

# Lecture 11: Classes

AP Computer Science Principles

# Modularity

# Modularity

- **modularity:** Writing code in smaller, more manageable components or modules. Then combining the modules into a cohesive system.
  - Modularity with methods. Break complex code into smaller tasks and organize it using methods. (This was discussed in the Methods lecture)
  - Modularity with Objects. Break a large program into objects. A program run is the interaction of these objects. For example, a game can have Player objects, Enemy objects, Weapon objects etc.. The interaction of these objects make up the game.

# Modularity with Methods

# Modularity with Methods

```
int centerX, centerY, xspeed, yspeed;  
void setup() {  
    size(800, 600);  
    centerX = 400; centerY = 300;  
    xspeed = 5; yspeed = 4;  
}  
void draw() {  
    background(255);  
    fill(255, 0, 0);  
    ellipse(centerX, centerY, 80, 80);  
    centerX = centerX + xspeed;  
    centerY = centerY + yspeed;  
    if (centerX > width || centerX < 0)  
        xspeed = -xspeed;  
    if (centerY > height || centerY < 0)  
        yspeed = -yspeed;  
}
```

How do we make this modular?

Break into smaller tasks and organize code using functions.

# Modularity with Methods

```
int centerX, centerY, xspeed,  
yspeed;  
void setup() {...}  
void draw() {  
    background(255);  
  
    drawBall();  
    moveBall();  
    bounceBall();  
}  
void drawBall() {  
    fill(255,0,0);  
    ellipse(centerX,centerY,80,80);  
}
```

```
void moveBall() {  
    centerX = centerX + xspeed;  
    centerY = centerY + yspeed;  
}  
  
void bounceBall(){  
    if(centerX>width || centerX<0)  
        xspeed = -xspeed;  
    if(centerY>height || centerY<0)  
        yspeed = -yspeed;  
}
```

Modularity helps with debugging. For example, if the ball is not bouncing properly, we only need to look at the bounceBall method.

# Modularity with Classes

# Bouncing Ball

```
int centerX, centerY, xspeed, yspeed;  
void setup() {  
    size(800, 600);  
    xspeed=5, yspeed=7;  
}  
void draw() {  
    background(255);  
    fill(255, 0, 0);  
    ellipse(centerX,centerY,80,80);  
    centerX = centerX + xspeed;  
    centerY = centerY + yspeed;  
    if (centerX > width || centerX < 0)  
        xspeed = -xspeed;  
    if (centerY > height || centerY < 0)  
        yspeed = -yspeed;  
}
```

# More Balls

The previous code doesn't scale nicely. What if we want 5 balls? 100 balls? Clearly, we don't want to write following code.

```
int centerX1, centerY1, xspeed1, yspeed1;  
int centerX2, centerY2, xspeed2, yspeed2;  
...  
int centerX100, centerY100, xspeed100, yspeed100;
```

And if we want to move the balls, we have this redundant code:

```
posx1 += velx1;  
posy1 += vely1;  
...  
posx100 += velx100;  
posy100 += vely100;
```

In object-oriented programming(OOP), we want to combine the data of the ball(position, velocity, etc...) and its behavior (move, bounce) into one unit of code called a class.

# More Balls

In the last lecture, we used arrays to do the following.

```
int numBalls = 10;  
int[] x = new int[numBalls];  
int[] y = new int[numBalls];  
int[] xspeed = new int[numBalls];  
int[] diameter = new int[numBalls];
```

Even still, data from different balls are spread across many variables.  
We like each ball to be its own entity, with its own set of properties and behaviors.

# Classes and objects

- **class:** A program entity that represents either:
  1. driver/client/main class: The class that has the setup() and draw() methods. This class drives the program.
  2. **Object class: A template for a new type of objects.**
    - The Car **class** is a template for creating Car **objects**.
- **object:** An entity that combines state(data) and behavior(methods).
  - **object-oriented programming (OOP):** Programs that perform their behavior as interactions between objects. Java is object-oriented.

# Ipod Analogy

Example:

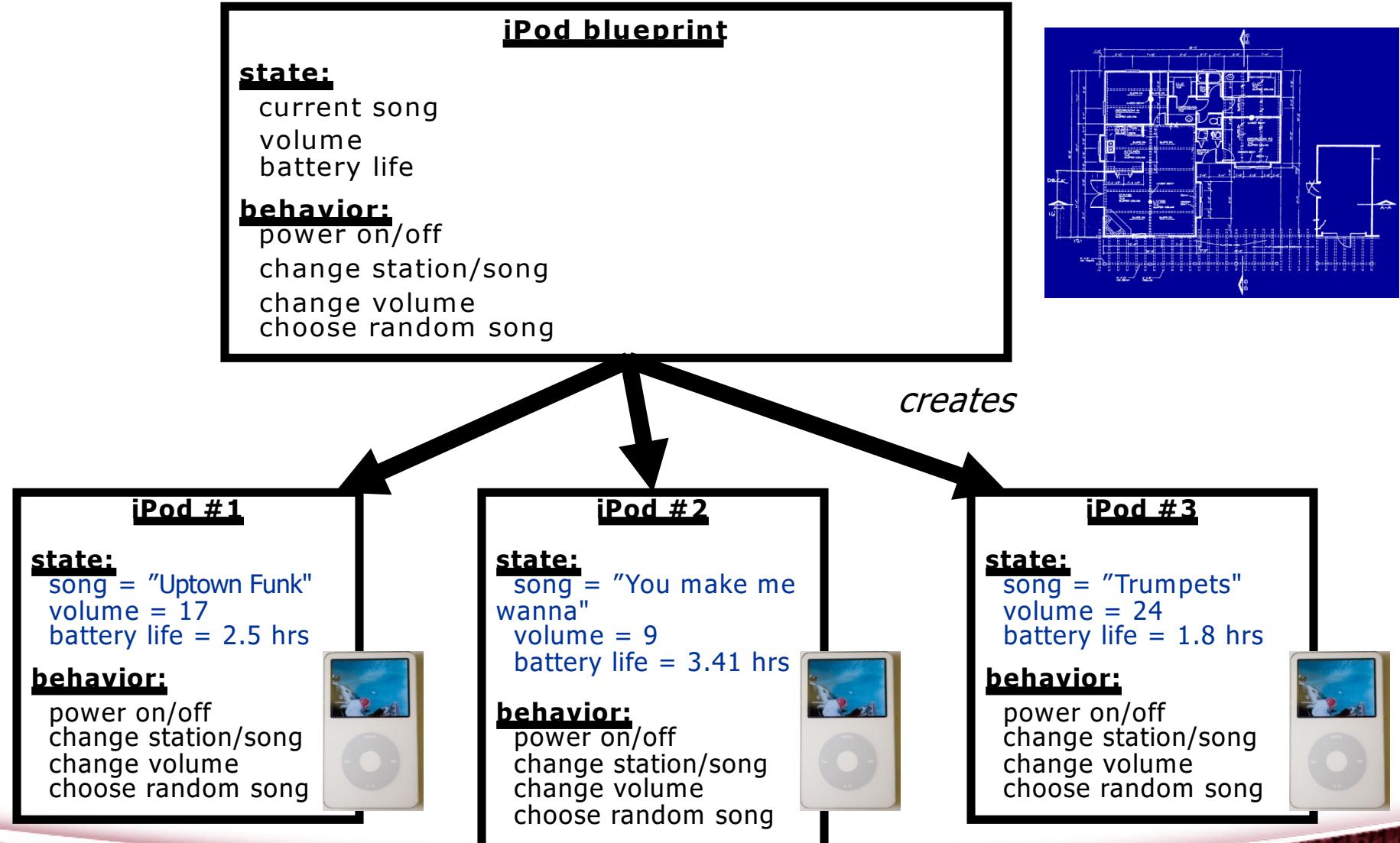
The Ipod **class** provides the template or blueprint for state(data) and behavior(methods) of an ipod **object**.

Its state or data can include the current song, current volume and battery life.

Its behavior or methods can include change song, change volume, turn on/off, etc...

Two different ipod objects can have different states/data. However, their template share the same implementation/code.

# Blueprint analogy



# Object Class

- **Object class: A template for a new type of objects.** It combines state(data) and behavior(methods).

The main file that contains the setup() and draw() methods is called the **main(driver or client) class**. To create an object class, click on  and select "New Tab".

A new file is created in the same folder as the main class. The name of the file is the same as the name of this object class. Note that the name of this folder is the same as the name of the main class.

## Syntax:

```
class ClassName {  
    //body of the Class  
}
```

# Object Class

- **Object class:** A template for a new type of objects. It combines state(data) and behavior(methods).
  - Once an object is created, it becomes a **type** much like String or int are types.
- An object class contains:
  - **Attributes of object:** Field variables that contains the state or data of the object. These are global variables, i.e., variables whose scope is the entire object class.
  - **Constructors:** Special methods that initialize the data variables.
  - **Behaviors:** Methods that give the object its behaviors and functionality. These methods have access to and can modify the field variables.

# Data: Field variables

Data: **Field variables** that contains the state of the object. These are global variables, i.e., variables whose scope is the entire object class.

```
class Ball{  
    //field variables  
    int centerX, centerY;  
    int xspeed, yspeed;  
  
    ...  
    //other code not shown  
}
```

# Constructors

**Constructors:** A class may have one or more constructors to initialize the field variables. If no constructor is given, field variables are initialized to default values. A constructor is **a method** and **has the same name as the class and has no return type.**

```
class Ball{  
    //field variables  
    int centerX, centerY, xspeed, yspeed;  
    //a constructor  
    Ball(){  
        centerX = width/2;  
        centerY = height/2;  
        xspeed = 5; yspeed = 4;  
    }  
    //other code not shown ...  
}
```

# Behaviors

**Behaviors:** A class may have one or more methods that give the object its behavior or functionality. Often these behaviors modify the data of the object.

```
class Ball{  
    int centerX, centerY;  
    int xspeed, yspeed;  
    Ball() {  
        centerX = width/2;  
        centerY = height/2;  
        xspeed = 5; yspeed = 4;  
    }  
    // behavior  
    void move() {  
        centerX += xspeed;  
        centerY += yspeed;  
    }  
}
```

# Full Ball Class

```
class Ball{  
    //field variables  
    int centerX, centerY;  
    int xspeed, yspeed;  
    // constructor  
    Ball() {  
        centerX = width/2;  
        centerY = height/2;  
        xspeed = 5;  
        yspeed = 4;  
    }  
    // behaviors  
    void move() {  
        centerX += xspeed;  
        centerY += yspeed;  
    }  
    void bounce() {  
        if (centerX > width || centerX < 0)  
            xspeed = -xspeed;  
  
        if (centerY > height || centerY < 0)  
            yspeed = -yspeed;  
    }  
    void display() {  
        fill(255, 0, 0);  
        ellipse(centerX, centerY, 80, 80);  
    }  
}
```

# Accessing variables and methods

The variables and methods in the Ball class can now be access outside of it, e.g. in the **main class**. To access a field variable or method of an object, use the **dot notation** with the name of the object.

//inside the main class

```
Ball b; // declare a variable of data type Ball.  
void setup() {  
    size(800,600);  
    b = new Ball(); // calling constructor to initialize  
                    // field variables. This is creating  
                    // a Ball object or instance.  
    print(b.centerX + " " + b.centerY); // 400 300  
    b.move(); // b.centerX=405, b.centerY=304.  
    b.move(); // b.centerX=410, b.centerY=308.  
    b.display(); //display ball at (410,308).  
}  
}
```

# Non-OOP Bouncing Ball

```
int centerX, centerY, xspeed, yspeed;  
void setup() {  
    size(800, 600);  
    centerX = 400; centerY = 300;  
    xspeed = 5; yspeed = 4;  
}  
void draw() {  
    background(255);  
    fill(255, 0, 0);  
    ellipse(centerX, centerY, 80, 80);  
    centerX = centerX + xspeed;  
    centerY = centerY + yspeed;  
    if (centerX > width || centerX < 0)  
        xspeed = -xspeed;  
    if (centerY > height || centerY < 0)  
        yspeed = -yspeed;  
}
```

In object-oriented programming(OOP), we want to combine the data of the ball(position, speed, size, etc...) and its behavior (draw, move, bounce) into one entity or object.

Thus, all of these variables can be removed from the main program and put into another file entirely. This is called **encapsulation**.

# OOP Bouncing Ball

Here's the object-oriented version of the bouncing ball program we wrote earlier in the class.

```
// in the main class (file name "Main.pde" in folder "Main" )  
Ball b; //declare a new variable of type Ball.  
  
void setup(){  
    size(800,600);  
    b = new Ball(); // calling constructor to initialize  
                    // field variables.  
}  
  
void draw(){  
    background(255);  
    b.display();  
    b.move();  
    b.bounce();  
}
```

# Ball Class

```
// filename Ball.pde
class Ball{
    //field variables
    int centerX, centerY;
    int xspeed, yspeed;
    // constructor
    Ball() {
        centerX = width/2;
        centerY = height/2;
        xspeed = 5; yspeed = 4;
    }
    // behaviors
    void move() {
        centerX += xspeed;
        centerY += yspeed;
    }
    void bounce() {
        if (centerX > width || centerX < 0)
            xspeed = -xspeed;

        if (centerY > height || centerY < 0)
            yspeed = -yspeed;
    }
    void display() {
        fill(255, 0, 0);
        ellipse(centerX, centerY, 80, 80);
    }
}
```

# Creating multiple objects

```
Ball b1, b2; //two ball objects.  
void setup() {  
    size(800,600);  
    b1 = new Ball();  
    b2 = new Ball();  
}  
void draw() {  
    background(255);  
    b1.display();  
    b2.display();  
    b1.move();  
    b2.move();  
    b1.bounce();  
    b2.bounce();  
}
```

What does this program do?

The balls overlap! The constructor initializes both balls to have the same speed and initial location.

# Overloading Constructors

A class may have **one or more** constructors to initialize the field variables. A constructor with parameters may be added to give clients the ability to create an Ball object at any location and speed.

```
class Ball{  
    int centerX, centerY, xspeed, yspeed;  
//2 constructors  
Ball(){  
    centerX = width/2; centerY = height/2;  
    xspeed = 5; yspeed = 4;  
}  
Ball(int x, int y, int speedX, int speedY){  
    centerX = x; centerY = y;  
    xspeed = speedX; yspeed = speedY;  
}  
// other code not shown..  
}
```

# Creating multiple objects

```
Ball b1, b2; //two ball objects.  
void setup() {  
    size(800,600);  
    b1 = new Ball();  
    b2 = new Ball(width/4, height/3,-3,5);  
}  
void draw() {  
    background(255);  
    b1.display();  
    b2.display();  
    b1.move();  
    b2.move();  
    b1.bounce();  
    b2.bounce();  
}
```

The balls no longer overlap!

# Lab 1

Recreate the main class and the Ball class from this lecture. DO NOT copy and paste.

Create multiple ball objects and let it move/bounce around the screen.

# Lab 2

Add an additional variable `radius` to the `Ball` class. This will allow you to create balls of different sizes.

Add one more constructor(for a total of 3) to initialize location, speed, and radius to random values.

Create an array of 100 `Ball` objects! Initialize them to random locations, speeds and sizes by using the constructor above. Make them move and bounce around the screen.

```
int numBalls = 100;  
Ball[] balls = new Ball[numBalls];
```

# Lab 2 (continue)

Add an additional boolean variable `visible` to the Ball class with initial value `true` and the method `void checkVisible()` that sets the `visible` variable to `false` if the mouse touches the ball.

Modify the `display()` method to only draw visible balls.

Do you see what this program does?

# Lab 3

The lab will give you a chance to see how multiple classes and their objects interact. **It is critical that everyone gets this entire lab to work. Please ask for help if something doesn't!**

This lab requires you to download from Classroom four .pde files.(ObjectLab.pde, Rectangle.pde, Ellipse.pde and Circle.pde). Please edit and modify these files. DO NOT CREATE YOUR OWN!

Put all of these files into a folder called ObjectLab since ObjectLab.pde is the main/driver class. This lab will walk you through completing the code for each class in this order: Rectangle, Ellipse and then Circle. Please complete each correctly before moving on to the next.

# Lab 3: Rectangle Class

You will first complete the Rectangle object class.

**5 Field Variables:** Integers `x` and `y` for position of a rectangle. Integers `w` and `h` for the width and height, respectively. And a color `c` for its color. `color` is a reserved type in Processing. It can encapsulate both greyscale and RGB colors.

Example:

```
color c1 = color(255) // c1 is white.  
color c2 = color(255, 0, 0); //c2 is red;  
fill(c2);  
rectangle(10, 20, 100, 200); // red fill rectangle.
```

# Lab 3: Rectangle Class

**One default constructor:** This class has one default constructor (without any parameters) to initialize the field variables. Initialize rectangle to locate at the middle of the screen and w = 200 and h = 100 and c to be red (or choose your own values).

## 3 Methods(Behaviors):

void display(): Draw rectangle.

boolean hover(): Returns true if mouse is hovering inside the rectangle.

void chooseColor(): Calls hover(). Depends on the result of that call, alternate between two colors.

**Main Driver Class: Once you completed Rectangle, test it by creating an object, initializing it in setup() and then calls it in draw() to make sure this class hovers properly before moving on to the Ellipse class.**

# Lab 3: Ellipse Class

Once you successfully completed and tested Rectangle, you will complete Ellipse next.

**3 Field Variables:** Integers  $x$  and  $y$  for position of an ellipse. And a color  $c$  for its color. Notice that there are no variables for width and height. In your display() method, simply draw it at a fixed width/height, for example, 200 by 100.

**2 Constructors:** A default constructor(without parameters) to initialize an ellipse(anywhere but the center to avoid overlapping with the rectangle object. An additional constructor that takes three parameters to allow the driver class to initialize the position and color to any value.

# Lab 3: Ellipse Class

## 5 Methods(Behaviors):

void display(): Draw ellipse.

void moveUP, moveDown, moveRight, moveLeft: Move a fixed amount(20 pixels for example) in each direction.

**Main Driver Class: Once you completed Ellipse, you will need to create an object and initialize it in setup() and display it in draw. In addition, you will need to implement the void keyPressed() method to respond to keyboard inputs appropriately on an ellipse object.**

# Lab 3: Circle Class

Once you successfully completed and tested Rectangle and Ellipse, you will complete the final class, Circle. This class will require you to work with arrays.

**5 Field Variables:** Integers `x` and `y` for position of a circle. A float radius. And an ARRAY of colors `c` for three possible colors(e.g. red, green and blue). And an integer `currentColorIndex` for the position of the current color.

**1 Constructor:** A default constructor(without parameters) to initialize a circle appropriately(avoid overlapping with previous objects). Create an array of three colors and initialize them to e.g. red, green and blue.

# Lab 3: Circle Class

## 3 Methods(Behaviors):

void display(): Draw circle filled with the current color.

void checkClick(): Calls distance() to check distance between mouse and current clicked position; if it is in the current circle, call changeColor() to cycle to the next color.

void changeColor(): Update indexcurrentColor to the next color in the color array.

**Main Driver Class: Once you completed Circle, test it by creating an object, initializing it in setup() and then displaying it in draw(). In addition, implement void mouseClicked() by calling checkClick() on the circle.**

# Homework

- 1) Please reread this lecture multiple times! This material is very hard but very important!**
- 2) Complete the labs.

# References

Part of this lecture is taken from the following book.

- 1) Stuart Reges and Marty Stepp. Building Java Programs: A Back to Basics Approach. Pearson Education. 2008.
- 2) Daniel Shiffman. Learning Processing. Morgan Kaufmann.2015.