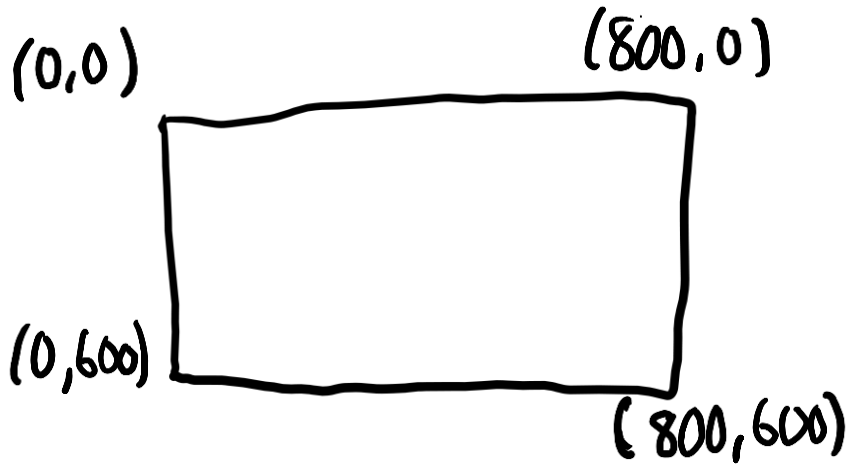
NULL

- Default value for any object reference variable before it is initialized
  - `Scanner scnr; // will be null until assigned a value`
- Keyword
- Can be stored in a reference variable
  - `String name = null;`
- Means that the variable currently refers to no existing object
- Cannot be assigned to primitive datatypes (literal vs. reference)
- Good practice when writing methods to ensure that object reference parameters are not null

CLASS VARIABLES

- Local variables
  - Declared inside a class method
  - Scope is limited to that method
  - Cannot be used by other methods
- Instance variables
  - Declared inside the class but not in a method
  - Have scope of the entire class
  - Can be used in any class method
  - Can be accessed via the object (think .length for arrays)
  - Often, we want to limit this access

PROTECTING OBJECT FIELDS



- Imagine a class representing a pixel on your computer screen
  - The class has int fields for the x and y position
  - If the x and y fields are public, the user can access them through the object:

```
Pixel p = new Pixel();
int x = p.x;
```

  - What happens if the user tries this?

```
p.x = -5;
```

  - We can declare our fields as `private` to prevent malicious intent or user mistakes
  - This restricts access to within the class itself (no longer can be accessed outside of the class)
  - So how do we allow correct access to these fields?

MUTATORS/SET METHODS

- Methods whose only job is to provide “checked” access to variables
- They *mutate* or change the state of the object

```
public class Pixel {
    private int x = 0;
    private int y = 0;

    public void setX(int newValue) {
            if (newValue < 0) {
                    x = 0;
            } else if (newValue > 800) {
                    x = 800;
            } else {
                    x = newValue; }}}
```

ACCESS MODIFIER COMPARISON

Public field/method

- Can be accessed from within the class it is defined
- Can be accessed from outside the class it is defined in
- Most of the time, we want our methods to be public (unless we don't want someone using our class to call that method)

Private field/method

- Can be accessed from within the class it is defined
- CANNOT be accessed from outside the class it is defined in
- Most of the time, we want our fields to be private
- We want the user to utilize our methods in order to access/modify the fields

ACCESSORS/GET METHODS

- Methods whose only purpose is to return the value stored in a private class field
- They *access* or *get* the value of the variable

```
public class Pixel {
    private int x = 0;
    private int y = 0;

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

ENCAPSULATION

- The idea of protecting our class data as much as possible
- Important rule for OOP style
  - Mark all instance variables as private
  - Create public getter/setter methods
- Even if you don't need extra checking now,
  - Good habit to get into
  - Helps limit changes you may have to make later

THIS

- Java keyword
- Used to refer to the current object
- Can be used to reference fields (or methods) of the class
- Necessary if we want to have instance (class) and local (method) variables of the same name

```
public class Pixel {
	private int x = 0;
	public void setX(int x) {
		x = x; // won't change x
	}
}
```

```
public class Pixel {
	private int x = 0;
	public void setX(int x) {
		this.x = x;
	}
}
```

CONSTRUCTORS

- Method that is automatically called when the object is created
- Shares the same name as the class (including capitalization)
- Used to initialize anything we may need once the object is created
- Have no return type

```
public class Pixel {
	private int x = 0;
	private int y = 0;
	public Pixel(int x, int y) {
		this.x = x;
		this.setY(y);
}}
```

DEFAULT & MULTIPLE CONSTRUCTORS

- Default constructor: constructor that takes in no arguments (no-arg)
  - Will be run behind the scenes by the compiler if you do not explicitly define one
  - If you explicitly define a constructor, the compiler does not do this
- Constructor overloading (multiple constructors)
  - Overloading refers to having multiple methods with the same name but different parameters
  - You can have more than one constructor if they take different parameters
  - EX: you can write your own no-arg constructor *and* a constructor that takes two parameters

STATIC KEYWORD

Static Method

- Can be executed without a class instance (object)
- Not dependent on the state of an object
- CAN'T use instance variables (directly)
- Can use static and local variables
- Can't use non-static methods of the class

```
int num = Math.abs(-5);
```

Non-Static Method

- Must be called via a class instance (object)
- Dependent on the state of the object
- Can use instance and local variables
- Can call non-static methods of the class

```
ArrayList<Integer> nums = new
ArrayList<>();

nums.add(5);
```

STATIC KEYWORD

Static Variable

- Variable has one memory location shared by all instances of the class (objects)
- One value per class, not per object
- Can be accessed without an object

Non-Static Variable

- Variable has the same number of memory locations as there are instances of the class
- One value per object, not class
- Must be accessed via an object

STATIC VARIABLE EXAMPLE

```
public class Pixel {
        public static int numPixels = 0;
        public int x;
        public int y;
        …
}
```

```
Pixel p1 = new Pixel();
p1.x = 5;
p1.y = 5;
Pixel.numPixels++; // numPixels = 1
Pixel p2 = new Pixel();
p2.x = 35;
p2.y = 74;
p2.numPixels++; // numPixels = 2
```

FINAL KEYWORD

- Variables
  - Indicates a *constant*
  - The value cannot be changed after it is set
  - `final public double PI = 3.14;`
- Methods
  - Indicates the method cannot be overridden
- Classes
  - Indicates the class cannot be extended

ACTIVITY

- Write a class for a Wright State student
- It should include the following fields:
  - Student's name
  - Student's age
  - Student's UID
  - Student's GPA
- Write two constructors:
  - One that takes in all of the above as parameters
  - A no-arg constructor that gives all of the above default values
- Write some methods:
  - A setter for the GPA that ensures the input is within a valid GPA range
  - A setter that ensures the student's age is valid
  - A getter for their name

ACTIVITY CONTINUED

- In the main method of a separate class (main class):
  - Create an object for the class you created

- Add a field to the class to represent whether the student is an alumni or not
- Write a method that checks if the student is able to graduate
  - If they have a GPA of at least 2.5, they may graduate
  - If they successfully graduate, update their alumni status
  - Have the method return a boolean to indicate whether they graduated successfully or not