Software Design Patterns: Session One

Sahar Mostafa
UC Berkeley Extension
Spring 2020

Ice Breaker

Hi, my name _____ , what I do _____, interesting fact about me _____, my goal out of this course _____

Welcome to the Software Design Patterns Course

Best practices for building software

Design Patterns

Cloud Services

Our Goals

Resources

Canvas

https://onlinelearning.berkeley.edu/login/canvas

Books

Design Patterns: Elements of Reusable Object-Oriented Software 1st Edition

Elemental Design Patterns 1st Edition Optional

GitHub

https://github.com/SMostaf/COMPSCIX418.2

Online Resources

https://learning.oreilly.com/library/view/elemental-design-patterns/9780321712554/

Sahar Mostafa

Part Time Instructor UC Berkeley Extension

sahar.mostafa@berkeley.edu

https://www.linkedin.com/in/saharmostafa/

https://github.com/SMostaf/COMPSCIX418.2

Logistics

- **Tools Used**

  Canvas Course website

  GitHub

  IDE

- **Programming Languages**

  Python

  Java

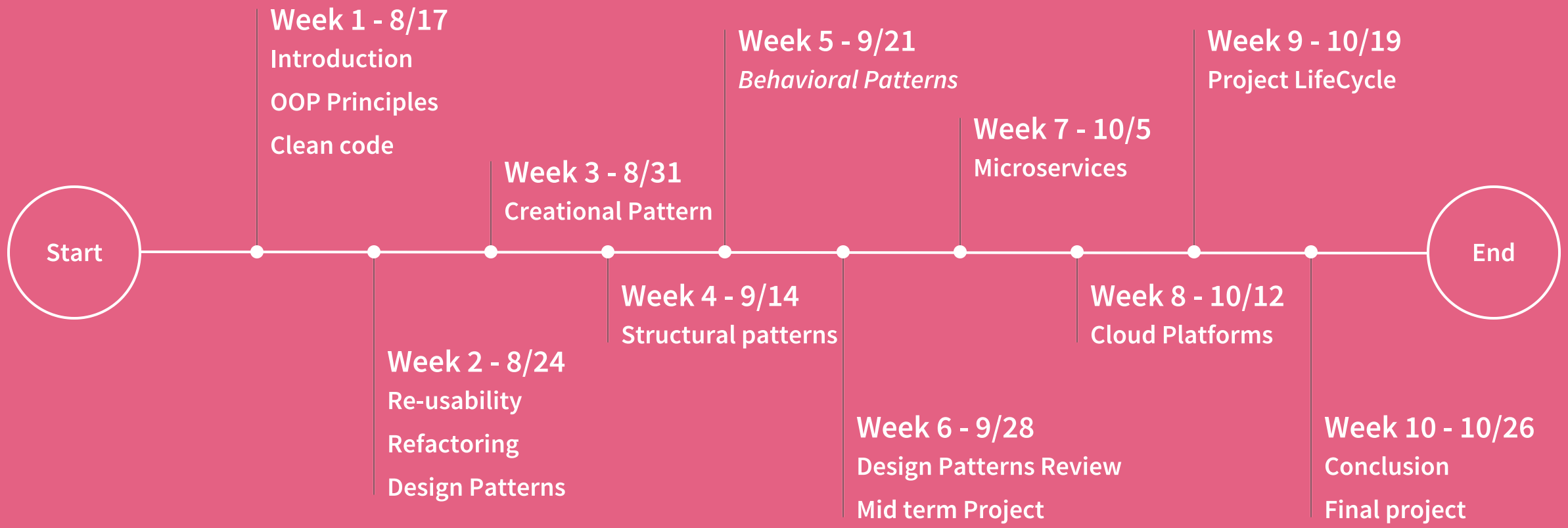
  C++

  Swift

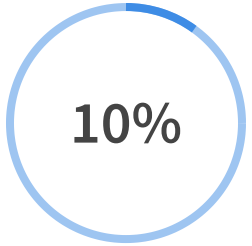
Group Question

Languages Poll

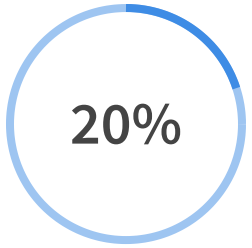
Syllabus and Timeline



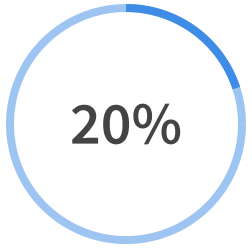
Grades



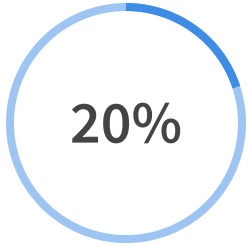
In class participation



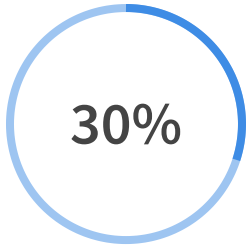
Group Assignments



Individual Assignments



Mid-term Project



Final Project

History

Design Patterns Emergence with publication Design patterns: Elements of Reusable Object Oriented Software in 1995

Gang Of Four GoF: all research and academic collection of wisdom

Tribal Wisdom and Collective Experience

Unselfconscious: design copied repeatedly - rarely changes

Self-conscious: free style on every turn with a set of variable aesthetics - architectural freedom

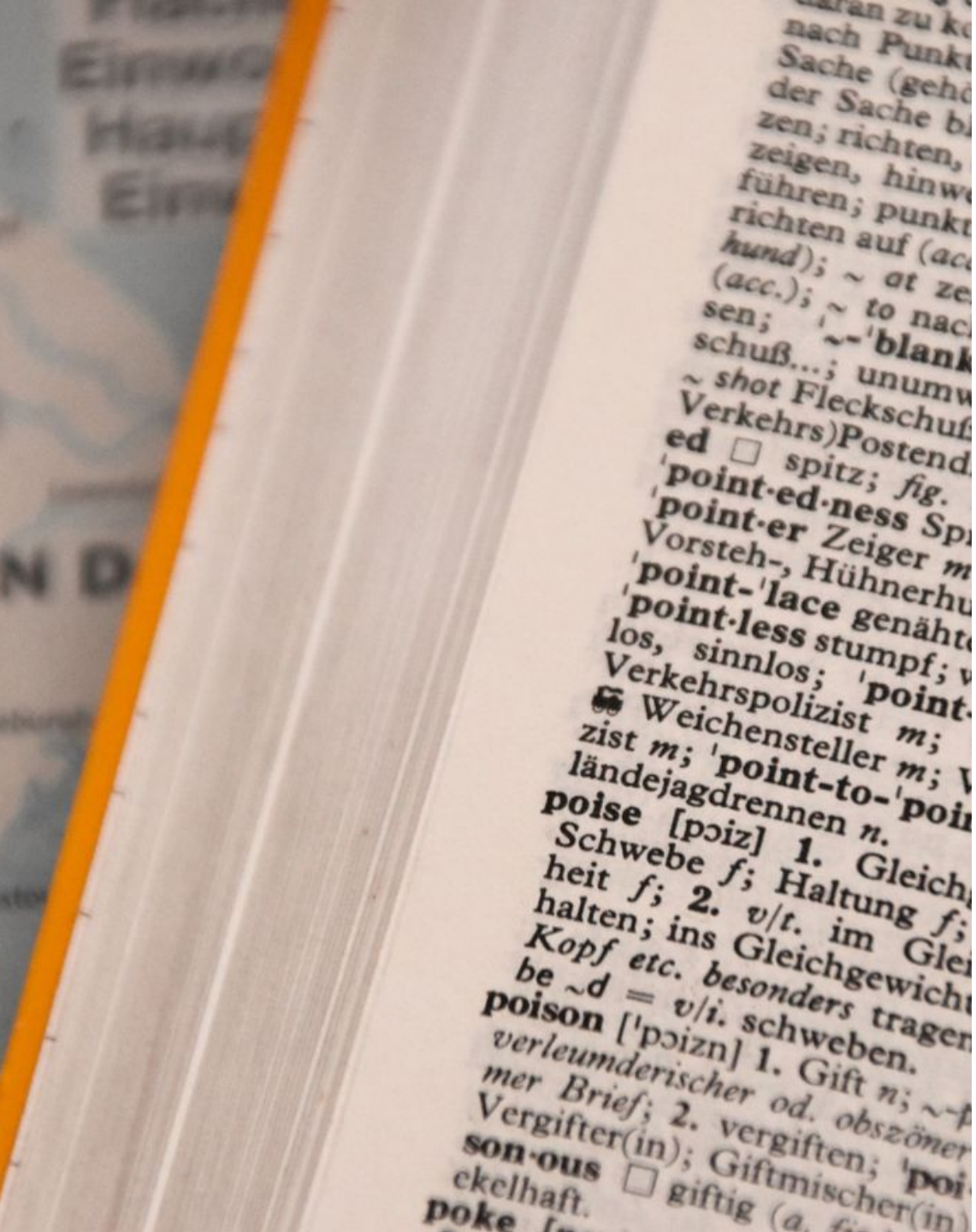
Why do we need Software Patterns?

A common solution to a recurring problem within a particular context.

Language independent concepts

- **Malignant**
- **AntiPattern**
- **Iatrogenic**

Mix between art and programming



Definition

An easy way to follow with a structure that naturally embraces the problem

Envision how code can be extended with a future outlook

Is it Art or Science?

Art since it is a combination of a set of aesthetics at the of the free will of the architect

Science from the analytics side of the problem, collect lessons learned and find a solution to a recurring problem

Patterns as concepts are the foundation of a science

Object Oriented Programming - A PIE

- **Abstraction**
  Common behavior
- **Inheritance**
  Extend behavior
- **Polymorphism**
  Change behavior according to type
- **Encapsulation**
  Information hiding

What?

Extract relevant data to expose to the caller upper layers

Why?

Avoid redundant declaration of behavior

Avoid sharing the internals of a system

How?

Look at the system, break it down to a set of default behavior

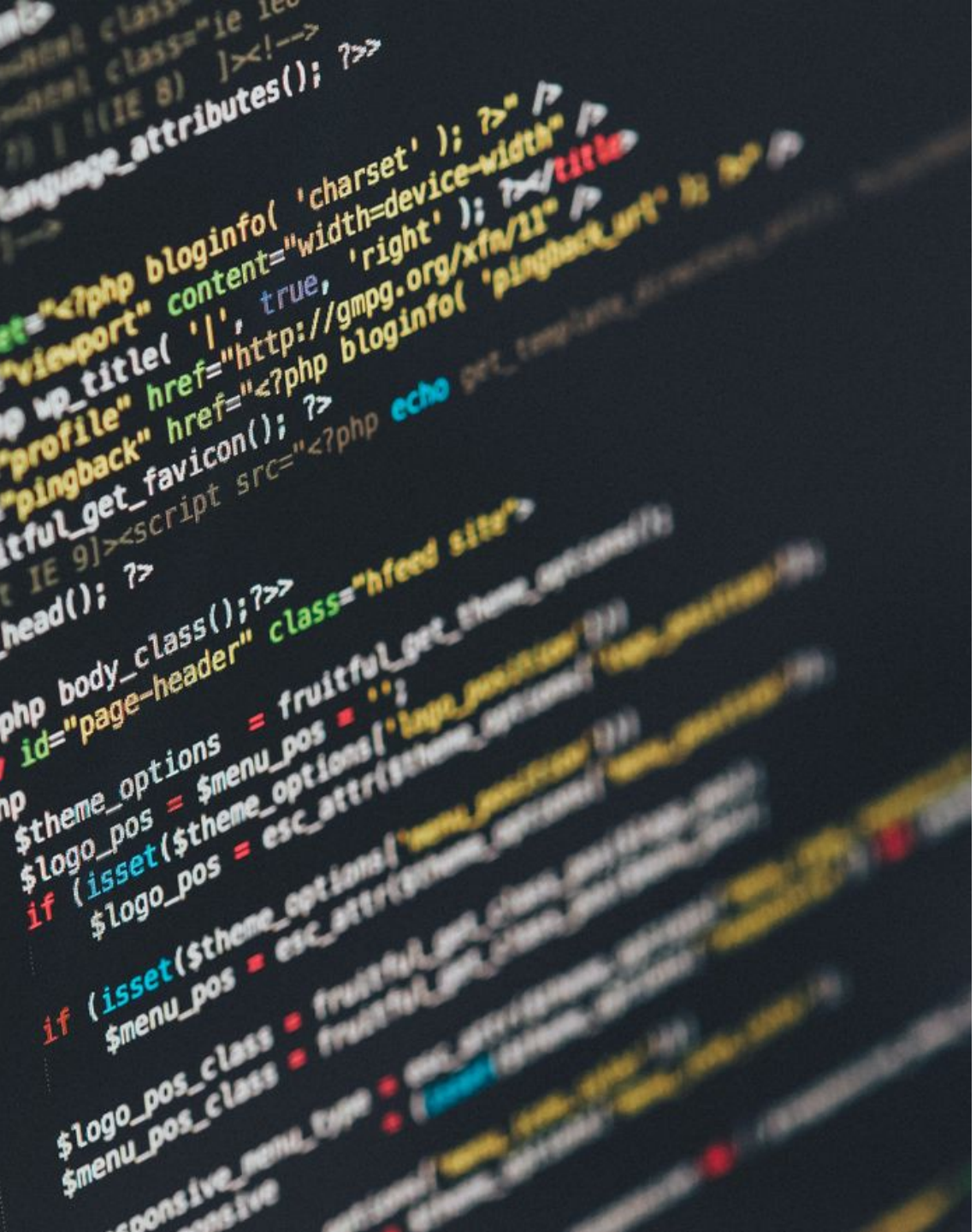
Abstraction

Use case

List declaration: an abstraction of a sequence of items hiding the details

Example: ArrayList vs LinkedList

Computer internals: to the client side we only care about the interfaces the computer shows, not the CPU utilizations, memory handling or disk space.



Example

```
abstract class Animal {
public abstract void sound();
}
```


15 mins Break

What?

Binding the state of the data with the code that manipulates it

Caller interacts with the object using the object's methods

Why?

Encapsulation keeps the data and the code safe from external interference

How?

Private variables Getter/Setter functions

Class abstracts away the implementation details related to the object's state.

Namespaces and packages are scopes
Methods and functions are scopes for the local variables declared within them

Encapsulation

Use Case

Access Modifiers

Public: access is public, any external entity can access the data or member function

Protected: access is protected, internal functions, classes and subclasses can access the data or function.

Private: access is private, member functions and inner classes can access the data or function

	default	private	protected	public
Same Class	Yes	Yes	Yes	Yes
Same package subclass	Yes	No	Yes	Yes
Same package non-subclass	Yes	No	Yes	Yes
Different package subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

https://www.geeksforgeeks.org/access-modifiers-java/

Access Modifiers

Class	Package	Subclass (same package)	Subclass (diff package)	Outside Class
public	Yes	Yes	Yes	Yes	Yes
protected	Yes	Yes	Yes	Yes	No
default	Yes	Yes	Yes	No	No
private	Yes	No	No	No	No

Example

```
//save by ClassA.java
package packageA;

public class A {

protected void printHello() {

  System.out.println("Hello");

 }
}
```

```
//save by B.java
package packageB;
import packageA.*;

class B extends A {

public static void main(String args[]) {
 B obj = new B();
 obj.printHello();
}
}
```

Private vs Protected vs Public

Design a class called Shape representing geometric shapes

Consider which methods can be private vs public and which methods can be overridden in the subclass

Consider a Rectangle class

Activity


15 mins Break

What?

isA relationship between a subclass and a super

classType relation

Why?

Objects using inheritance can extend a common behavior and build on top

With one common behavior, the code is well structured

How?

Define common behavior among objects

Drive a new class from an existing one

```
class Shape {
    protected int xPos;
    protected int yPos;
public Shape(int xPos, int yPos) {
        this.xPos = xPos;
        this.yPos = yPos;
    }
    public void draw() {
    }
}

class Circle extends Shape {
    private int radius;

    public void draw(int radius) {
     this.radius = radius;
   }
}

class Square extends Shape {
    private int length;
    public void draw(int length) {
        this.length = length;
    } }
```

What?

Objects have different behaviors based on their type

Why?

Allow objects to have different internal structures to share the same external interface

Same method to external callers

How?

Dynamic: subclass superclass

Override a method of its superclass providing different functionality

Compile time: method overloading

Run time: method overriding

Multiple methods within the same class that use the same name but a different set of parameters

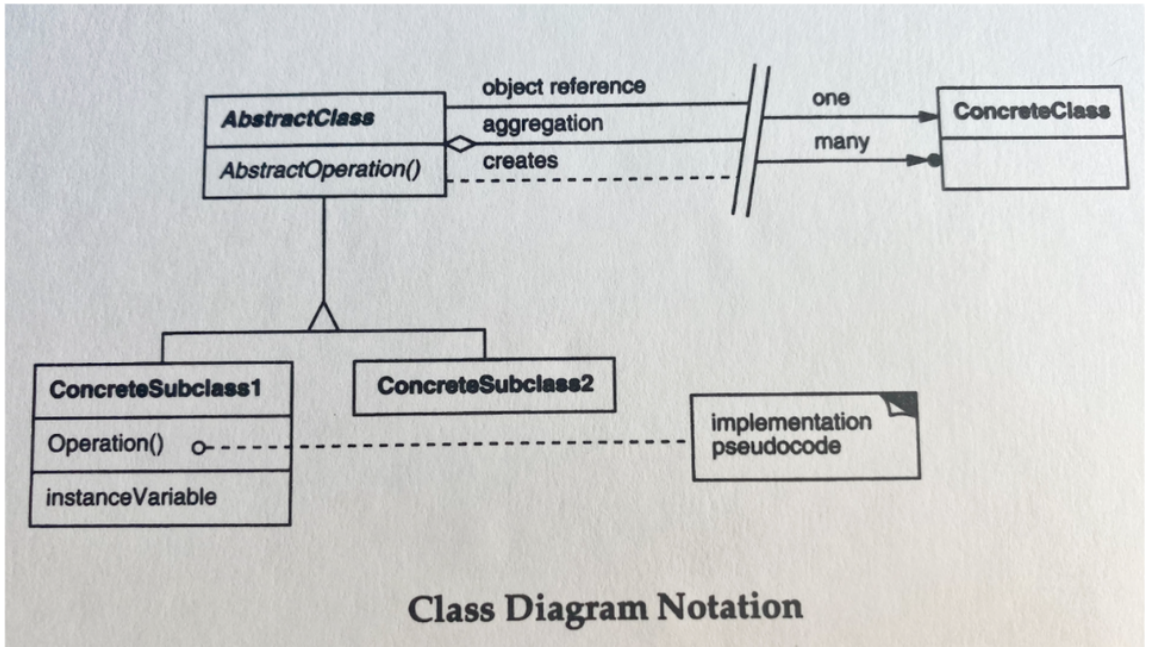
Polymorphism

Use Case

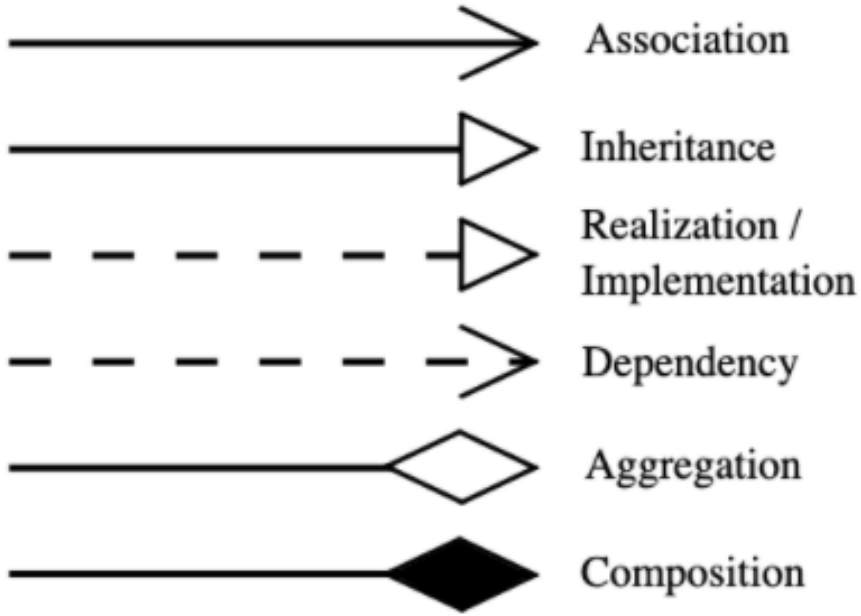
```
public class Drawing {
    public static void main(String args[])
    {
        Shape shape1 = new Circle(10);
        shape1.draw();

        Shape shape2 = new Square(10);
        shape2.draw();
    }
}
```

```
class Shape {
    protected int xPos;
    protected int yPos;
public Shape(int xPos, int yPos) {
        this.xPos = xPos;
        this.yPos = yPos;
    }
    public void draw() {
    }
}

class Circle extends Shape {
    private int radius;

    public void draw(int radius) {
     this.radius = radius;
   }
}

class Square extends Shape {
    private int length;
    public void draw(int length) {
        this.length = length;
    } }
```

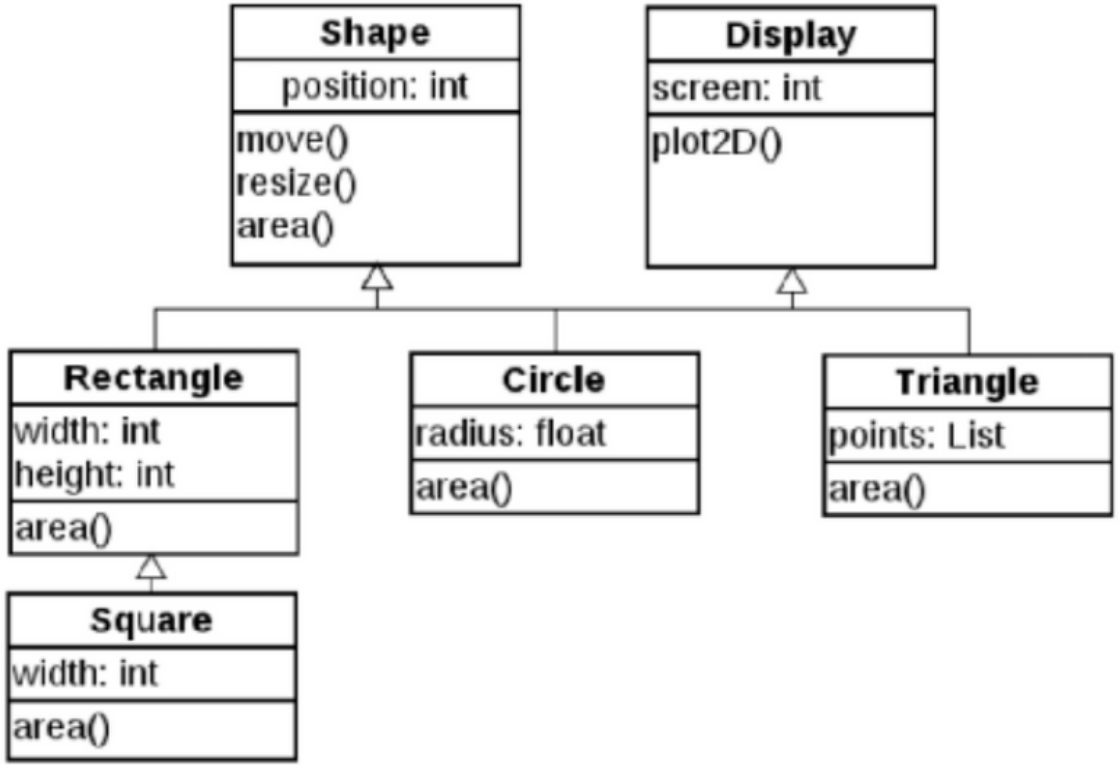
UML Diagram and Notation



Class Diagram Notation



Example



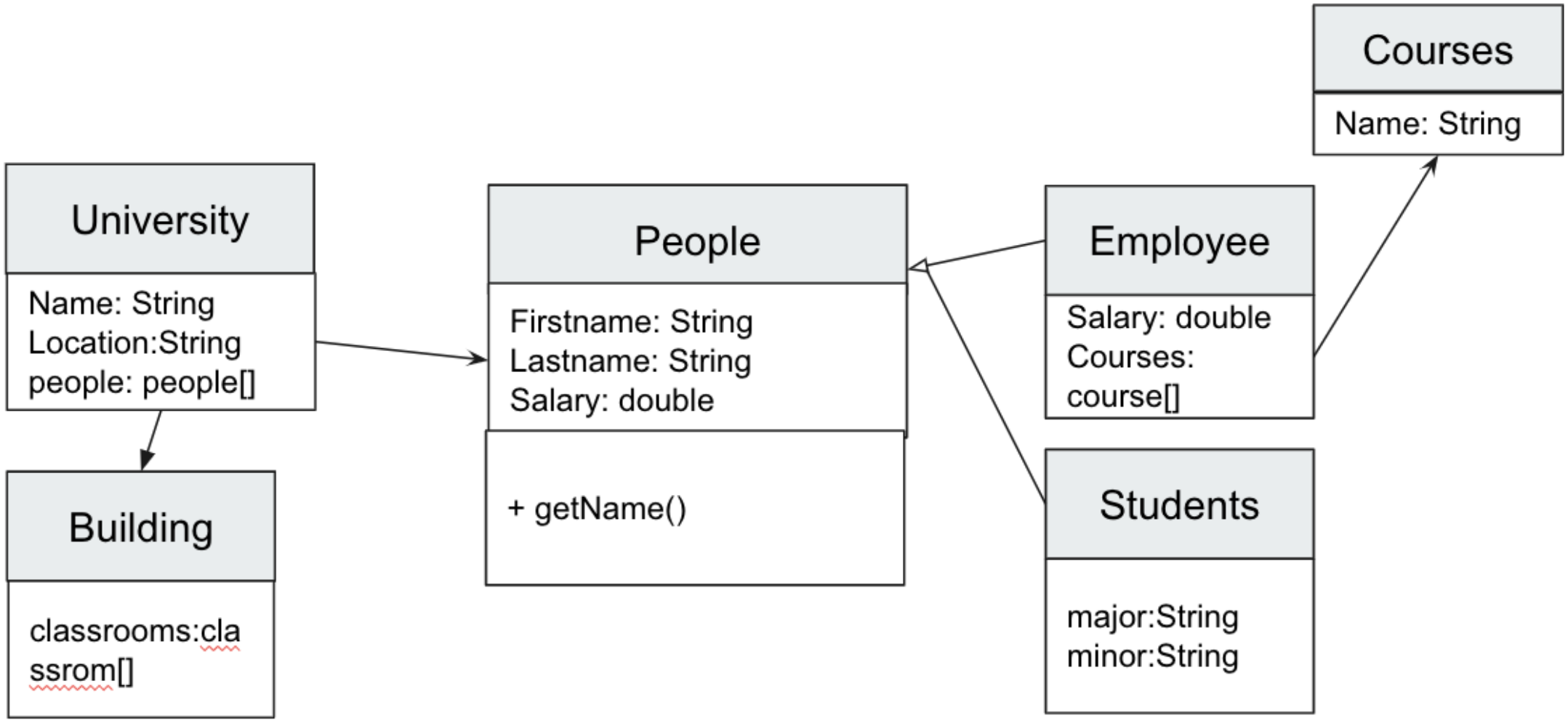
isA vs hasA exercise

Design a University System with building, classrooms, courses, students and teachers

Define relationship between each entity

Provide the UML Diagram

Activity


Courses
Name: String
University
Name: String
Location:String
people: people[]
People
Firstname: String
Lastname: String
Salary: double
+ getName()
Employee
Salary: double
Courses:
course[]
Building
classrooms:cla
ssrom[]
Students
major:String
minor:String


15 mins Break

Code Review

```
if (i % 3 == 0) {
    if (j % 3 == 0) {
        // i+1,j+1 i+1,j+2 i+2,j+1 i+2,j+2
        if (fullArray[i + 1][j + 1] != temp && fullArray[i + 1][j + 2] != temp && fullArray[i + 2][j + 1] != temp && fullArray[i + 2][j + 2] != temp) {
            bg = 1;
        }
    } else if (j % 3 == 2) {
        // i+1,j-1 i+1,j-2 i+2,j-1 i+2,j-2
        if (fullArray[i + 1][j - 1] != temp && fullArray[i + 1][j - 2] != temp && fullArray[i + 2][j - 1] != temp && fullArray[i + 2][j - 2] != temp) {
            bg = 1;
        }
    } else {
        // i+1,j+1 i+1,j-1 i+2,j+1 i+2,j-1
        if (fullArray[i + 1][j + 1] != temp && fullArray[i + 1][j - 1] != temp && fullArray[i + 2][j + 1] != temp && fullArray[i + 2][j - 1] != temp) {
            bg = 1;
        }
    }
    // check only above

} else if (i % 3 == 2) {
    if (j % 3 == 0) {
        // i-1,j+1 i-1,j+2 i-2,j+1 i-2,j+2
        if (fullArray[i - 1][j + 1] != temp && fullArray[i - 1][j + 2] != temp && fullArray[i - 2][j + 1] != temp && fullArray[i - 2][j + 2] != temp) {
            bg = 1;
        }
    } else if (j % 3 == 2) {
        // i+1,j-1 i+1,j-2 i+2,j-1 i+2,j-2
        if (fullArray[i - 1][j - 1] != temp && fullArray[i - 1][j - 2] != temp && fullArray[i - 2][j - 1] != temp && fullArray[i - 2][j - 2] != temp) {
            bg = 1;
        }
    } else {
        // i+1,j+1 i+1,j-1 i+2,j+1 i+2,j-1
        if (fullArray[i - 1][j + 1] != temp && fullArray[i - 1][j - 1] != temp && fullArray[i - 2][j + 1] != temp && fullArray[i - 2][j - 1] != temp) {
            bg = 1;
        }
    }
    // check only below
} else {
    if (j % 3 == 0) {
        // i-1,j+1 i-1,j+2 i-2,j+1 i-2,j+2
        if (fullArray[i + 1][j + 1] != temp && fullArray[i - 1][j + 1] != temp && fullArray[i + 1][j + 2] != temp && fullArray[i - 1][j + 2] != temp) {
            bg = 1;
        }
    } else if (j % 3 == 2) {
        // i+1,j-1 i+1,j-2 i+2,j-1 i+2,j-2
        if (fullArray[i + 1][j - 1] != temp && fullArray[i - 1][j - 1] != temp && fullArray[i + 1][j - 2] != temp && fullArray[i - 1][j - 2] != temp) {
            bg = 1;
        }
    } else {
        // i+1,j+1 i+1,j-1 i+2,j+1 i+2,j-1
        if (fullArray[i + 1][j + 1] != temp && fullArray[i + 1][j - 1] != temp && fullArray[i - 1][j - 1] != temp && fullArray[i - 1][j + 1] != temp) {
```

Code Review Best Practices

Code Review Process walk through: follows guidelines, covered **by unit tests**

Define the **goal** of the classes in the Pull Request

Consistency and easy to read

Look for repeated code

Any opportunity to abstract any common logic

Exception handling, will the system halt?

Can the newly introduced class be **extended** in the future

Summary

1. What is a software pattern?
2. Why do we need a pattern?
3. How can we apply a pattern?
4. OOP Principles and Class Diagrams

Session Two

Next Week

Next Week

1. Review Session One
2. Re-usability and Refactoring
3. Software Design Patterns Types
4. In class exercises


O'REILLY
Designing Data-Intensive Applications
THE BIG IDEAS BEHIND RELIABLE, SCALABLE, AND MAINTAINABLE SYSTEMS
Martin Kleppmann
85th ANNIVERSARY EDITION
THE
ART SPIRIT
ROBERT HENRI