OOP:
Objects and Classes

- Foundation of Object-Oriented Programming (OOP)
- Covers:
  - Defining classes and creating objects
  - Constructors
  - Instance vs static members
  - The this keyword
  - Data encapsulation

What is OOP?

- OOP stands for Object-Oriented Programming.
- Object-oriented programming has several advantages over procedural programming:
  - OOP is faster and easier to execute
  - OOP provides a clear structure for the programs
  - OOP helps to keep the Java code DRY "Don't Repeat Yourself", and makes the code easier to maintain, modify and debug
  - OOP makes it possible to create full reusable applications with less code and shorter development time

Object-Oriented Programming (OOP)

- Beyond procedural programming: Focus on "objects" rather than just actions.
- Models real-world entities.
- Enables complex system development.

What is a Class?

- A blueprint or template for creating objects.
- Defines the common structure (data fields) and behaviors (methods) for all objects of that type.
- Think of it like a cookie cutter, and objects are the cookies.
- Defines:
  - Fields (attributes/data)
  - Methods (behavior/functions)
- Example:

```
public class Circle {
double radius;
double getArea() {
    return radius * radius * Math.PI;
}
}
```

What is an Object?

- An object represents an entity with a unique identity, state, and behaviors.
  - Identity: Distinguishes it from other objects.
  - State (Attributes/Data Fields): Properties with current values (e.g., a circle's radius).
  - Behaviors (Methods): Actions the object can perform (e.g., a circle calculating its area).
- An object is an instance of a class
- Memory is allocated when object is created
- Can access fields and methods of the class

```
Circle c1 = new Circle();
c1.radius = 5.0;
System.out.println(c1.getArea());
```

OO Programming Concepts

- Object-oriented programming (OOP) involves programming using objects.
- An object represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, and even a loan can all be viewed as objects.
- An object has a unique identity, state, and behaviors. –
  - The state of an object consists of a set of data fields (also known as properties) with their current values. –
  - The behavior of an object is defined by a set of methods.

Creating Objects

- Syntax:

  ClassName objectName = new ClassName();
- new keyword creates the object
- Constructor initializes it

Accessing Object Members

- Use the dot operator (.) to access an object's data fields and methods.
  objectName.field
  objectName.method()
- Example:
  c1.radius = 10;
  double a = c1.getArea();

Constructors

- Special methods used to construct and initialize objects when they are created.
- Same name as class, no return type
- Automatically called when object is created (new operator)

Constructors

- Default Constructor
  - A constructor with no parameters
  - Java provides one if no constructor is written
- No-Arg Constructor
  - A constructor that takes no arguments.
  - If you don't define any constructors, Java provides a default no-arg constructor.
  - Example: public Circle() { radius = 1.0; }
- Parameterized Constructors :
  - Constructors that take arguments to initialize data fields.
    Example: public Circle(double newRadius) { radius = newRadius; }
- Overloaded Constructors : A class can have multiple constructors, as long as they have different parameter lists (different number of parameters, different types, or different order of types).

Constructor Example

```
public class Circle {
  double radius;
  public Circle(double r) {
    radius = r;
  }
}
Circle c = new Circle(3.5);
```

Primitive vs. Reference Types Revisited

- Primitive type variables: Store the actual value.
- Reference type variables: Store the memory address (reference) of an object.
- Assignment (=) for primitives copies the value; for references, it copies the reference, making both variables point to the same object.

Static Variables and Methods

- **Static (or class) variables**: Belong to the class itself, not to specific objects. All objects of the class share the same static variable.
- **Static methods**: Belong to the class and can be invoked without creating an object. They can only access static data fields and other static methods.
- Use ClassName.staticMember for clarity.

Instance Variables and Methods

- Declared without static
- Belong to each object individually
- Example:

```
public class Person {
String name;
void sayHello()
{
  System.out.println("Hello, " + name);
}
}
```

Instance vs. Static

- **Instance members**: Belong to an instance (object) of the class. Each object has its own copy of instance data fields. Instance methods operate on the object's data.
- **Static members**: Shared among all instances of the class.

The this Keyword

- Refers to the current object
- Used to distinguish fields from parameters
- Example:

```
public class Circle {
  double radius; // this is the instance variable
  public Circle(double radius) {
    this.radius = radius;
  }
}
```

Data Fields

- Also called attributes or instance variables
- Define the properties of an object
- Can be private, public, or protected
  - private: accessible only within the class
  - public: accessible from any class
  - protected: accessible within the same package and subclasses

Methods

- Define behavior of objects
- Can return a value or be void
- Can have parameters or no parameters
- Access Modifiers on Methods:
  - public: method accessible from any class
  - private: method accessible only within the same class
  - protected: method accessible within the same package and subclasses
- Example:

```
public double getArea() {
return radius * radius * Math.PI;
}
```

Encapsulation

- Encapsulation is one of the four pillars of OOP
- It means bundling data (fields) and methods that operate on that data into a single unit (class)
- Protects internal state by restricting direct access
- Fields are usually declared private
- Access is provided via public getter and setter methods Hiding data from direct access
- Use private fields and public getters/setters

Example of Encapsulation

```
private double radius;
public double getRadius() {
  return radius;
}
public void setRadius(double r) {
  radius = r;
}
```

Benefits of Encapsulation

- Promotes modularity: changes in one class don’t affect others directly
- Makes the code easier to maintain and debug
- Allows restricting access and modifying implementation without affecting users
- Enables validation or access control through getters/setters
- Improves code readability and reuse
- Control access to fields
- Protect object state
- Allows validation logic in setters

Access Modifiers

- private: Accessible only in same class
- public: Accessible everywhere
- protected and package-private discussed later
- Crucial for **encapsulation.**

Static Variables

- Declared with static
- Shared by all objects of the class
- Example :
  static int objectCount = 0;

Static Methods

- Declared with static keyword
- Belong to the class, not a specific object
- Can be called without creating an object
- Cannot access instance fields directly

Static Example

```
public class MathUtil {
  public static int square(int x) {
    return x * x;
  }
}
MathUtil.square(5);
```

Passing Objects to Methods

- Java uses "pass-by-value" for all arguments.
- For **primitive** types, the actual value is copied.
- For **reference** types, the reference (memory address) is copied. This means changes made to the object through the copied reference inside the method will affect the original object.
- **Objects** are passed by **reference**
- Method can **modify** the object fields

```
public static void changeRadius(Circle c) {
  c.radius = 9.0;
}
```

Arrays of Objects

- An **array** can hold **objects** (or rather, references to objects).
- When an array of objects is created, its elements are initially **null**.
- Each element must then be individually instantiated with new.
- Example:

```
Circle[] circles = new Circle[5];
for (int i = 0; i < circles.length; i++) {
 circles[i] = new Circle();
 }
```

Immutable Objects and Classes

- Immutable object: An object whose state cannot be modified after it's created.
- Immutable class: A class whose objects are immutable.
- Benefits: Thread safety, simpler reasoning about program state.
- To make a class immutable:
  - All data fields must be private and final.
  - No setter methods.
  - If mutable objects are used as data fields, they must be defensively copied.

Example: The Point Class

- Consider a simple Point class with x and y coordinates. We want to ensure that once a Point is created, its coordinates never change.

```
public final class ImmutablePoint { // Class is final to prevent subclassing
    private final int x;          // All fields are private and final
    private final int y;

    // Constructor to initialize fields
    public ImmutablePoint(int x, int y) {
        this.x = x;
        this.y = y;
    }
    // Only getter methods (no setters)
    public int getX() { return x; }
    public int getY() { return y; }
    // No way to change x or y after object creation
}
```

```
// Usage Example:
ImmutablePoint p1 = new ImmutablePoint(10, 20);
// p1.x = 30; // Compile-time error: cannot assign to final field
// To "change" a point, create a new one:
ImmutablePoint p2 = new ImmutablePoint(30, 40);
```

Object Composition

- A class can have fields of other class types
- Called "has-a" relationship
- Example:
  class Engine { }

  class Car {

    Engine e = new Engine();

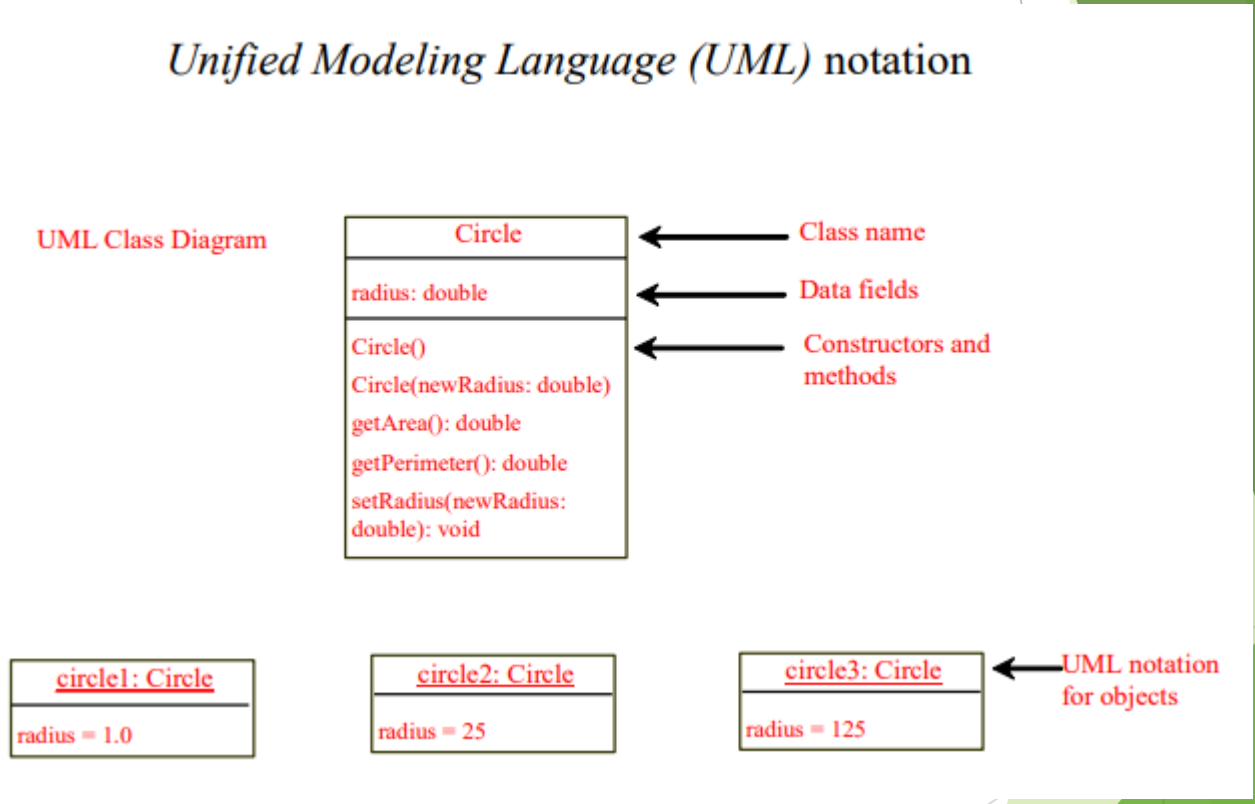
  }

UML Class Diagrams

- A visual way to describe classes
- Syntax:
  + public
  - private
  # protected

Example UML for Circle

```
+---------------+
|    Circle     |
+---------------+
| - radius: double |
+---------------+
| + getArea(): double |
| + setRadius(r: double): void |
+---------------+
```



Wrapper Classes (Intro)

- Wrapper classes allow primitive data types to be used as objects
- Each primitive has a corresponding wrapper:
  - int → Integer
  - double → Double
  - Useful with collections (explained later)
- Useful when working with Java Collections like ArrayList, which only store objects
- Enable methods and features available to objects such as .toString(), .compareTo(), etc.
- Provide utility methods for parsing and conversion (e.g., Integer.parseInt())

Using Wrapper Classes

```
Integer num = Integer.valueOf(5);
int n = num.intValue();
```

Wrapper Classes

Primitive	Wrapper
int	Integer
double	Double
char	Character
boolean	Boolean
byte	Byte
short	Short
long	Long
float	Float

Autoboxing/Unboxing

- Autoboxing: automatic conversion from a primitive type to its corresponding wrapper class (e.g., int → Integer)
- Unboxing: automatic conversion from a wrapper object back to its primitive type (e.g., Integer → int)
- Happens implicitly in assignments, method calls, and expressions
- Useful when storing primitives in collections like ArrayList<Integer>
- Example:
  - int a = 10;
  - Integer obj = a; // autoboxing
  - int b = obj; // unboxing
  - Autoboxing: int x = 5; Integer obj = x;
  - Unboxing: Integer obj = 7; int y = obj;
  - Happens automatically

The toString() Method

- Every object inherits from Object class

- toString() returns string representation
- Example:
  public String toString() {
  return "Circle with radius " + radius;
  }

Summary of Chapter 9

- Defined and created classes and objects
- Constructors and overloading
- Used this keyword
- Data encapsulation and accessors
- Instance vs static members
- Wrapper classes and toString()

Chapter 9 – Test Questions

- What is the correct way to create an object of a class named Student?
- A) Student = new Student();
- B) Student student = Student();
- C) Student student = new Student();
- D) new Student = Student();

Chapter 9 – Test Questions

- What does the keyword this refer to in a Java class?
- A) The name of the class
- B) The current object
- C) The parent class
- D) A static reference

Chapter 9 – Test Questions

- Which of the following correctly declares a constructor in Java?
- A) void Constructor() { }
- B) Student.Student() { }
- C) Student() { }
- D) public void Student()

Chapter 9 – Test Questions

- What is the purpose of a setter method?
- A) To print object data
- B) To create a new object
- C) To change the value of a private field
- D) To access static variables

Chapter 9 – Test Questions

- In Java, if a class has no constructor, what happens?
- A) Compilation fails
- B) A default constructor is added automatically
- C) Java throws an exception
- D) The class cannot be instantiated

Chapter 9 – Test Questions

Given :

```
public class Circle {
        double radius;
        Circle(double r) { radius = r; }
}
```

What does the following do?

```
Circle c = new Circle(5.0);
```

- A) Declares a variable only
- B) Initializes radius to 0
- C) Creates a Circle object with radius 5.0
- D) Syntax error

Chapter 9 – Test Questions

- Which of the following is true about instance variables?

A) They are shared by all objects

B) They are declared inside methods

C) Each object has its own copy

D) They must be static

Chapter 9 – Test Questions

What will be the output?

```
public class Test {
  int x = 5;
  public static void main(String[] args) {
    Test obj1 = new Test();
    Test obj2 = obj1;
    obj2.x = 10;
    System.out.println(obj1.x);
  }
}
```

A) 5

B) 10

C) 0

D) Compilation error

Chapter 9 – Test Questions

- What is information hiding in Java?

- A) Using this

- B) Declaring variables as public

- C) Using private fields with getters and setters

- D) Overriding methods

Chapter 9 – Test Questions

- A UML diagram shows:

- A) Object code only

- B) Class structure and relationships

- C) Variable values

- D) Loops and decisions

Short Answer

- Write a constructor for a class Book with fields title and price.
- Define a getter and setter for a private field balance in a class Account.
- What is the difference between static and instance variables?
- Explain why we use private fields with public setters/getters.
- What happens if two object references point to the same object and one changes a value?
- Create a class Rectangle with two fields: width and height, and a method getArea().
- How is this() used inside constructors?
- What does the term encapsulation mean in object-oriented programming?
- Describe the structure of a UML class diagram.
- Explain what happens when an object is created using new.