Classes, Motion, Encapsulation

Week 2
IAT-265
School of Interactive Arts and Technology

Topics

- Class and Objects
- Object-Oriented Programming (OOP)
  - Object and Client Program
  - Encapsulation
  - Data Hiding and Interfacing

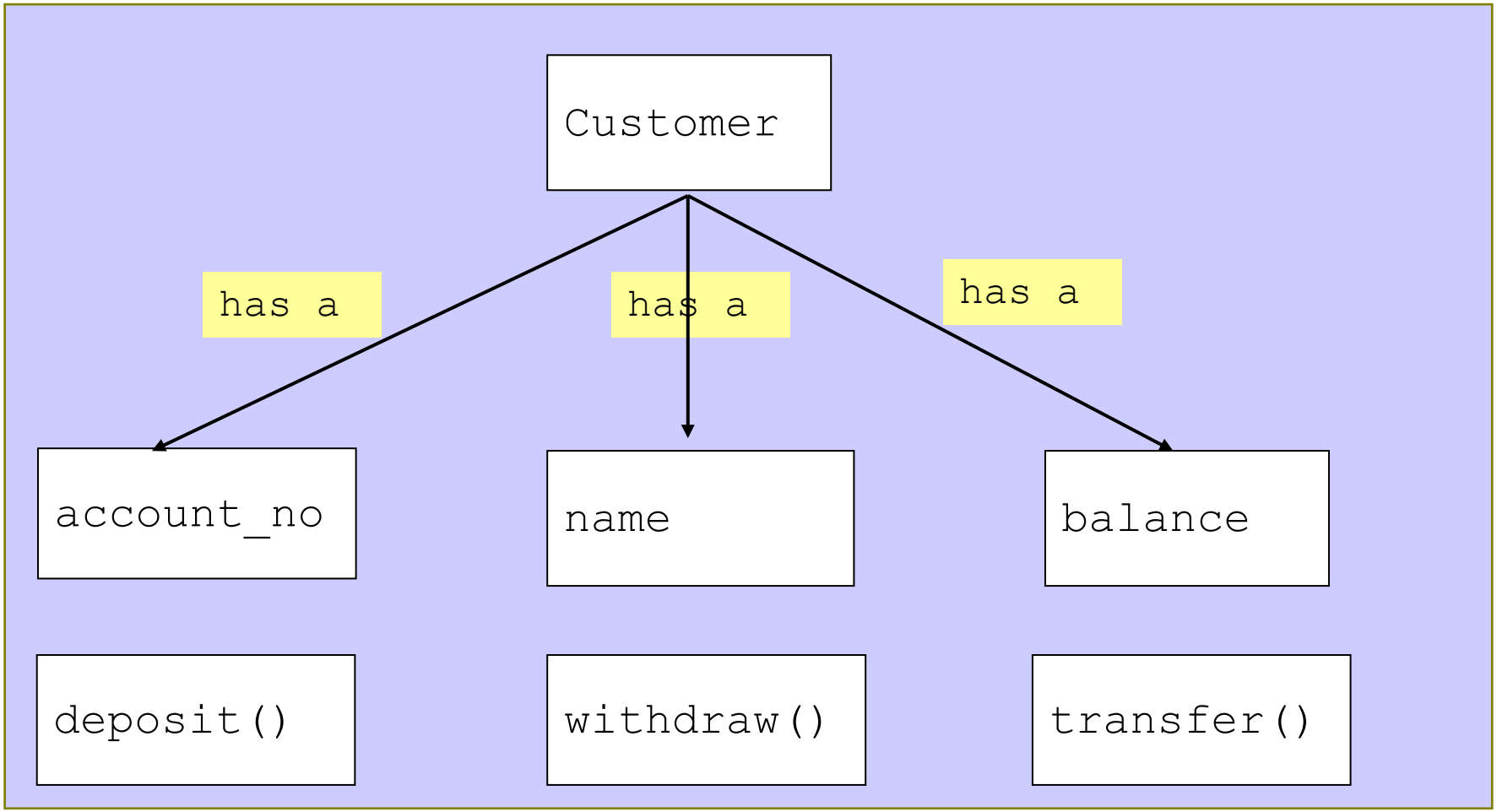
- Timer and Motion
  - Java interface: ActionListener

Recap: Class & Object

Why objects?

- We live in a world full of objects
  - Images, cars, remote controls, televisions, employees, students, bugs, fishes, …
  - Even for some abstract entity like bank account, it's convenient to integrate its data and procedures
- OOP languages have the added capability to encapsulate objects' properties and functions into one container – ***object***
  - The template for creating objects is called ***a class***

A Customer object wraps in all its Properties and Procedures



What an object can offer?

- Attributes – data about What it is
  - Properties/states of an object
    e.g. Bug: sizes, color, location, speed, aliveness …
- Behaviors – procedures regarding What it can do:
  - Capabilities of an object
    e.g. Bug: move, collide, dodge, eat …

Classes and Objects

- **Class**: A program entity that represents a blueprint (aka template) for a type of objects
  - A `Ball` class is a piece of code, serving as a blueprint for creating `Ball` objects
- **Object**: A runtime entity that combines attributes and behaviors
  - A `Ball` object is a specific, visible form at runtime
- One `Ball` class, many `Ball` objects

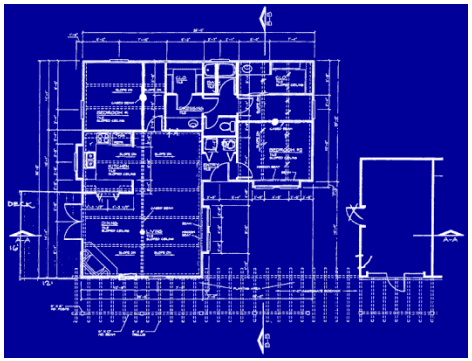
Blueprint Analogy

Courtesy to *Building Java Programs* (Chapter 8). S. Reges|M. Stepp Pearson 2013

iPod blueprint

state:
current song;
volume;
battery life;

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



instantiates

iPod #1

state:
song = "1,000,000 Miles"
volume = 17
battery life = 2.5 hrs

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

iPod #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 #3

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

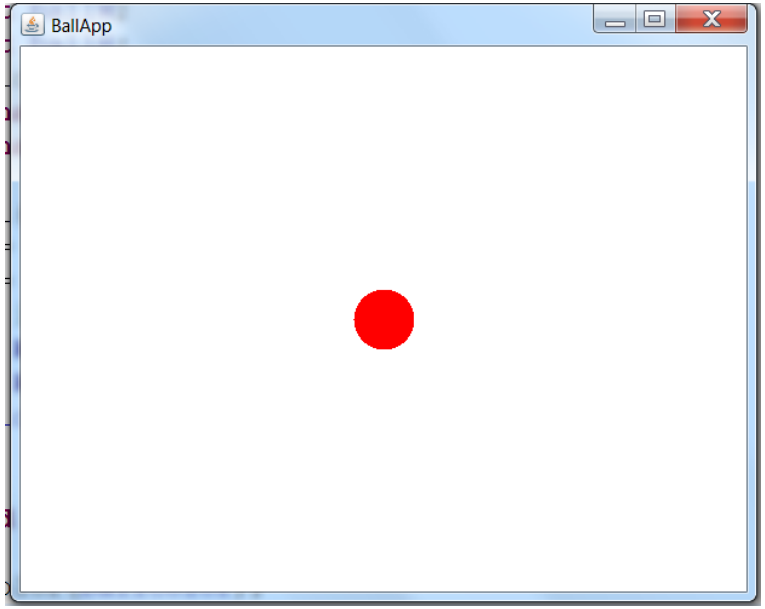
behavior:
power on/off
change station/song
change volume
choose random song

Defining a Class – Describing a type of objects

- A class is a construct that uses *fields* and *methods* to describe one type of objects at some *abstraction* level
- Fields: variables used to store data for every *instance* of the class (i.e. *object*)
  - Each instance has its own values for its fields
- Methods: how you *do things* to or with an instance's data
- Constructors: special methods for creating instances of the class - *instantiation*. It has two features:
  - a) same name as the class; b) no return type

Case Study: Ball

- Let's design the class first:



Ball
- xPos : int - yPos : int - ballW : int - ballH : int - ballColor : Color
+ Ball () + drawBall() : void

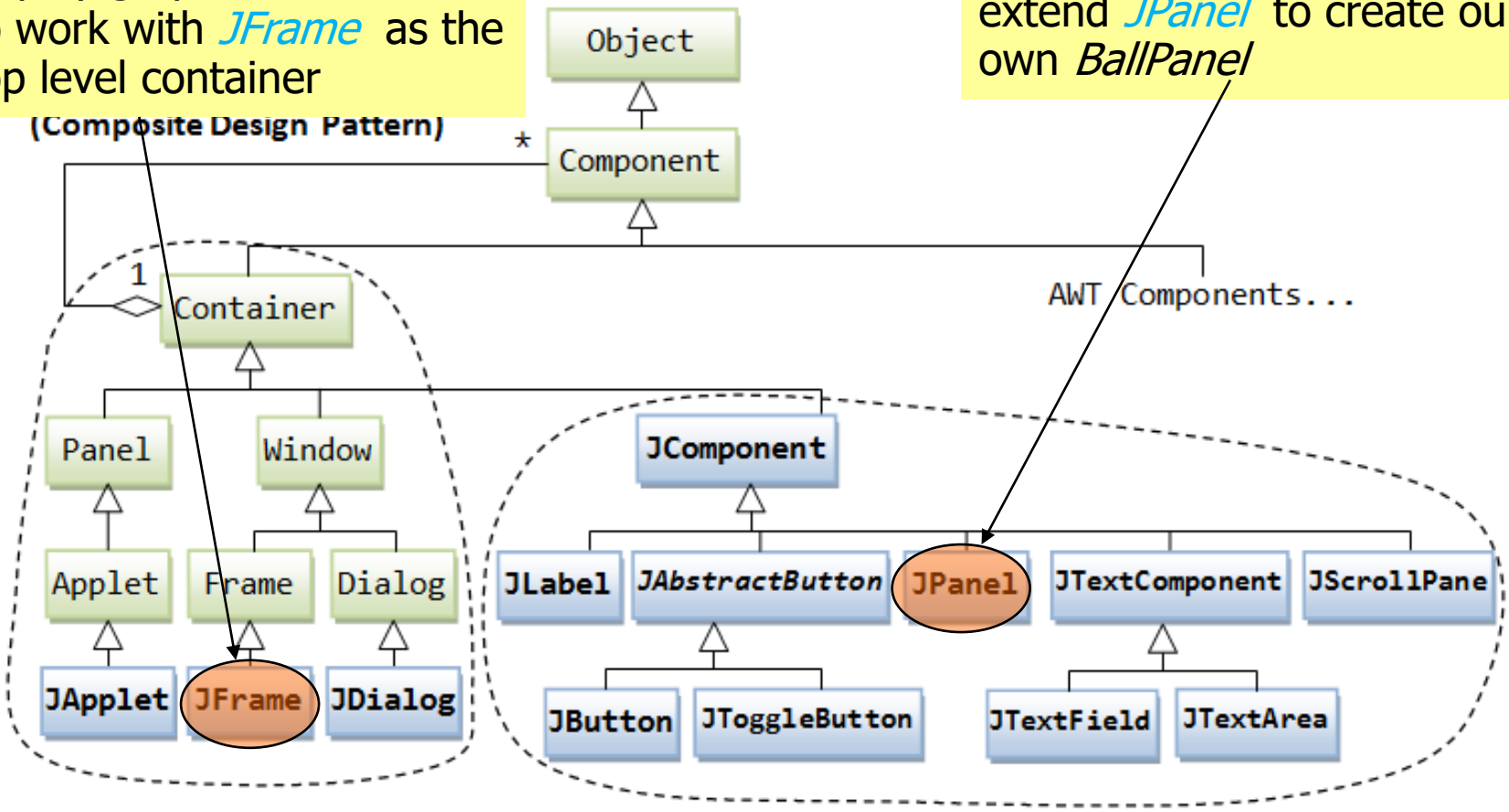
- The UML here is drawn with Violet UML Editor, which you can download for free at: VioletUmlEditor download

Recap: Java Classes for Rendering

- Two types of GUI elements:

To create a window for display graphics, we need to work with *JFrame* as the top level container

To have a panel to draw graphic shapes on, we need to extend *JPanel* to create our own *BallPanel*



- Source of the diagram: http://www.ntu.edu.sg/home/ehchua/programming/java/J4a_GUI.html#zz-2.2

Create Window and Panel

```
//Responsibility: creating canvas for
   rendering objects
class BallPane extends JPanel{
   // fields for properties
    public final static int PAN_WIDTH
    = 600;
    public final static int PAN_HEIGHT
    = 450;

    public BallPane() {
     // call JPanel's constructor
     super();
     // set panel's preferred size
     this.setPreferredSize(new
     Dimension(PAN_WIDTH, PAN_HEIGHT));
    }

    void paintComponent(Graphics g){
     // call JPanel's method for
    clearing background
     super.paintComponent();
    }}
```

```
//Responsibility: creating window frame
   to hold the panel object
class BallApp extends JFrame {
    public BallApp(String title) {
      super(title);
      this.setDefaultCloseOperation(
      JFrame.EXIT_ON_CLOSE);

     BallPane ballPane = new BallPane();
     this.add(ballPane);
     this.pack(); //window's size is set
        //by packing to BallPane's size
     this.setVisible(true);
    }

    //Driver method for project running
    public static void main(String[]
    args){
      new BallApp("BallApp");
    }
}
```

Ex: Define the Ball class

```
class Ball{
   // fields for properties
    private int xPos;
    private int yPos;
    private int ballW;
    private int ballH;
    private Color ballColor;
```

```
// constructor: initialize fields with
fixed values
public Ball() {
    ballW = 50;
    ballH = 50;
    //make it center at the panel's center
    xPos = (BallPane.PAN_WIDTH-ballW)/2;
    yPos = (BallPane.PAN_HEIGHT-ballH)/2;
    ballColor = Color.RED;
}

// method to draw the ball that needs a
Graphics object for drawing
public void drawBall (Graphics g) {
    g.setColor(ballColor);
    g.fillOval(xPos, yPos, ballW, ballH);
}
```

```
class BallPane{
   …
   //constants for panel sizes
    public final static int
    PAN_WIDTH = 600;
    public final static int
    PAN_HEIGHT = 450;
    …
                                    }
```

Golden Rule 1 for Object Communications

- To pass a value from one object to another, make them static constants
  - They can then be accessed by another object by its class name
- Applicable only to insensitive constants in strict encapsulation context – will be penalized if used otherwise in this course

Use the Ball Class to create ball objects

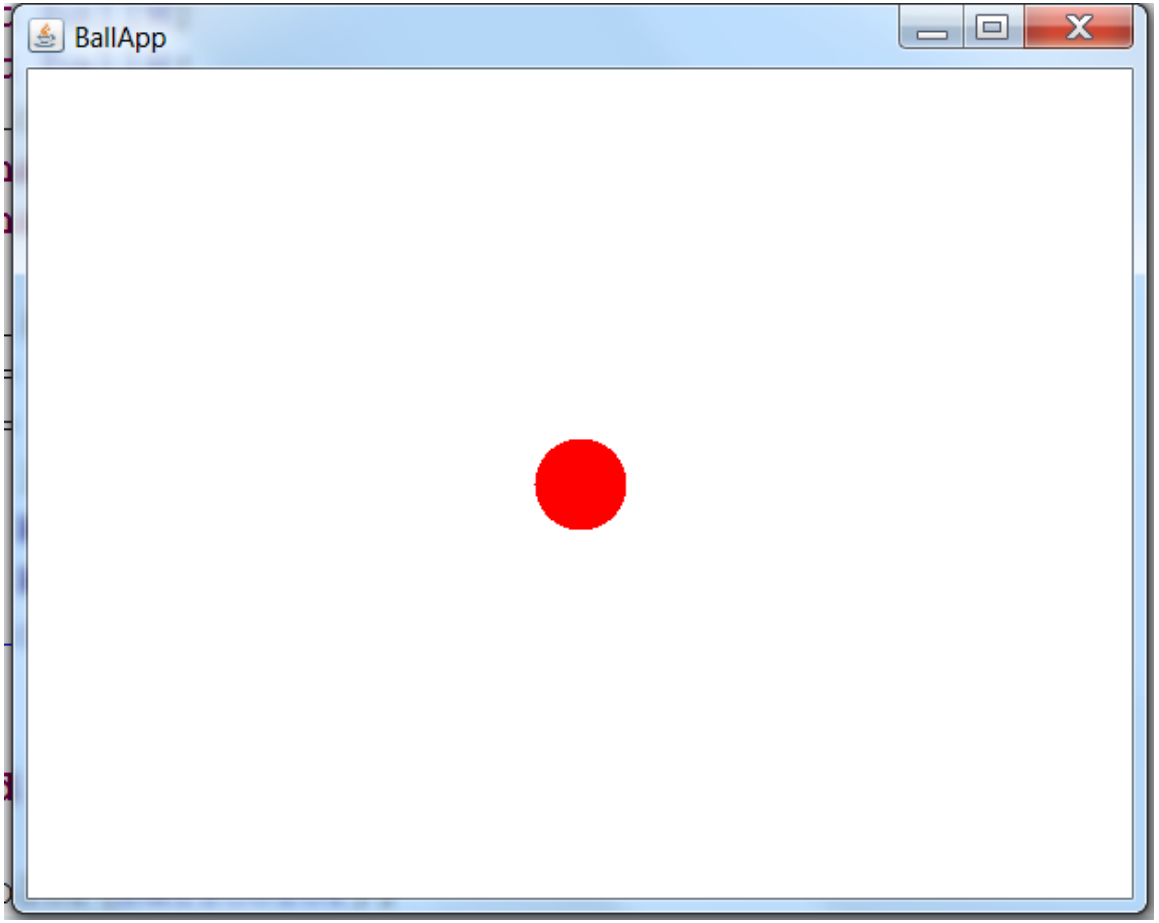
- A class defines a *type* – our Ball class is a user defined type
- You can now *declare* variables using the class and *initialize* them with a new instance of it (*new* + constructor(…))

```
private Ball ball, ball1, ball2, ball3;

public BallPane(){
   ball = new Ball();
   ball1 = new Ball();
   ball2 = new Ball();
   ball3 = new Ball();
}

void paintComponent(Graphics g) {
   super.paintComponent(g);
   ball.drawBall(g);
   ball1.drawBall(g);
   ball2.drawBall(g);
   ball3.drawBall(g);
}
```

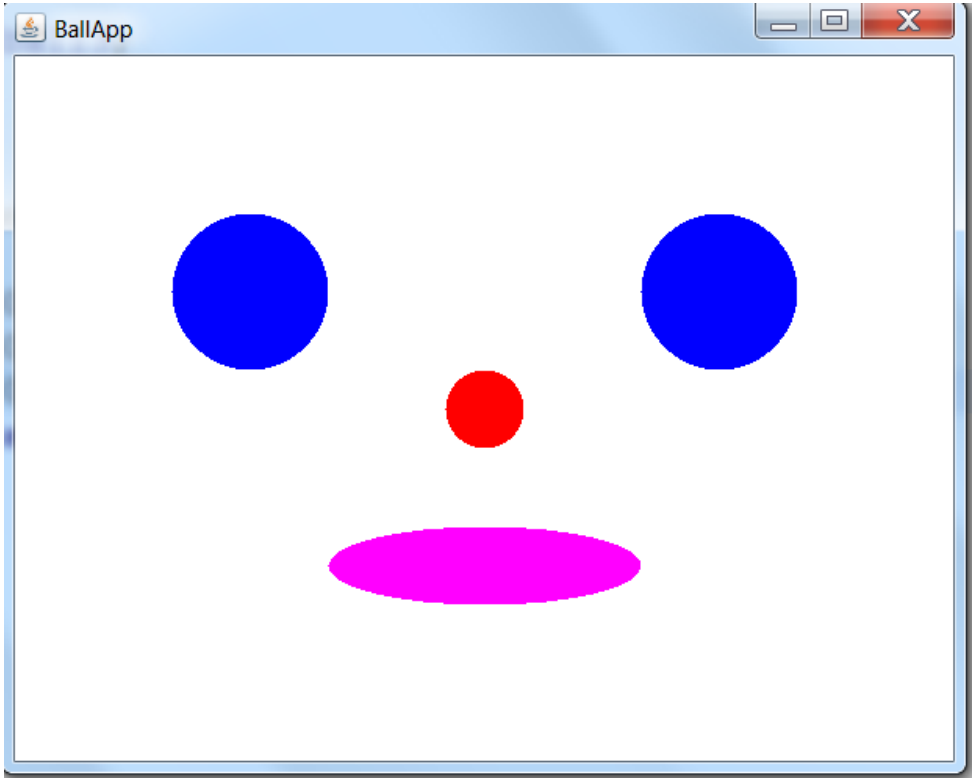
The result ...



- Why there is only one ball displayed?

What if we want to have balls …

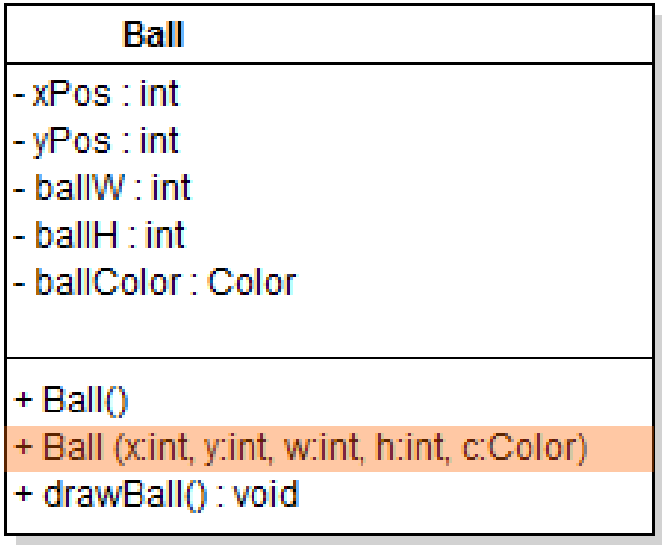
- which have different locations, sizes, shapes, and colors like the followings?



Let's change the code design by …

- Overloading the Ball constructor with parameters for:
  - *x*, *y* coordinates
  - *width* and *height*
  - *color*

- Recap: what is overloading?



- So that we can pass in different arguments to these parameters to create ball objects with different locations, sizes, shapes, and colors

Golden Rule 2 for Object Communications

- To pass in data owned by another object, use parameters
  - Those data will then be passed in as arguments when the method is being called
  - Universally applicable to any communication between different objects

Codify per the updated design

```
class Ball{
   // fields for properties
    private int xPos;
    private int yPos;
    private int ballW;
    private int ballH;
    private Color ballColor;
    …

    // constructor: initialize
    fields with fixed values
    public Ball() {
        …     //same as before
    }
```

```
    // constructor: initialize fields
    with parameters
    public Ball(int x, int y, int w,
    int h, Color c){
        xPos = x;
        yPos = y;
        ballW = w;
        ballH = h;
        ballColor = c;
    }

    // method to draw the ball
    public void drawBall (Graphics g) {
        g.setColor(ballColor);
        g.fillOval(xPos, yPos, ballW,
        ballH);
    }
}
```

Use the new Constructor to create ball objects

- We can now call the *overloaded constructor* to create three ball objects with different locations, sizes, shapes, and colors

```
Private Ball ball, ball1, ball2, ball3;

public BallPane(){
   ball = new Ball();      //still call the old constructor
   ball1 = new Ball(100, 100, 100, 100, Color.BLUE);
   ball2 = new Ball(400, 100, 100, 100, Color.BLUE);
   ball3 = new Ball(200, 300, 200, 50, Color.MAGENTA);
}

void paintComponent(Graphics g) {
  super.paintComponent(g);
  ball.drawBall(g);
  ball1.drawBall(g);
  ball2.drawBall(g);
  ball3.drawBall(g);
}
```

Notes to Parameters

- Methods without parameters give us convenience in terms of their defining and calling, but have to work with fixed values → limiting their functionality

- Methods with parameters, although more demanding in definition and calling, provide more flexible functionalities
  - Arguments for parameters are used to effect the methods' functionalities and generate different outcomes

Summary on Class & Object

- To describe a type of objects – define a class
  - Fields: hold an object's data: properties/states
  - Constructors:
    - special methods used to instantiate objects from a class
    - Features: a) same name as the class; b) no return type
  - Methods: allow objects to do things

- To create (instantiate) and use an object:
  - Declare a reference variable using a class as the type
  - Initialize it with an instance of the class (=*new constructor(…)*)
  - Call the object's methods following the pattern: *referenceVariable.method(…)*

- Constructors and methods can be overloaded – same name different parameters
- A method can effect its functionality and/or output by way of parameters

Animation

- The basic idea behind animation:
  - Draw a graphical image as a frame over and over with slight changes from one frame to another
- The changes may be one or more of the followings:
  - Displacement (motion)
  - Size, orientation, color, shape etc.
- In every frame
  1. Update objects' state (location, direction of movement, orientation, shape, color)
  2. Display modified objects (redraw)

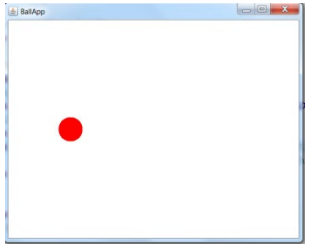
Timer-based Animation

- Swing library provides a *Timer* class specifically for GUI based animation
  - *javax.swing.Timer**

* Please note Java has another Timer class: *java.util.Timer*, which works differently and is a more general-purpose timer. Swing timer is a more appropriate choice for GUI based animation

How Java Timer works

- *javax.swing.Timer* works like a metronome:
  - Once it's started, it ticks away …
  - For each *tick*, certain actions that we can specify are performed

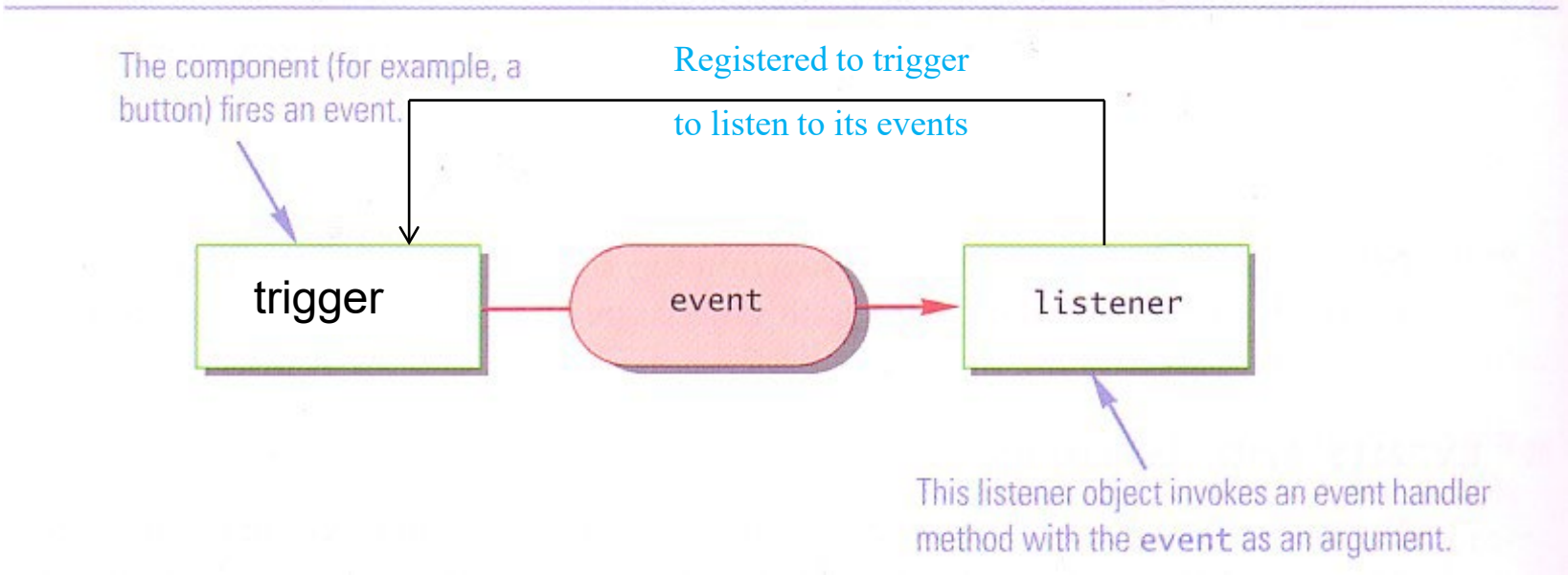


- To understand how this works, you need to know *Java Event Model*
  - We'll use a ball object's motion as an example to demonstrate

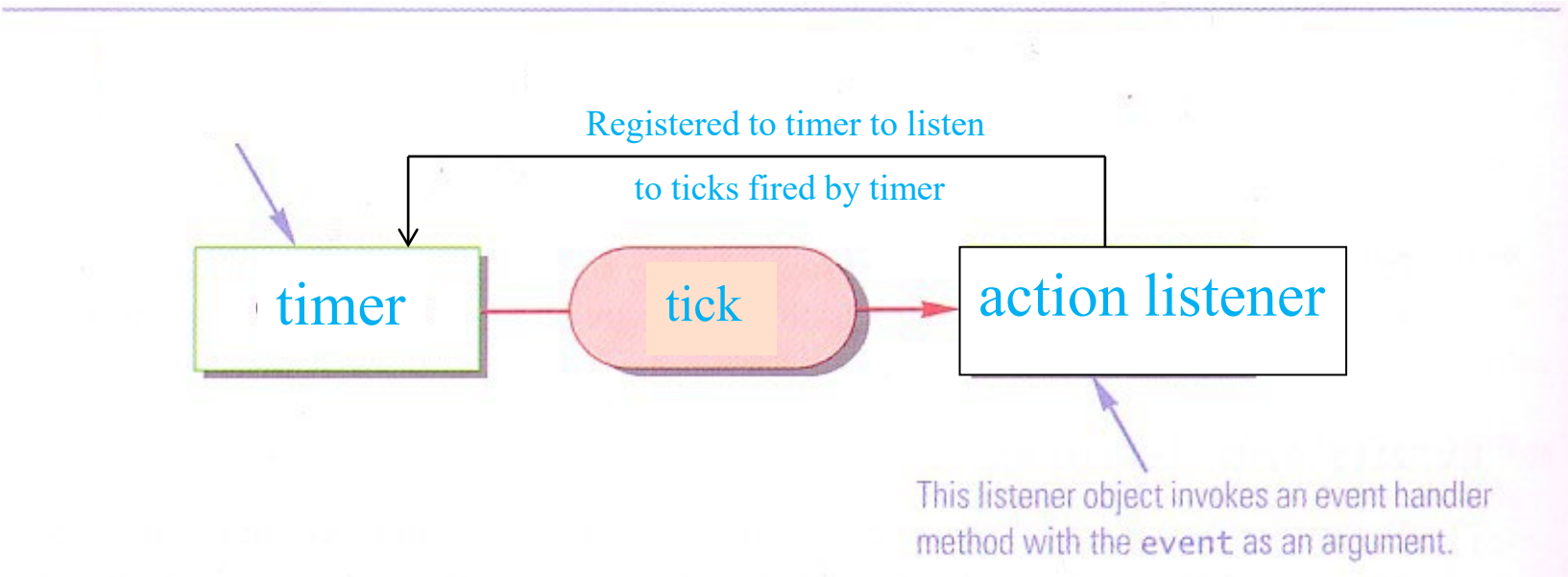
Java Event Model

- Some source object (aka *trigger*, e.g. *timer*, *mouse*, *keyboard*, *button*, *window*, etc.) fires Events
  - Events: *timer ticks*, *clicks a mouse*, *press a key*, *click a button*, *selects a menu item*, *opens a window*, …
- Any object interested in hearing about a particular event can register itself with the source object as a *listener*
- When the event happens, JRE notifies the listeners automatically
- Each listener decides how it will respond with its *specially defined method(s)* - *event handler(s)*

Java Event Model



When it comes to Motion



Specifically how can we make a ball move?

- We need to have:
  - A *Timer object* to fire *ticks*
  - An *ActionListener instance* that registers itself to the *Timer object*
  - The *ActionListener instance* listens to the *ticks* and responds by *updating the ball's position* and then *repaint* it
    - In our example, we have *BallPanel* and *Ball* objects. **Which one should we make an instance of *ActionListener*?**
    - *JPanel Component* (*BallPanel* specifically). Why?

About ActionListener

- How can we turn a component, such as the *BallPanel*, into an action listener?
  - Any Java component can become an *action listener* by implementing the ***ActionListener*** *interface*
  - The *interface* here is referring to another Java construct (beyond *superclass*) used for *inheritance/polymorphism*

Java interface

- We will cover *interface* along the line of *inheritance/polymorphism* in more detail in the future. For now just the *basics* to help us understand *ActionListener*

- An *interface* is a class-like construct that *declares methods without any implementation*. E.g. we can define an interface as follows:

```
public interface Mover {
    void move();
    boolean changeGear(int newGear);
}
```

- As you can see, an *interface* by itself is "*useless*", as it doesn't have any method body with code to be executed
  - For it to function, there must be some class to implement it and thereby implement ALL the methods it declares – with either concrete, executable code or dummy implementation

Class and Interface

- An *Interface* provides a description of a *role* declared with *some capabilities* (via *method declarations*)

- A class can choose to implement the *interface* to assume the role and thereby take on those capabilities – by implementing all the methods declared by the interface with executable code

```
public class MountainBike extends Bicycle implements Mover {
    ...
    void move() {
      xPox += xSpeed;
    }
    boolean changeGear(int newGear) {
      gear =newGear
    }
}
```

- By implementing an interface, it's like a class having signed a contract, enforcing itself to have the functionalities as specified by the interface

ActionListener – a built-in Java interface

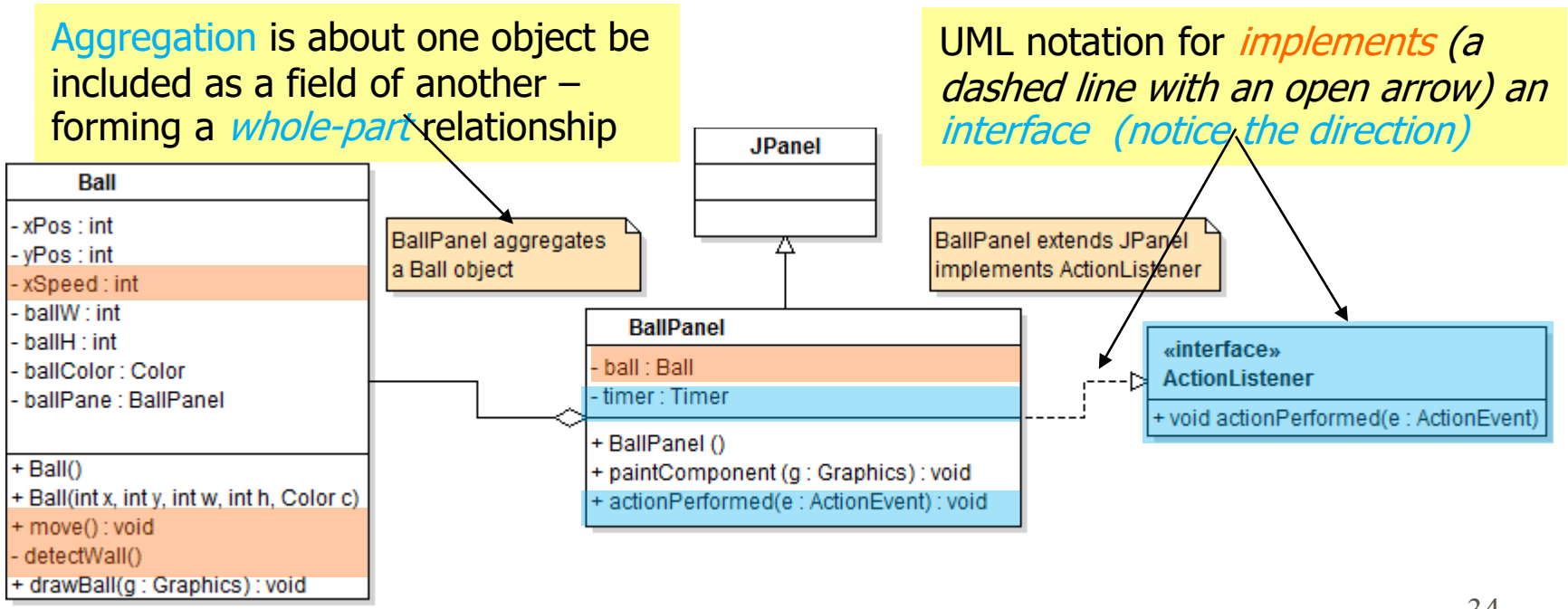
- Defined in *java.awt.event* package as follows:

```
public interface ActionListener extends EventListener{
    void actionPerformed(ActionEvent e);
}
```

- So any class that declares itself "*implements*" the ***ActionListener*** interface must implement the method it declared
  - By doing so, an object of the class becomes an ***ActionListener*** instance, and thereby can register itself to a *timer* to listen to its ticks

To make a ball move …

- Make BallBanel *implements* the **`ActionListener`** interface, and include a *Timer* object and a *Ball* object
- To make the ball move, it should include *speed* attribute and define a *move()* method



Codify per the Design

```
class BallPanel extends JPanel
   implements ActionListener {

   private Ball ball
   private Timer timer;

   //constructors
   public BallPanel() {
       super();
       setPreferredSize(new
        Dimension(600, 450));
       setBackground(Color.white);
       ball = new Ball();
       …
       //create and start timer
       timer = new Timer(33,null);
       timer.start();

       //register BallPanel to timer
       timer.addActionListener(this);
       }
```

```
//All the rest is the same as
   before
   …

//implements the method declared
   by ActionListener
@Override
public void
   actionPerformed(ActionEvent e){
   //Ball's move() will be called
   for each tick of timer
   ball.move();

   //update the display after
   move
   repaint();
}

}
```

Notes to Swing Timer

- Here is Swing Timer's constructor
  - Timer(int *delay*, ActionListener *listener*)
    - *delay* – time interval (in milliseconds) between every two ticks, which is the reversal of *framerate*
      - *framerate* = 1000/*delay*, to have a *framerate* of *30fps* → delay = 33
    - *listener* – an initial ActionListener instance passed in when a Timer object is created, can be *null* if none or add some later

Notes to Swing Timer (1)

- There are two ways to register an *ActionListener* object to a *Timer* object
  - By passing in as an argument when a *Timer* object is created, e.g.:

```
//Passing BallPanel to timer when it's created
timer = new Timer(33,this);
```

  - By calling Timer's method *addActionListener(ActionListener listener)* :

```
//no initial listener passed in
timer = new Timer(33,null);

//register BallPanel to timer
timer.addActionListener(this);
```

- The former only allows to have one listener registered, the latter allows to have as many as you want

Notes to *implements* interface

- Although Java allows a class to "*extends*" from a single class only, however, it allows it to "*implements*" as many interfaces as necessary, as long as they are delimited with ","

- This is a nice feature, as it will allow our program to have animations and mouse interactions simultaneously, e.g.:

```
class BallPanel extends JPanel implements
ActionListener, MouseListener {

   …              //Need to implement here all methods
                    declared by both interfaces
}
```

Summary on Timer-Based Motion

- To create animations, you need to use a Swing Timer object
  - *javax.swing.Timer* (rather than *java.util.Timer*)
- Timer-based animation is based on Java Event Model
  - *Timer object* fires *ticks*, *ActionListener* objects can register to the *Timer object* to listen to its *ticks*, and respond by *updating the objects' positions* and then *repaint* it
- To make any object a *timer listener*, e.g. the *JPanel* instance, make it "implements" the *ActionListener* interface
- An *Interface* is a Java construct that provides a *description of a role with some capabilities* - via ***method declarations*** *without implementation*
  - A class can choose to implement an interface to assume the role and thereby provide the capabilities as declared

Random Behavior

- There is often a need to have some values randomly fluctuate within a range of values
  - E.g. a random location on the screen
    or a random speed within the <0, maxSpeed>

Math.random()

- Returns values between [0.0,1.0)
- To get a value *x* within [a, b)
  - *x = a+ Math.random()*(b-a)*

```
ball1 = new Ball(100, 100, 100, 100, Color.BLUE,
  (int)(1+ Math.random()*5)); //within [1, 6)

ball2 = new Ball(400, 100, 100, 100, Color.BLUE,
  (int)(-2.0 + Math.random()*4)); //within [2.0, 2.0)
```

- Demo

Questions for Java's *paint method*

public void paintComponent(Graphics g)

- If you check the case studies, do you see anywhere it gets called?
- If not, what triggers it to operate in the window environment?

Painting Mechanism of AWT and Swing

- System-triggered *painting* when:
  - the component (the *JPanel* object) is first made visible on the screen
  - The component (the *JPanel* object) is resized
  - So in this sense, it is a *call-back method* (i.e. called by the system based on some events)

- Application-triggered *painting* :
  - *when objects (like the balls in our case studies) get updated,* program calls *repaint()* which calls *paintComponent()* implicitly

- So *paintComponent()* method is only a *semi-call-back method*

Encapsulation

- With Data Hiding

Object-Oriented Programming (OOP)

- Object-Oriented Programming **(OOP)**: about creating programs that perform their actions via interactions between objects

- Such interaction is really a process of information communication: *storing*, *processing*, and *messaging* (*sending* and *receiving*)
  - An Object can provide services (with the data it holds) to other objects

  - Other objects that use the object (and therefore its services) are its clients (aka client objects or programs)

Objects as Client

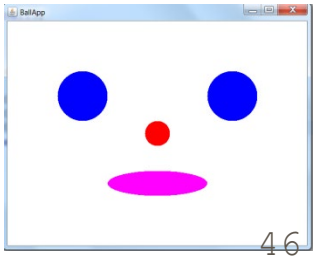
- **Client object**: an object that uses another object's functionalities
  - Example: **`BallPanel`** is a *client object* of ***`Ball`*** object

BallPanel.java (client program)

```
public class BallPanel … {
  public BallPanel(){
      ball = new Ball();
      ...
  }
  void paint(Graphics g) {
      ball.drawBall(g);
      ...
  }
}
```

Ball.java (class)

```
public class Ball {
    ...
}
```



OOP's Principles

- As OOP is actually a communication process among objects, some *protocols*, just like other communication system (e.g. Internet), needs to be followed

- Three general principles of OOP
  - **Encapsulation** – our major focus next
  - Inheritance
  - Polymorphism

Encapsulation

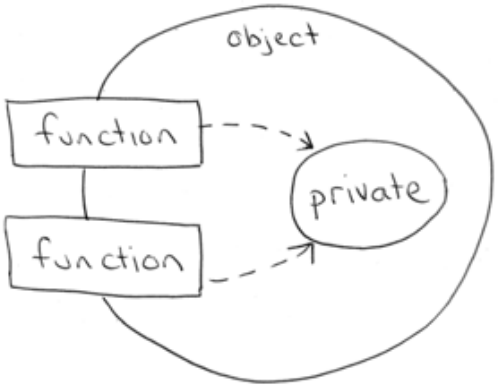
- Encapsulation is defined with two meanings:
  - An Object holds within itself both attributes and behaviors (what we had focused on in IAT-167)
  - An Object hides its data from client objects
    - Data hiding is actually the major part of encapsulation

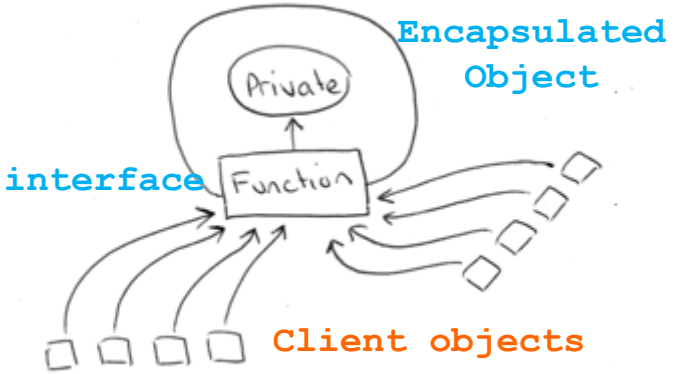
Data Hiding

- Hide an object's attributes from client objects using private keyword
  - *private type name*
- Client objects can only perform interaction with private fields of an object through it's public methods (if any)
  - Such methods serve as the interface* for the object to interact with client objects

* Note: we use the term interface here in a generic sense rather than Java's special construct *interface – which we have just discussed*

Data Hiding and Interfacing





- This means that the data is locked away inside the object
- The public methods (aka functions) provide the only means for doing something with that data
- These public methods served as object's interface to client objects
- With encapsulation, the data inside an object can only be used or mutated by calling the object's public methods

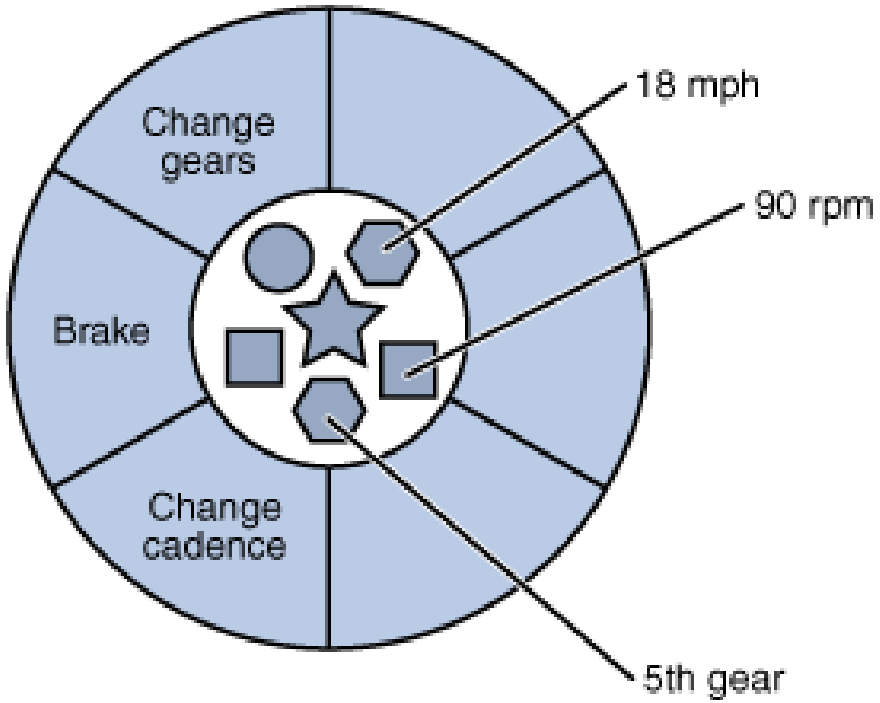
Encapsulation for a Bicycle object

- Properties/States

  int gear ;
  float speed ;
  float cadence ;

- Behaviors

  void setGears(int g);
  void setCadence(float c);
  int getGear();
  float getSpeed();
  void brake(float level);
  ...



Encapsulation for Bicycle object

- An object's private fields can't be accessed by any client programs

```
class Bicycle  {
   private int cadence = 0;
   private int speed = 0;
   private int gear = 1;

     //Constructor
   public Bicycle () { }

} //end of Bicycle

//Tried to access privates from a client program - BikeApp
void main(String[] args) {
   Bicycle bike = new Bicycle ();
   if(bike.gear != 5)
     bike.gear = 5;
   System.out.print(bike.gear);
}
```

Illegal !!
Can't access or change private fields from a client program

public getter and setter methods

- In OOP, getter and setter methods, if defined, are the common way for client programs to access or change an object's private fields

- The typical patterns for getter and setter methods' signatures:
  - to return the current value of a private field, we need a getter method (aka *accessor*):

    public varType getVarName()

    e.g. `public int getGear()`

  - to change the value of a private field, we need a setter method (aka *mutator*):

    public void setVarName(varType newValue)

    e.g. `public void setGear(int g)`

Example of Setter & Getter

```
class Bicycle {
      …
      private int gear = 1;
      …
     //Allows clients to change
       the gear's value
     public void setGear(int g){
       gear = g;
     }

     //Allows client to get
       the gear value
     public int getGear(){
       return gear;
     }
}
```

```
//To access private field via getter and
   setter from a client program

void main(String[] args) {
   Bicycle bike = new Bicycle ();

    //If the current gear is not 5
   Call the setter to set gear
   if(bike.getGear()!= 5){
      bike.setGear(5);
    }

   //Call the getter to return the
   current gear value
   System.out.print(bike.getGear());
}
```

Golden Rule 3 for Object Communications

- By way of defining and calling getter/setter methods
  - getter method allows you to get data from another object - by way of return value
  - setter method allows you to change the data of another object – by way of parameter

Why Hide Data?

- Controls access
  - Prevents data from being tampered maliciously or incidentally by client programs, e.g. unwanted increase or decrease of an `Account`'s balance
- Ensures data integrity when necessary
  - For instance, we can ensure the gear value assigned doesn't go out of range of 1 to 10 as follows:

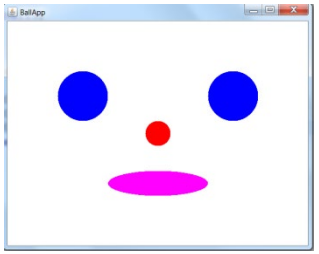
```
void setGear(int g) {
  if(g < 1 || g > 10)
      System.out.print ("wrong data!");
  else
      gear = g;
}
```

- Good for code maintainability
  - When an object updates its internal implementation, as long as the interface stays the same, the client programs don't need to change its implementation

Design Encapsulation: Keep the interface Minimal

- When designing a class, it's important to maintain the availability of the essential services that its objects provide but minimize exposure of their data as little as possible
- This means that an object should provide the minimal public interface
  - The goal is to provide the client programs with the exact interface to do their jobs – no more no less
  - If the public interface is NOT properly restricted, it'll lead to problems that might break the system's integrity and security

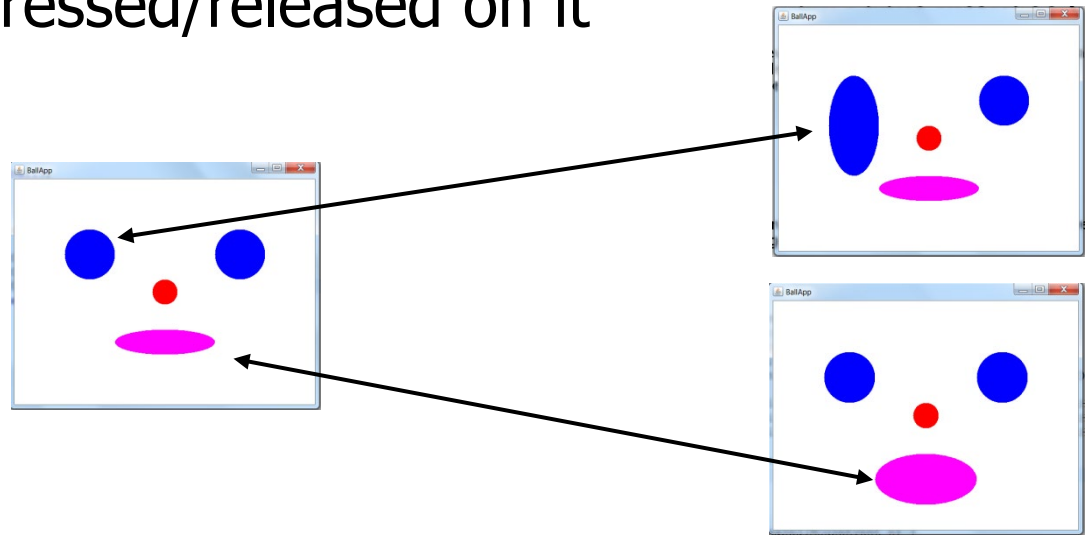
Case Study: Ball class with Minimal Public Interface



- So far, the Ball class has only two public methods for its interface: *drawBall()* and *move()* methods
- If our goal is just to create some Ball objects and display them, the current interface is good enough
- We don't need to provide *getter* or *setter* methods for clients to access or mutate any of its private fields

Case Study: Ball class with Minimal Public Interface (1)

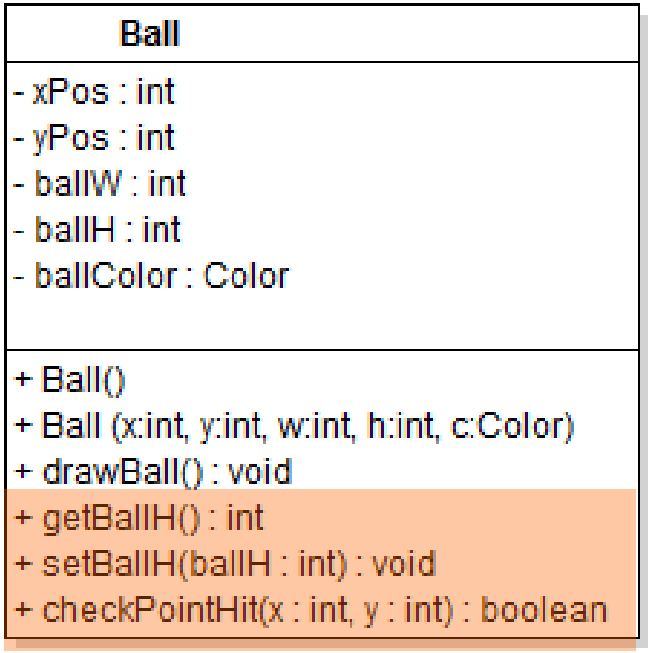
- However, if we now change the goal to make the `Ball` object double/restore its height whenever the mouse is pressed/released on it



- Apparently the panel needs a way to *retrieve* and *change* the ball's height based on the mouse event

Modify the Public Interface to meet the Goal

- To achieve this, the client program must be able to call some methods to:
  - Retrieve its current *ballH*
  - Reset it to a new value (e.g. doubling)
- Apparently we need to provide getter and setter methods for *ballH*
  - Beyond that we need a method to determine if a point (e.g. mouse cursor) hits on the ball or not



- Since the new goal doesn't need to touch on any other Ball's properties, we will provide only methods as such to keep the public interface minimal

Codify per the updated design

```
class Ball{
   // fields for properties
    private int xPos;
    private int yPos;
    private int ballW;
    private int ballH;
    private Color ballColor;


    //constructors
        …     //same as before

    //Method to draw the ball
                  …     //same as
       before
```

```
//Allows clients to change ballH
public void setBallH(int h){
   ballH = h;
}

//A "read-only" access to ballH
public int getBallH(){
   return ballH;
}

//method to detect a point hitting
   on the ball object
public boolean checkPointHit(int x,
   int y) {
    boolean hit = false;
    if (x > xPos && x < xPos + ballW
    && y > yPos && y < yPos + ballH){
       hit = true;
    }
    return hit;
}
}
```

Call the new methods from BallPanel

- We will cover mouse interaction later, for now just focus on how these new methods are used in the client program

```
void mousePressed(MouseEvent e) {
    //return mouse's current position
    int x = e.getX();
    int y = e.getY();

    if(ball.checkPointHit(x, y))
       ball.setBallH(ball.getBallH()*2);
    if(ball1.checkPointHit(x, y))
       ball1.setBallH(ball1.getBallH()*2);
    if(ball2.checkPointHit(x, y))
       ball2.setBallH(ball2.getBallH()*2);
    if(ball3.checkPointHit(x, y))
       ball3.setBallH(ball3.getBallH()*2);

       repaint(); //call paint()implicitly
                  to update display
}
```

```
void mouseReleased(MouseEvent e){
    //return mouse's current position
    int x = e.getX();
    int y = e.getY();

    if(ball.checkPointHit(x, y))
       ball.setBallH(ball.getBallH()/2);
    if(ball1.checkPointHit(x, y))
       ball1.setBallH(ball1.getBallH()/2);
    if(ball2.checkPointHit(x, y))
       ball2.setBallH(ball2.getBallH()/2);
    if(ball3.checkPointHit(x, y))
       ball3.setBallH(ball3.getBallH()/2);

       repaint(); //call paint()implicitly
                  to update display
}
```

Summary on Encapsulation

- Encapsulation is defined with two meanings
  - An Object wraps both attributes and behaviors within itself
  - An Object hides its data from client programs
- Data hiding
  - Use private keyword to hide fields from client programs
  - Use public getter/setter or other methods to allow client program to access part or all of the fields when it's necessary
- Data hiding is the major part of encapsulation, which uses access control to ensure object data's integrity and security as well as code maintainability
- One goal of designing a class for encapsulation is to keep the public interface **minimal** to protect system's integrity and safety

Readings

- Required
  - Reading pack in Canvas