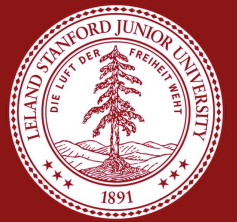
Final Exam Review 2

CS106A, Summer 2019

Trey Connelly && Aditya Chander
Slides mostly from Sarai Gould && Laura Cruz-Albrecht



Plan for Today

- HashMaps
- Classes
- Interactors
- Closing Remarks

Plan for Today

- HashMaps
- Classes
- Interactors
- Closing Remarks

HashMaps

A variable type that represents a collection of key-value pairs

- You access values by key, and all keys are unique
- Keys and values can be any type of Object (use wrapper classes to store primitives)
- Resizable – can add and remove pairs
- Has a variety of methods you can use, including `.containsKey`, `.put`, `.get`, `.keySet`, etc.

HashMaps

Data Type of "keys" in our HashMap

Data Type of "values" in our HashMap

Repeated types of key->value pairs

```
HashMap<String, String> firstMap = new HashMap<String, String>();
```

Our First HashMap

```
import java.util.*;
HashMap<String, String> sounds = new HashMap<String, String>();
```

sounds

Our First HashMap

```
import java.util.*;
HashMap<String, String> sounds = new HashMap<String, String>();
sounds.put("dog", "woof");
sounds.put("cat", "meow");
```

sounds

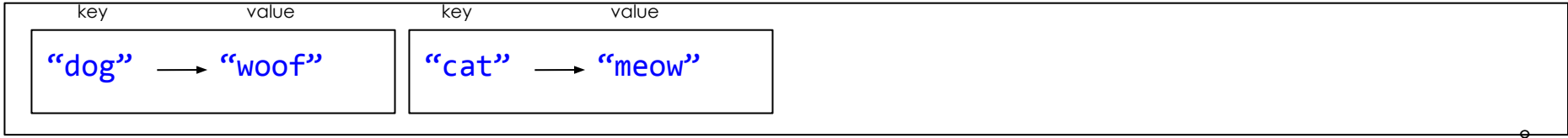
key	value	key	value
"dog"	"woof"	"cat"	"meow"

Our First HashMap

```
import java.util.*;
HashMap<String, String> sounds = new HashMap<String, String>();
sounds.put("dog", "woof");
sounds.put("cat", "meow");

String dogSound = sounds.get("dog");
println(dogSound);
println(sounds.get("cat"));
```

```
woof
meow
```

sounds

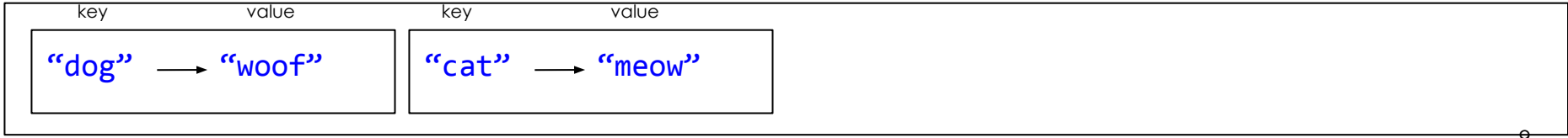


Our First HashMap

```
import java.util.*;
HashMap<String, String> sounds = new HashMap<String, String>();
sounds.put("dog", "woof");
sounds.put("cat", "meow");

String dogSound = sounds.get("dog");
println(dogSound);
println(sounds.get("cat"));
println(sounds.get("hippopotamus")); // this isn't in our map!
```

```
woof
meow
```

sounds

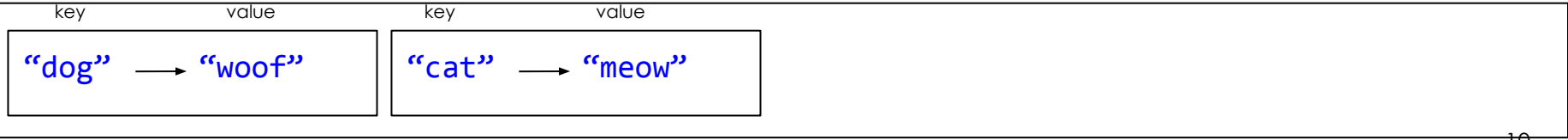


Our First HashMap

```
import java.util.*;
HashMap<String, String> sounds = new HashMap<String, String>();
sounds.put("dog", "woof");
sounds.put("cat", "meow");

String dogSound = sounds.get("dog");
println(dogSound);
println(sounds.get("cat"));
println(sounds.get("hippopotamus")); // this isn't in our map!
```

```
woof
meow
null
```

sounds

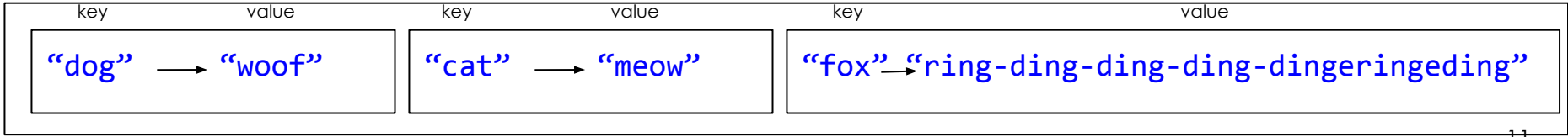


Our First HashMap

```
import java.util.*;
HashMap<String, String> sounds = new HashMap<String, String>();
sounds.put("dog", "woof");
sounds.put("cat", "meow");
// what does the fox say?*
sounds.put("fox", "ring-ding-ding-ding-dingeringeding");

String dogSound = sounds.get("dog");
println(dogSound);
println(sounds.get("cat"));
println(sounds.get("hippopotamus")); // this isn't in our map!
```

```
woof
meow
null
```

sounds

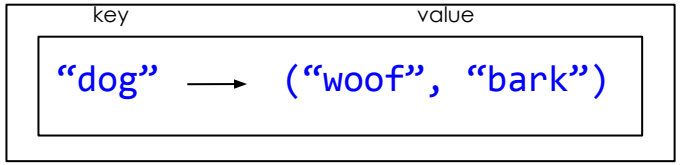


* Music reference… Not just crazy.

Our *Improved* HashMap

```
import java.util.*;

HashMap<String, ArrayList<String>> animalSounds = new HashMap<>();

ArrayList<String> dogSounds = new ArrayList<>();

dogSounds.add("woof");

dogSounds.add("bark");

animalSounds.put("dog", dogSounds);
```

sounds

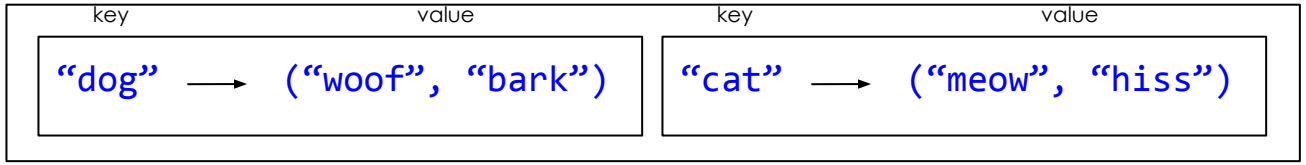


Our *Improved* HashMap Part 2

```
...
animalSounds.put("cat", catSounds); // (meow, hiss)

...
for(String animal : animalSounds.getKeySet()){

    ArrayList<String> sounds = animalSounds.get(animal);

    for(String sound : sounds){
        println(sound);
    }
}
```

```
woof
bark
meow
hiss
```

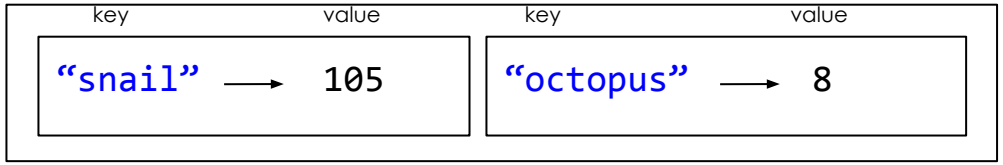
sounds



Review: What Does This Code Do?

```
import java.util.*;
HashMap<String, Integer> numLimbs = new HashMap<>();
numLimbs.put(“dog”, 4);
numLimbs.remove(“dog”);
numLimbs.put(“snail”, 1);
numLimbs.put(“snail”, 105);
numLimbs.put(“octopus”, 8);
println(numLimbs.get(“dog”));
println(numLimbs.get(“snail”));
println(numLimbs.get(“octopus”));
```

numLimbs

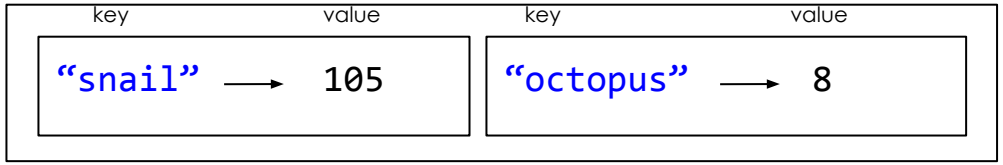


Review: What Does This Code Do?

```
import java.util.*;
HashMap<String, Integer> numLimbs = new HashMap<>();
numLimbs.put("dog", 4);
numLimbs.remove("dog");
numLimbs.put("snail", 1);
numLimbs.put("snail", 105);
numLimbs.put("octopus", 8);
println(numLimbs.get("dog"));
println(numLimbs.get("snail"));
println(numLimbs.get("octopus"));
```

```
null
105
8
```

numLimbs

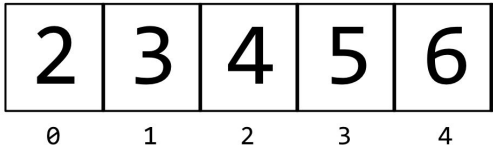


Review: HashMap Commands

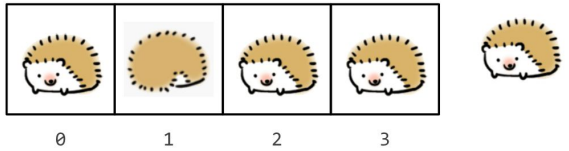
```
HashMap<String, String> myMap = new HashMap<String, String>();

// Adds key->value pairs
myMap.put("dog", "woof");
myMap.put("cat", "meow");

// Removes key and the associated value
myMap.remove("dog", "woof");

// A beautiful way to access each key
for (String key : myMap.keySet()) {
     println(key);
     // print the value associated with the key
     println(myMap.get(key));
}
```

Review: Data Structures

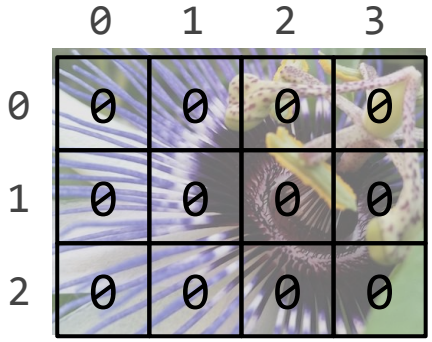
Arrays



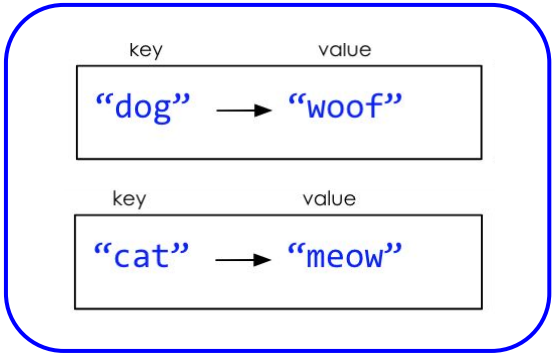
ArrayLists



2D Arrays



HashMaps



Which Data Structure to Use?

- Use an array if…
  - Order matters for your information
  - You know how many elements you will store
  - You need the most efficiency
- Use an ArrayList if…
  - Order matters for your information
  - You do not know how many elements you will store, or need to resize
  - You need to use ArrayList methods
- Use a HashMap if…
  - Order doesn't matter for your information
  - You need to store an association between two types of information
  - You do not know how many elements you will store, or need to resize
  - You need to use HashMap methods

Which data structure to use?

Data structure name	Ordered?	Fixed size?	Value only or key-value?	Can store primitives?
array/2D array	yes	yes	Value only	yes
ArrayList	yes	no	Value only	no (objects only)
HashMap	no	no	Key-value	no (objects only)

Plan for Today

- HashMaps
- Classes
- Interactors
- Closing Remarks

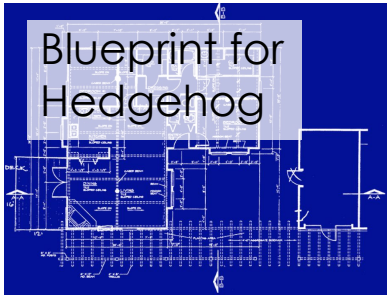
A class defines a new variable type.

Classes Are Like Blueprints

Hedgehog Class (blueprint)

State: Has name
Has color
Has cuteness level

Behavior: Can eat
Can run*
Can curl up



Hedgehog #1 (variable)

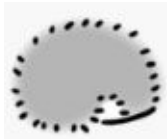
State: **name = "Walnoot"**
color = Brown
cuteness = 10 (Very cute)

Behavior: Can eat
Can run
Can curl up

Hedgehog #2 (variable)

State: **name = "Nutmeg"**
color = Snowflake
cuteness = 15 (VERY cute)

Behavior: Can eat
Can run
Can curl up



Hedgehog #3 (variable)

State: **name = "Ruffles"**
color = Beige
cuteness = 50 (speechless)

Behavior: Can eat
Can run
Can curl up

Making a Class ~ 3 Ingredients

1. Define the **variables** each instance stores (state)
2. Define the **constructor** used to make a new instance
3. Define the **methods** you can call on an instance (behaviors)

Using Constructors

```
BankAccount dukeAccount = new BankAccount("Duke", 50);
BankAccount karelAccount = new BankAccount("Karel");
```

dukeAccount

```
name = "Duke"
balance = 50

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

karelAccount

```
name = "Karel"
balance = 0

BankAccount(name) {
     this.name = name;
     this.balance = 0;
}
```

When you call a constructor (with **new**):

1. Java creates a new "instance" of that class
2. The constructor initializes the object's state (instance variables)
3. The newly created object is returned to your program*

Review: BankAccount

```
public class BankAccount {
	// Step 1: the data inside a BankAccount
	private String name;
	private double balance;

	// Step 2: how to create a new BankAccount
	public BankAccount(String name) {
		this.name = name;
		this.balance = 0;
	}
	// Step 3: the things a BankAccount can do
	public void deposit(double amount) {
		this.balance += amount;
	}
	public boolean withdraw(double amount) {
		if (this.balance >= amount) {
			this.balance -= amount;
			return true;
		}
		return false;
	}
}
```

Review: Getters & Setters

```
public class BankAccount {
     private String name;
     private double balance;
     …
     // “setter”
     public void setName(String newName) {
          if (newName.length() > 0) {
               this.name = newName;
          }
     }
     // “getters”
     public String getName() {
          return this.name;
     }
     public double getBalance() {
          return this.balance;
     }
}
```

GRect.java

```
public class GRect {
   private double width;
   public GRect(double width, double height) {
      …
   }
  …
}
```

Unpacking GRect

GRect.java

```
public class GRect {
```

3 Ingredients:

GRect.java

```java
public class GRect {

  // 1. Instance variables
  private double width = 0;
  private double height = 0;
  private double yc = 0;
  private double xc = 0;
  private boolean isFilled = false;
  private boolean isVisible = false;
```

3 Ingredients:

1. Define the variables each instance stores

GRect.java

```java
public class GRect {

  // 1. Instance variables
  private double width = 0;
  private double height = 0;
  private double yc = 0;
  private double xc = 0;
  private boolean isFilled = false;
  private boolean isVisible = false;

  // 2. Constructor(s)
  public GRect(double width, double height) {
    this.width = width;
    this.height = height;
  }
```

3 Ingredients:

1. Define the variables each instance stores

2. Define the constructor used to make a `new` instance

GRect.java

```java
public class GRect {

  // 1. Instance variables
  private double width = 0;
  private double height = 0;
  private double yc = 0;
  private double xc = 0;
  private boolean isFilled = false;
  private boolean isVisible = false;

  // 2. Constructor(s)
  public GRect(double width, double height) {
    this.width = width;
    this.height = height;
  }

  public GRect(double x, double y,
               double width, double height) {
    this.xc = x;
    this.yc = y;
    this.width = width;
    this.height = height;
  }
```

3 Ingredients:

1. Define the variables each instance stores

2. Define the constructor used to make a `new` instance

GRect.java

```java
public class GRect {

  // 1. Instance variables
  private double width = 0;
  private double height = 0;
  private double yc = 0;
  private double xc = 0;
  private boolean isFilled = false;
  private boolean isVisible = false;

  // 2. Constructor(s)
  public GRect(double width, double height) {
    this.width = width;
    this.height = height;
  }

  public GRect(double x, double y,
               double width, double height) {
    this.xc = x;
    this.yc = y;
    this.width = width;
    this.height = height;
  }

  // 3. Public methods
  public double getWidth() {
    return this.width;
  }

  public double getHeight() {
    return this.height;
  }

  public void setFilled(boolean newIsFilled) {
    this.isFilled = newIsFilled;
  }

  public void move(double dx, double dy) {
    this.xc += dx;
    this.yc += dy;
  }

}
```

3 Ingredients:

1. Define the variables each instance stores
2. Define the constructor used to make a `new` instance
3. Define the methods you can call on an instance

Making a Ball variable type

1. Define the variables each instance stores (think: state/properties)

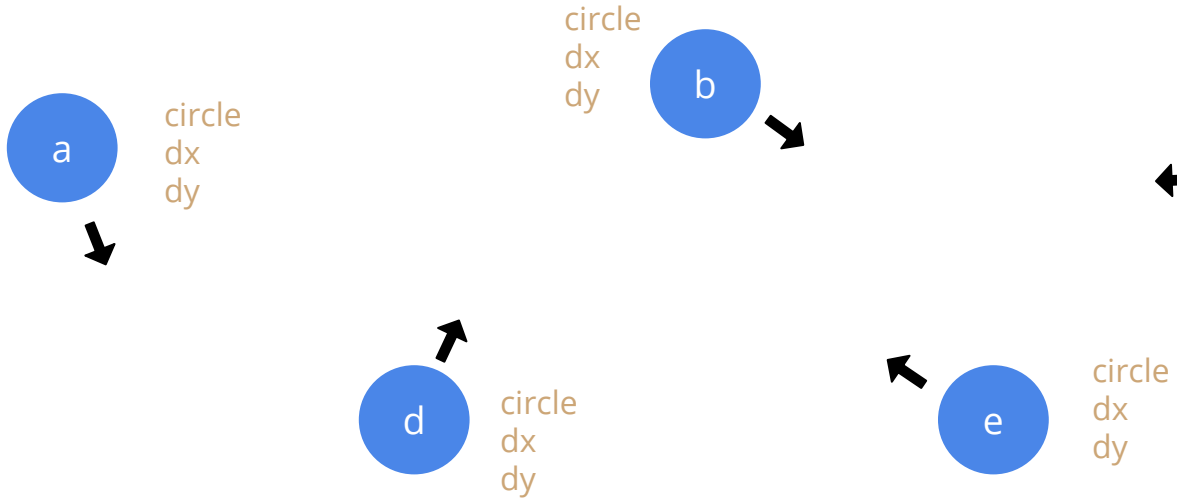
   Each ball has its own GOval (let's call it circle)
   Each ball has its own dx
   Each ball has its own dy

2. Define the constructor used to make a `new` instance

   Set initial values for all the instance vars

3. Define the methods you can call on an instance (think: behaviors)

   heartbeat()
   getGOval()



But if each Ball instance has a copy of each instance variable...

... how does Java know which one to use?

`this`

* all class methods and constructors have access to a `this` reference

GRect.java

```java
public class GRect {

  // 1. Instance variables
  private double width = 0;
  private double height = 0;
  private double yc = 0;
  private double xc = 0;
  private boolean isFilled = false;
  private boolean isVisible = false;

  // 2. Constructor(s)
  public GRect(double width, double height) {
    this.width = width;
    this.height = height;
  }

  public GRect(double x, double y,
               double width, double height) {
    this.xc = x;
    this.yc = y;
    this.width = width;
    this.height = height;
  }

  // 3. Public methods
  public double getWidth() {
    return this.width;
  }

  public double getHeight() {
    return this.height;
  }

  public void setFilled(boolean newIsFilled) {
    this.isFilled = newIsFilled;
  }

  public void move(double dx, double dy) {
    this.xc += dx;
    this.yc += dy;
  }

}
```

3 Ingredients:

1. Define the variables each instance stores
2. Define the constructor used to make a `new` instance
3. Define the methods you can call on an instance

GRect.java

```java
public class GRect {

  // 1. Instance variables
  private double width = 0;
  private double height = 0;
  private double yc = 0;
  private double xc = 0;
  private boolean isFilled = false;
  private boolean isVisible = false;

  // 2. Constructor(s)
  public GRect(double width, double height) {
    this.width = width;
    this.height = height;
  }

  public GRect(double x, double y,
               double width, double height) {
    this.xc = x;
    this.yc = y;
    this.width = width;
    this.height = height;
  }

  // 3. Public methods
  public double getWidth() {
    return this.width;
  }

  public double getHeight() {
    return this.height;
  }

  public void setFilled(boolean newIsFilled) {
    this.isFilled = newIsFilled;
  }

  public void move(double dx, double dy) {
    this.xc += dx;
    this.yc += dy;
  }

}
```

3 Ingredients:

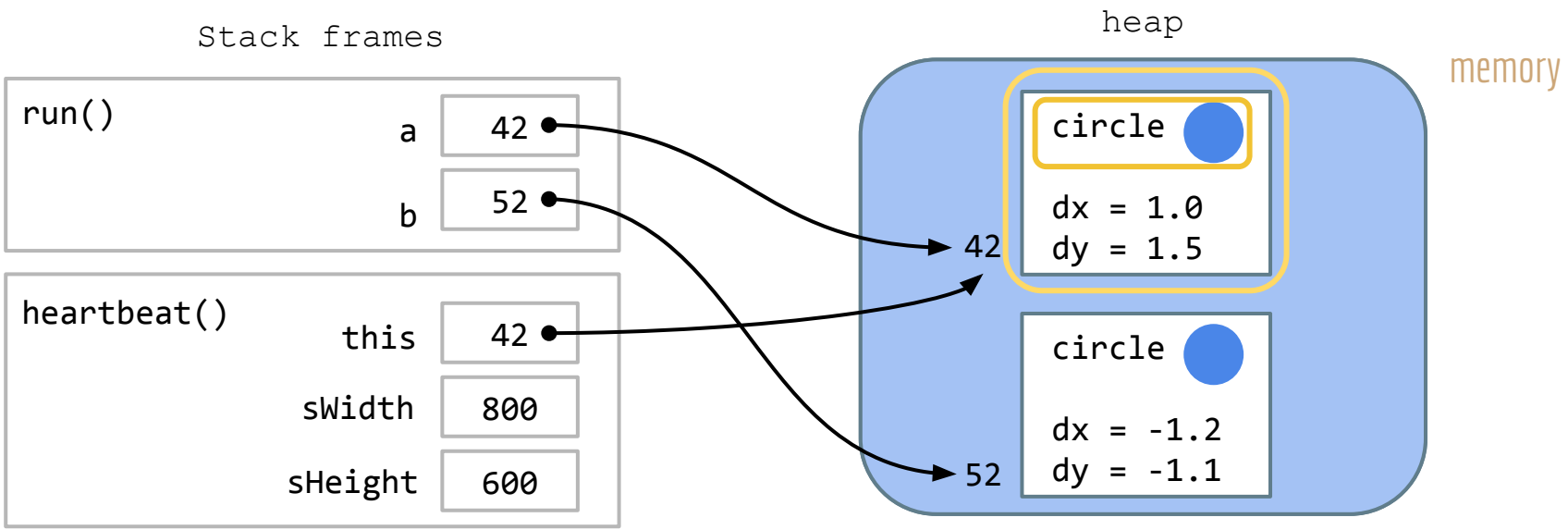
1. Define the variables each instance stores
2. Define the constructor used to make a `new` instance
3. Define the methods you can call on an instance

```
public class BouncingBall extends GraphicsProgram {
        public void run() {
                // make a few new bouncing balls
                Ball a = new Ball();
                Ball b = new Ball();

                // call a method on one of the balls
                a.heartbeat(getWidth(), getHeight());
        }
}
```

```
public void heartbeat(int sWidth, int sHeight) {
        this.circle.move();
        reflectOffWalls(sWidth, sHeight);
}
```

heartbeat() was called on ball **a**
⇒ So, **this** refers to **a**

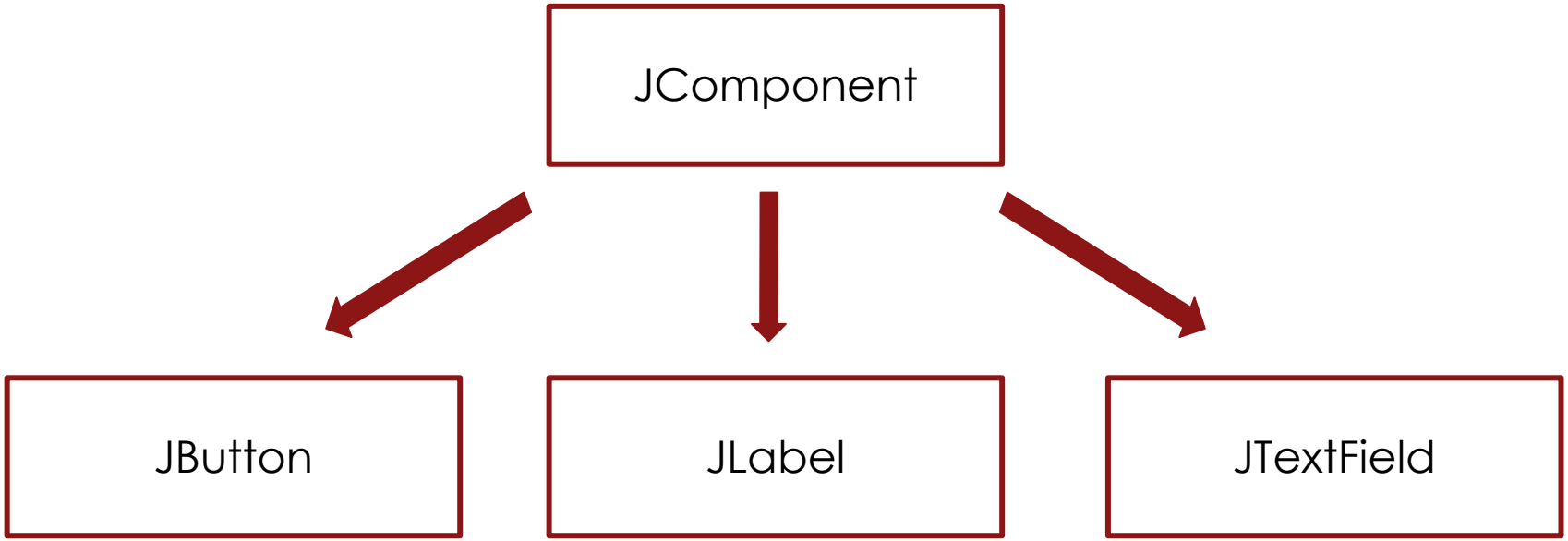
code



Plan for Today

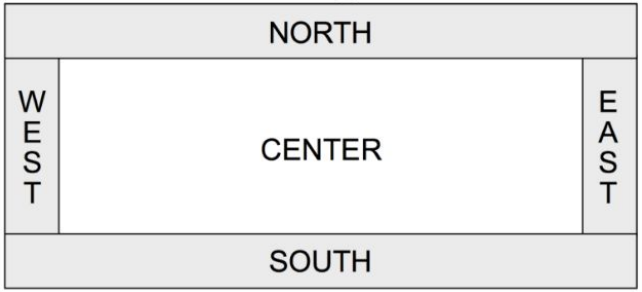
- HashMaps
- Classes
- Interactors
- Closing Remarks

Review: Interactors



Review: Interactors

Interactors can be placed in 5 regions on the screen.



- The center is usually where things happen!
  - The ConsoleProgram adds the Console there.
  - The GraphicsProgram add the Canvas there.
- We only see the other regions of the screen if we add interactors there using `add(component, REGION)`
- Interactors are automatically centered in their region.

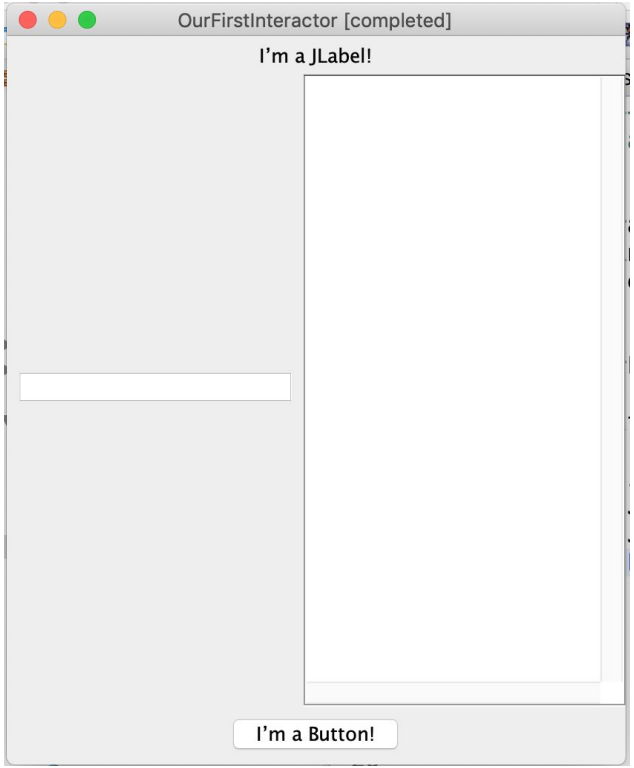
Building interactors in your program

1. Add interactors in `init()` in order
2. `addActionListeners()` to listen for button presses
3. `.addActionListener(this)` on text fields for ENTER
   - Plus (usually) setActionCommand(command)
4. Implement `actionPerformed`
5. Java will call `actionPerformed` whenever an action event occurs

Review: Our First Interactor

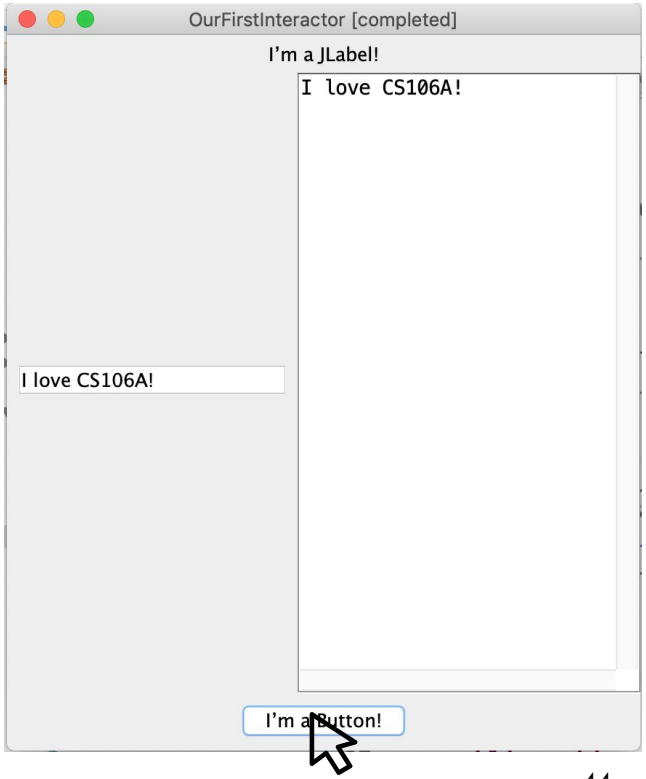
```
import javax.swing.*;
import java.awt.event*;

public class ourFirstInteractor extends ConsoleProgram {

     private JTextField textField = new JTextField(15);

     public void init(){
          add(new JLabel("I'm a JLabel!"), NORTH);
          add(new JButton("I'm a Button!"), SOUTH);
          add(textField, WEST);
          addActionListeners();
     }

}
```

In order to detect actions in these fields, we must addActionListeners()

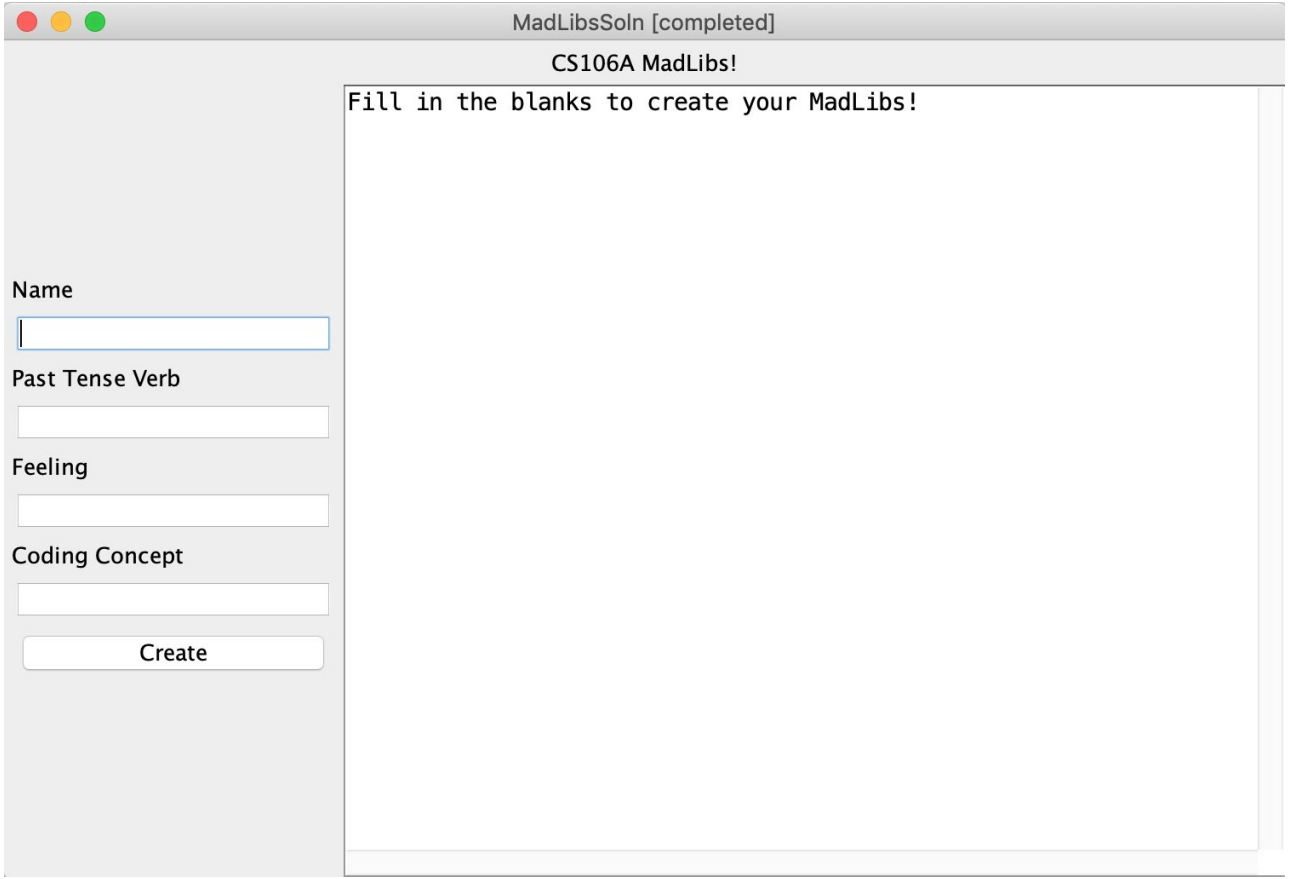


Our First Interactor

```
import javax.swing.*;
import java.awt.event*;

public class ourFirstInteractor extends ConsoleProgram {

     ... // added JComponents, etc here

     public void actionPerformed(ActionEvent e){
          // If we click the button, this will trigger
          String text = textField.getText();
          println(text);


     }
}
```



MadLibs


MadLibsSoln [completed]
CS106A MadLibs!
Fill in the blanks to create your MadLibs!
Name
Past Tense Verb
Feeling
Coding Concept
Create

MadLibs Code

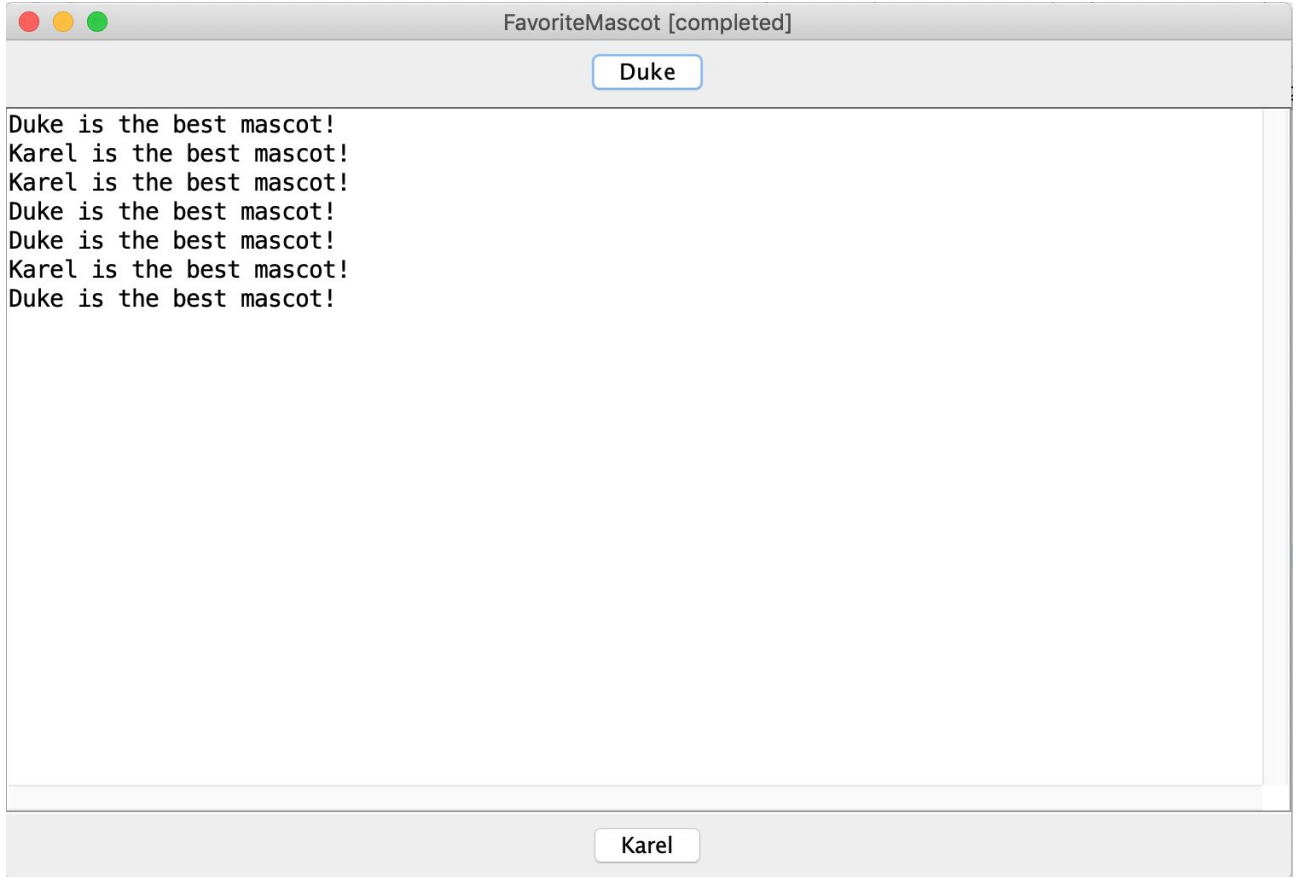
```
public class MadLibsSoln extends ConsoleProgram {

    private JTextField nameField = new JTextField(15);
    private JTextField verbField = new JTextField(15);
    private JTextField feelingField = new JTextField(15);
    private JTextField codeField = new JTextField(15);

    public void init() {
        add(new JLabel("CS106A MadLibs!"), NORTH);
        add(new JLabel("Name"), WEST);
        add(nameField, WEST);
        add(new JLabel("Past Tense Verb"), WEST);
        add(verbField, WEST);
        add(new JLabel("Feeling"), WEST);
        add(feelingField, WEST);
        add(new JLabel("Coding Concept"), WEST);
        add(codeField, WEST);
        add(new JButton("Create"), WEST);
        addActionListeners();
    }

    public void run(){
        println("Fill in the blanks to create your MadLibs! \n");
    }

    public void actionPerformed(ActionEvent e) {
        String name = nameField.getText();
        String verb = verbField.getText();
        String feeling = feelingField.getText();
        String code = codeField.getText();

        createMadLib(name, verb, feeling, code);
    }

    private void createMadLib(String name, String verb, String feeling, String code) {
        println("On " + name + "'s first day of CS106A they accidentally woke up");
        println("late for class! They " + verb + " to class, but when " + name);
        println("arrived at Bishop Auditorium, no one was there! " + name + " was ");
        println(feeling + ". Well, good thing no one uses " + code + " anyways.");
    }
}
```

Review: actionPerformed

Method	Description
`e.getActionCommand()`	a text description of the event *(e.g., the text of the button clicked)*
`e.getSource()`	the interactor that generated the event

```
public void actionPerformed(ActionEvent e){

     String command = e.getActionCommand();
     if(command.equals("Button 1")){
          println("Button 1 was pressed");
     } else if (command.equals("Button 2")){
          println("Button 2 was pressed");
     }

}
```

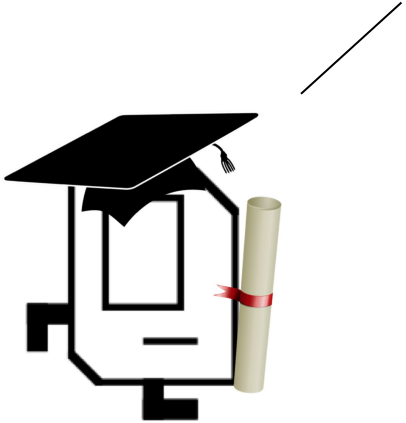
Two Button Example



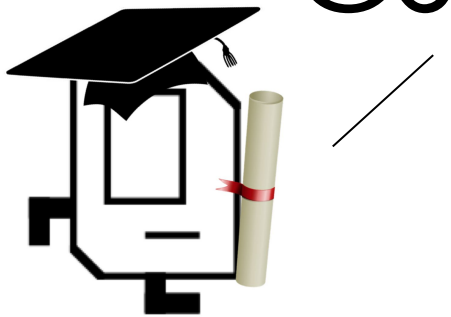
Two Button Example Code

```
public class FavoriteMascot extends ConsoleProgram {

    private JButton duke = new JButton("Duke");
    private JButton karel = new JButton("Karel");

    public void init() {
        add(duke, NORTH);
        add(karel, SOUTH);
        addActionListeners();
    }

    public void actionPerformed(ActionEvent e){
        if(e.getSource() == duke){
            println("Duke is the best mascot!");
        } else if (e.getSource() == karel){
            println("Karel is the best mascot!");
        }
    }

}
```

Anyone can be a computer scientist!



You don't have to be a Computer Scientist to use CS to solve problems!



We've enjoyed learning with you and can't wait to see what you do next!

Last ever CS106A slide in Java…!

Thank you for a great quarter!

Really last slide

Hehe =D