

# CS105

## Introduction to Object-Oriented Programming

**Prof. Dr. Nizamettin AYDIN**

**naydin@itu.edu.tr**

**nizamettin.aydin@ozyegin.edu.tr**

# **Abstract Classes and Interfaces**

# Outline

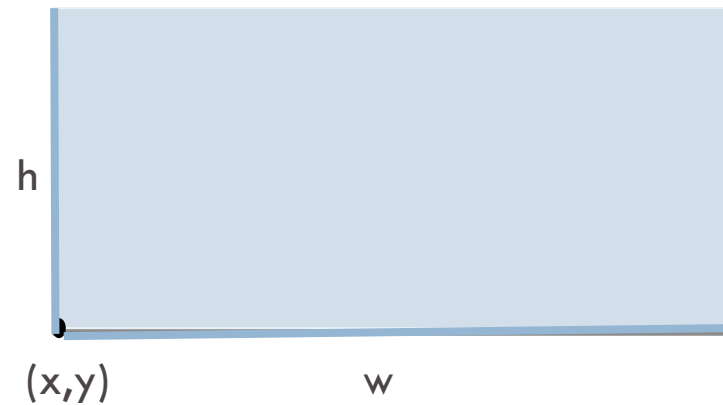
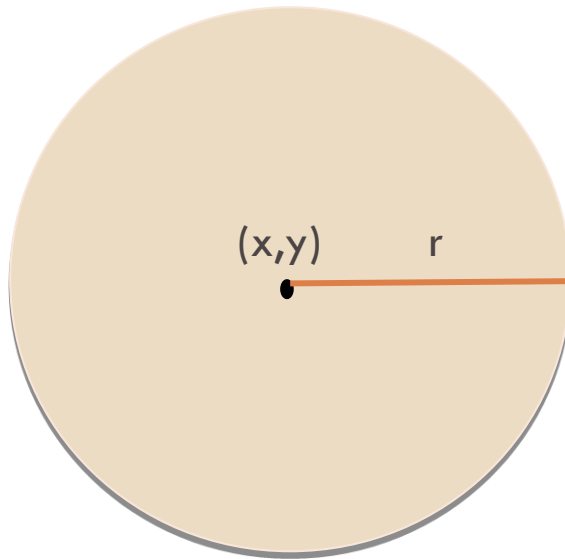
- Shapes
- Inheritance
- Abstract Classes
- Abstract and Concrete Classes
- Abstract Functions
- Interface
- Adapter Class
- instanceof operator

# Shapes

- Let's implement classes for shapes
  - Rectangle
  - Circle
  - etc.
- What is common in all these shapes?
  - **x** and **y** coordinates that hints about the location of the shape.

# Shapes

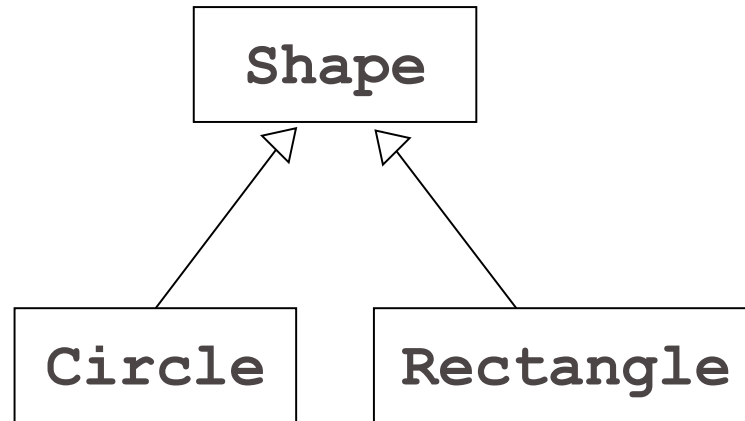
- $(x,y)$  coordinate •



- In circle we hold an additional radius, in rectangle we have height and width.

# Inheritance

- We can have a shape class.
- Other shapes can inherit from the shape class.



- Sometimes a class should define a method that logically belongs in the class, but that class cannot specify how to implement the method.
- For instance:
  - Every shape has an area.
  - Logically, every shape should have a **getArea** method.
  - But ...

- Every shape has an area.
- Logically, every shape should have a **getArea** method.
- But, the area of every shape is calculated differently.
  - Area of Circle =  $\text{pow}(\text{radius}, 2) \times \text{pi}$
  - Area of Rectangle =  $\text{height} \times \text{width}$
- There is not any implementation of **getArea** method in the **Shape** class that is correct for all subclasses of Shape.
  - Therefore, we need to enforce the subclasses of Shape to implement the **getArea** method.



# Abstract Classes

- At this point
  - Every shape has an area.
  - But there is not a possible way to implement the **getArea** method in the **Shape** class.
  - Therefore, maybe we should not let the instantiation of a **Shape** object, even when we have the **Shape** class.
    - **instantiate**: create a new instance
  - Can we?
  - Yes we can, with use of abstract classes.
- Classes that cannot be used to instantiate objects are **abstract classes**.

# Abstract and Concrete Classes

- Classes that cannot be used to instantiate objects are **abstract classes**.
- Classes that can be used to instantiate objects are **concrete classes**.
- **Concrete class** is the default class.

# Abstract Classes

- Classes that cannot be used to instantiate objects are **abstract classes**.
- They are used as **super-classes** during inheritance and provide common attributes and behaviors to its **sub-classes**.

# Shape Class (Concrete)

```
public class Shape {  
    private int x;  
    private int y;  
  
    public Shape (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public int getX() {  
        return x;  
    }  
    public int getY() {  
        return y;  
    }  
}
```

```
public static void main(String[] args) {  
    Shape s = new Shape(0, 1);  
    s.getX();  
}
```

# Shape Class (Abstract)

```
public abstract class Shape {  
    private int x;  
    private int y;  
  
    public Shape (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public int getX() {  
        return x;  
    }  
    public int getY() {  
        return y;  
    }  
}
```

- You make a class **abstract** by declaring it with keyword **abstract**.

# Shape Class (Abstract)

```
public abstract class Shape {  
    private int x;  
    private int y;  
  
    public Shape (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public int getX() {  
        return x;  
    }  
    public int getY() {  
        return y;  
    }  
}
```

```
public static void main(String[] args) {  
  
    Shape s = new Shape(0, 1);  
  
    s.getX();  
  
}
```

✘ Cannot instantiate the type Shape  
Press 'F2' for focus

# Abstract Classes

- Abstract classes are incomplete.
- Their sub-classes can complete these incomplete parts and become concrete classes.
- If they don't, sub-classes will be also abstract.
  
- What do we mean by incomplete?
  - Remember the `getArea()` function.

# Abstract Functions

- A method that has been declared but not implemented is an abstract function.

```
public abstract float getArea();
```

- The keyword **abstract** needs to be used.
- The body of the method is missing.
  - **incomplete function**
- Constructors and static methods cannot be abstract.



# Shape Class (Abstract)

```
public abstract class Shape {
    private int x;
    private int y;

    public abstract float getArea();

    public Shape (int x, int y) {
        this.x = x;
        this.y = y;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
}
```

# Abstract Classes

- A class which contains at least one abstract function is an abstract class and must be declared abstract.
- A class can still be an abstract class even if it does not contain any abstract methods but contain the abstract keyword.
- Concrete classes provide implementations of every method they declare.
- A concrete sub-class needs to implement all the abstract methods inherited from the abstract super-class.

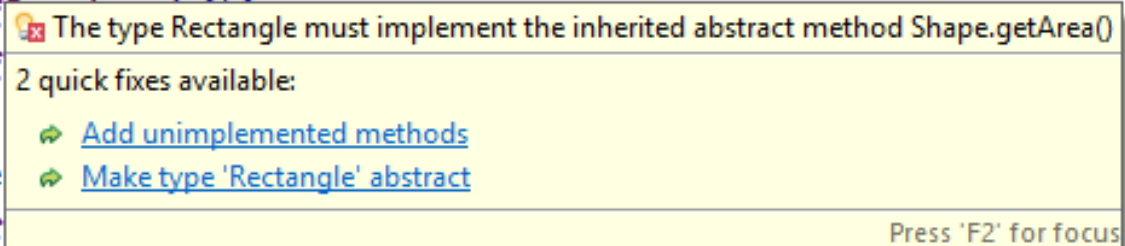
# Abstract Classes

- When inheriting from an abstract class
  - If the sub-class implements all the inherited abstract methods, it can be instantiated
  - If the subclass does **not** implement all the inherited abstract methods, it too must be abstract.

# Circle and Rectangle Classes

- Inheriting from abstract Shape class.

```
public class Rectangle extends Shape{
    private f
    private f
    public Re
        super
        width = w;
        height = h;
    }
}
```



The type Rectangle must implement the inherited abstract method Shape.getArea()  
2 quick fixes available:  
➤ [Add unimplemented methods](#)  
➤ [Make type 'Rectangle' abstract](#)  
Press 'F2' for focus

# Circle and Rectangle Classes

- Inheriting from abstract Shape class.
  - One solution is to make Rectangle class abstract as well

```
public abstract class Rectangle extends Shape{
    private float width;
    private float height;

    public Rectangle (int x, int y, float w, float h)    {
        super(x, y);
        width = w;
        height = h;
    }
}
```

# Circle and Rectangle Classes

- Inheriting from abstract Shape class.
  - The other solution is to implement the `getArea` method.

```
private float width;  
private float height;  
  
public Rectangle (int x, int y, float w, float h)    {  
    super(x, y);  
    width = w;  
    height = h;  
}  
public float getArea()    {  
    return width*height;  
}  
}
```

# Circle and Rectangle Classes

- Inheriting from abstract Shape class.
  - Same for the circle class..

```
public class Circle extends Shape {
    private float radius;

    public Circle (int x, int y, float radius) {
        super(x, y);
        this.radius = radius;
    }
    public float getArea() {
        return radius*radius*3.14f;
    }
}
```

# Using Shapes

```
public class ShapesMain {  
    public static void main(String[] args) {  
  
        Rectangle rect = new Rectangle(0, 10, 10, 5);  
        Circle circ = new Circle(10, 10, 5);  
  
        System.out.println(rect.getArea());  
        System.out.println(circ.getArea());  
    }  
}
```

---

50.0

78.5



# Quick Note

- Not all hierarchies contain abstract classes.
- Not all super-classes needs to be abstract.

# Remember the last class

- We have the following classes:
  - Shape is not abstract

```
public class Rectangle extends Shape{
    private float width;
    private float height;

    public Rectangle (int x, int y, float w, float h)    {
        super(x, y);
        width = w;
        height = h;
    }
    public float getArea()    {
        return width*height;
    }
}
```

```
public class Shape {
    private int x;
    private int y;

    public Shape (int x, int y)    {
        this.x = x;
        this.y = y;
    }
    public int getX()    {
        return x;
    }
    public int getY()    {
        return y;
    }
}
```

```
public static void main(String[] args)    {

    Shape s = new Rectangle(10, 10, 20, 5);
    System.out.println(s.getArea());
}
```

# Remember the last class

- When Shape is abstract, we don't get that compiler error.

–Why?

```
public abstract class Shape {  
    private int x;  
    private int y;  
  
    public abstract float getArea();  
  
    public Shape (int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
    public int getX() {  
        return x;  
    }  
    public int getY() {  
        return y;  
    }  
}
```

```
public static void main(String[] args) {  
  
    Shape s = new Rectangle(10, 10, 20, 5);  
    System.out.println(s.getArea());  
}
```

# Remember the last class

- When Shape is abstract, we don't get that compiler error.
  - Why?
- `getArea` method has been declared in `Shape` class
- Any object that `Shape` can refer to needs to implement this `getArea` method in order to be instantiated.

```
public static void main(String[] args) {  
  
    Shape s = new Rectangle(10, 10, 20, 5);  
    System.out.println(s.getArea());  
}
```

- There are things we cannot do with abstract classes.
- Lets see **interfaces**...

# Interface

- Interfaces offer a capability requiring that unrelated classes implement a set of common methods
- An **interface** only declares the public behaviors of a class but does not implement them.
  - Based on this definition, in an interface
    - All methods are implicitly **public**
    - All methods are implicitly **abstract**
      - There are not any concrete methods
    - There are not any attributes
      - It does not contain any class instance
      - It can contain constants (**final** variables)

# Example interface

- Use the keyword **interface**

```
public interface Shape {  
    public float getArea();  
}
```

- Can we instantiate an interface?
  - No.
- Actually an interface is a very abstract class
  - None of its methods are implemented
  - All methods are abstract

# When do you need an interface?

- You would write an interface when you want classes of various types to all have a certain set of capabilities (behaviors).
  - You can write methods that work for more than one kind of class.
- Very common in GUI implementations.

```
interface KeyListener {  
    public void keyPressed(KeyEvent e) ;  
    public void keyReleased(KeyEvent e) ;  
    public void keyTyped(KeyEvent e) ;  
}
```



# Interface

- A class can **extend** a class.
- A class can **implement** an interface.

```
public interface Shape {  
    public float getArea();  
}
```

```
public class Circle implements Shape {  
    private int x;  
    private int y;  
    private float radius;  
  
    public Circle (int x, int y, float radius) {  
        this.x = x;  
        this.y = y;  
        this.radius = radius;  
    }  
    public float getArea() {  
        return radius*radius*3.14f;  
    }  
    public int getX() {  
        return x;  
    }  
    public int getY() {  
        return y;  
    }  
}
```

# Interface

- A class can only extend one class.
- A class can implement multiple interfaces.
  - This lets the class fill multiple “roles”
  - In writing Applets, it is common to have one class implement several different listeners
- Example:

```
class MyApplet extends Applet
                    implements ActionListener,
KeyListener {
    ...
}
```

- When a class implements an interface, the class needs to implement all the declared methods of the interface.
- If all the declared methods are not implemented, then the class becomes an abstract class.
  - At this point, we need to use the keyword abstract

```
public class Circle implements Shape {
```

```
    private int x;
```

```
    private int y;
```

```
    private float radius;
```

```
    public Circle(int x, int y, float radius) {
```

```
        this.x = x;
```

```
        this.y = y;
```

```
        this.radius = radius;
```

```
    }
```

```
    public int getX() {
```

```
        return x;
```


```
    }
```

```
    public int getY() {
```

```
        return y;
```


```
    }
```

```
}
```

 The type Circle must implement the inherited abstract method Shape.getArea()

2 quick fixes available:

 [Add unimplemented methods](#)

 [Make type 'Circle' abstract](#)

Press 'F2' for focus

```
public abstract class Circle implements Shape {
    private int x;
    private int y;
    private float radius;

    public Circle (int x, int y, float radius) {
        this.x = x;
        this.y = y;
        this.radius = radius;
    }
    public int getX() {
        return x;
    }
    public int getY() {
        return y;
    }
}
```

---

- You can even **extend** an interface (to add methods):

```
public interface ShapeExtended extends Shape {  
    public float getPerimeter();  
}
```

# Interface

- You can even *extend* an interface (to add methods):

```
public interface ShapeExtended extends Shape {  
    public float getPerimeter();  
}
```

```
interface KeyListener {
```

```
    public void keyPressed(KeyEvent e);
```

```
    public void keyReleased(KeyEvent e);
```

```
    public void keyTyped(KeyEvent e);
```

```
}
```

```
interface FunkyKeyListener extends KeyListener {
```

```
    public void funkykeyEvent(KeyEvent e);
```

```
}
```



# Interface

- When you implement an interface, you need to implement all the declared functions.
- There can be a lot of methods

```
interface KeyListener {  
    public void keyPressed(KeyEvent e) ;  
    public void keyReleased(KeyEvent e) ;  
    public void keyTyped(KeyEvent e) ;  
}
```

- What if you only care about a couple of these methods, not all?

# Adapter Class

- An adapter class implements an interface and provides empty method bodies

```
class KeyAdapter implements KeyListener {  
    public void keyPressed(KeyEvent e) { };  
    public void keyReleased(KeyEvent e) { };  
    public void keyTyped(KeyEvent e) { };  
}
```

- You can override only the methods you care about
- This isn't elegant, but it does work
- Java provides a number of adapter classes

# Example...

- With interface you can write methods that work with more than one class

```
interface RuleSet {  
    boolean isLegal(Move m, Board b);  
    void makeMove(Move m);  
}
```

- Every class that implements `RuleSet` must have these methods

## ...Example...

```
class CheckersRules implements RuleSet {  
    public boolean isLegal(Move m, Board b)  
    { ... }  
    public void makeMove(Move m) { ... }  
}
```

```
class ChessRules implements RuleSet {  
    public boolean isLegal(Move m, Board b)  
    { ... }  
    public void makeMove(Move m) { ... }  
}
```

## ...Example

- Is this a legal statement?

```
RuleSet rulesOfThisGame = new ChessRules();
```

- This assignment is legal because a `rulesOfThisGame` object is a `RuleSet` object.
- Is this a legal statement?

```
if (rulesOfThisGame.isLegal(m, b)) {  
    rulesOfThisGame.makeMove(m);  
}
```

- This statement is legal because, `whatever` kind of `RuleSet` object `rulesOfThisGame` is, it must have `isLegal` and `makeMove` methods

# instanceof operator

- **instanceof** is a keyword that tells you whether a variable “is a” member of a class or interface

```
class Dog extends Animal implements Pet {...}  
Animal fido = new Dog();
```

- Are these true or false?

fido instanceof Dog

fido instanceof Animal

fido instanceof Pet

# Vocabulary ...

- **abstract method**
  - a method which is declared but not defined (it has no method body)
- **abstract class**
  - a class which either (1) contains abstract methods, or (2) has been declared abstract
- **Instantiate**
  - to create an instance (object) of a class
- **Interface**
  - Similar to a class, but contains only abstract methods (and possibly constants)

# ...Vocabulary

- **Adapter class**
  - A class that implements an interface but has only empty method bodies
- **Final methods**
  - methods that cannot be overridden
  - all private or static methods are implicitly final
- **Static (early) binding**
  - Binding occurs during compile time
  - Uses reference type during binding
- **Dynamic (late) binding**
  - Binding occurs during run time
  - Uses object type during binding



**Any Questions?**