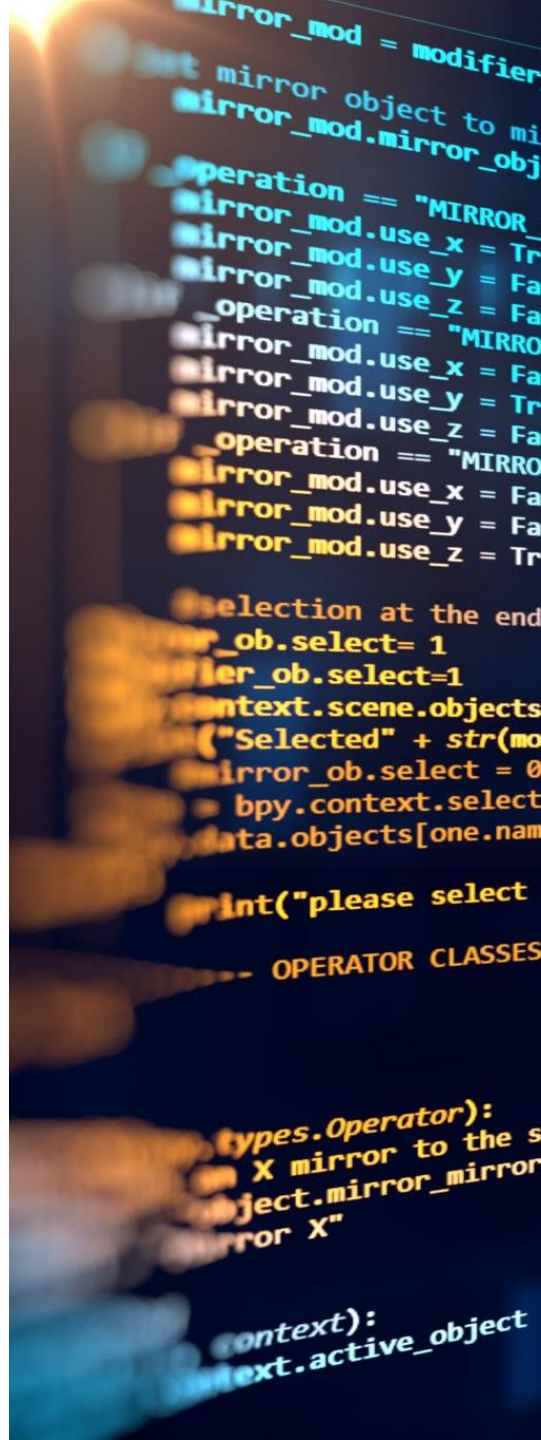
CS 1027
Fundamentals of Computer Science II

Inheritance in Java

Ahmed Ibrahim



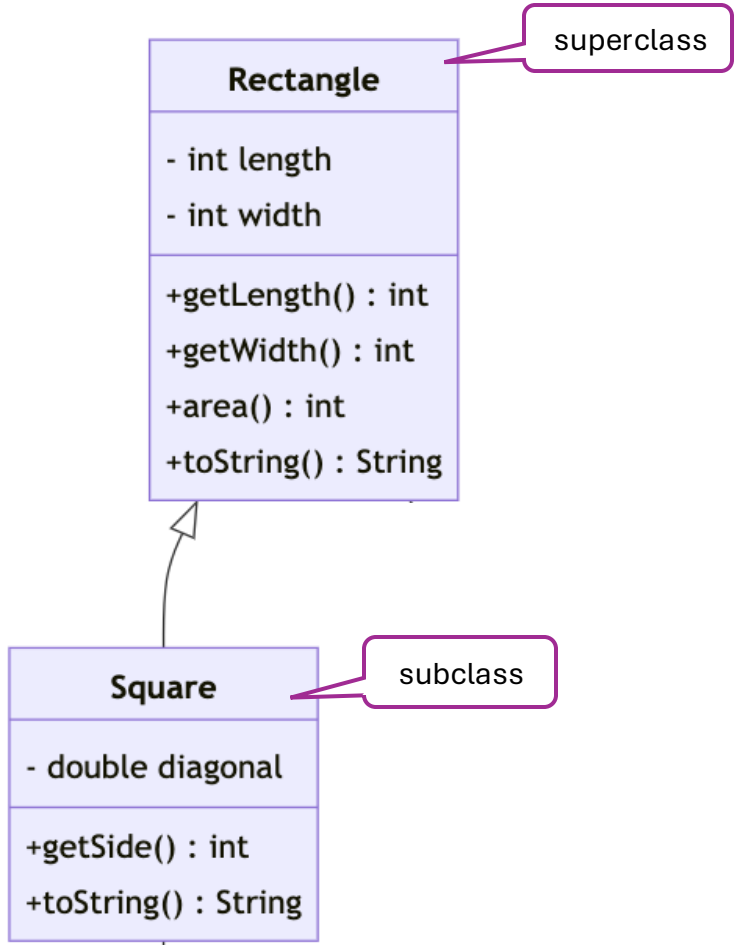
Recap

- Inheritance allows the creation of a new class (subclass) from an existing one (superclass), enabling code reuse and extension.
- **Terminology**
  - Superclass: The parent/base class
  - Subclass: The derived/child class
  - Overriding: Subclasses can redefine methods of the superclass
- **Benefits** – Reusability, maintainability, flexibility, and encapsulation
- **Key Concepts**
  - super keyword for accessing the parent class
  - this keyword for accessing the current instance
- **Examples** – Rectangle, Square, BankAccount, CheckingAccount, SavingsAccount Classes

Recall: Is-a Relationship

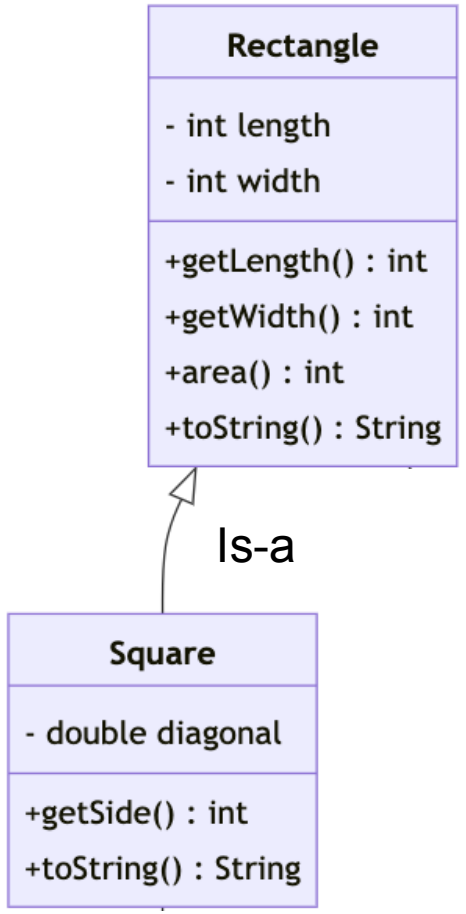
- A derived class is a more specific version of the original class.
- So, if A is a subclass of B, then an object of type A is also an instance of class B.
- Example: An object of class Square is also an object of class Rectangle
  - Is it true that a Rectangle object is also a Square?

Reverse Is Not True



Superclass Variables

```
Rectangle r = new Rectangle(5, 7);
Square s = new Square(4);

//Rectangle reference to a Square object
Rectangle r1 = new Square(4);
```



- Because an object of class Square is also an object of class Rectangle
- A variable of the superclass type may **reference** an **object** of a subclass type. This is a key feature of inheritance that enables ***polymorphism***!?

Superclass Variables (cont.)

- **Polymorphism** allows **one interface** (ex., Vehicle class) to be used for different data types (Car, bicycle, … etc.). This means you can write more general and flexible code that works with objects of different subclasses but treats them **uniformly**.
- For instance, if you have a *method* that accepts a Rectangle as a parameter, polymorphism allows you to pass any object that extends a Rectangle (such as a Square) without modifying the method. This is because, in Java, an object of a subclass is always treated as an instance of its superclass.

Question!

Which of the following statements is valid in Java inheritance?

A) Square s1 = new Rectangle(2, 3); **Not Valid**

B) Rectangle r1 = new Square(6);

C) Rectangle r = new Square(); **Technically Valid**

D) Rectangle r = s; where s is of type Square **Technically Valid**

Superclass and Subclass Variable Relationships

- A **variable** of the superclass type may reference an object of a subclass type.

```
Rectangle r = new Square(5);
```

- However, a **variable** of the subclass type cannot reference an object of the superclass type.

```
Square s = new Rectangle(4, 7);
```

Object Creation & Type Consistency

- The object type is determined when it is created and **does not change** throughout the execution of a program.

```
// Here, x is created as a Square object
Rectangle x = new Square(5);
```

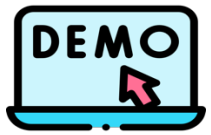
Square Class

```java
// Superclass
class Rectangle {
    private int length;
    private int width;

    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }

    public void displayType() {
        System.out.println("I am a Rectangle");
    }
}

// Subclass
class Square extends Rectangle {
    public Square(int side) {
        super(side, side);
    }

    @Override
    public void displayType() {
        System.out.println("I am a Square");
    }
}
```

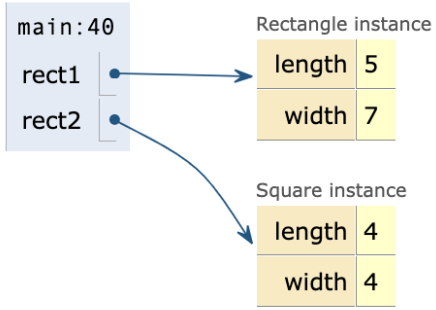
Another Example

- Even though rect2 is referenced as a Rectangle, it is still a Square object
- Its type was determined at the time it was created (new Square(4))

```
28  public class Main {
29      public static void main(String[] args) {
30          // Creating a Rectangle object
31          Rectangle rect1 = new Rectangle(5, 7);
32          // Creating a Square object but
33          // referencing it as a Rectangle
34          Rectangle rect2 = new Square(4);
35          // Outputs: I am a Rectangle
36          rect1.displayType();
37          // Outputs: I am a Square
38          rect2.displayType();
39
40      }
```



Memory:



Reference Objects of Different Types

Because of inheritance, a **variable** can refer to objects of different types during its lifetime.

```
Rectangle shape; // Declaring a variable

// Reassign the variable
shape = new Rectangle(4, 7);
System.out.println("Area of Rectangle: " +
shape.area());

// Reassign the variable
shape = new Square(5);
System.out.println("Area of Square: " +
shape.area());
```

Animal Class

```java
// Superclass
class Animal {
    public void makeSound() {
        System.out.println("Animal makes a sound");
    }
}

// Subclass 1
class Dog extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Bark");
    }
}

// Subclass 2
class Cat extends Animal {
    @Override
    public void makeSound() {
        System.out.println("Meow");
    }
}
```

Behavior Depends on Object Type

- Remember, What's printed depends on the actual type of the object (*not the* ***reference type***).

```
24  public class Main {
25      public static void main(String[] args) {
26          // Superclass reference holding a subclass object
27
28          // Animal reference to a Dog object
29          Animal myAnimal1 = new Dog();
30          // Animal reference to a Cat object
31          Animal myAnimal2 = new Cat();
32
33          // Outputs: Bark
34          myAnimal1.makeSound();
35          // Outputs: Meow
36          myAnimal2.makeSound();
37      }
```

Question!

- Consider the following code:

```
Shape s1 = new Circle();
Shape s2 = new Rectangle();
Shape s3 = new Triangle();
s1.draw();
s2.draw();
s3.draw();
```

- Assuming Shape is a superclass with a draw() method overridden by its subclasses (Circle, Rectangle, and Triangle), which of the following statements is TRUE?

A) All calls to draw() will execute the Shape version of the method.

B) Each draw() call executes the version defined by the actual object's type, not the reference type.

C) The code will result in a compilation error since s1, s2, and s3 are all declared as Shape.

D) Polymorphism does not allow s1, s2, and s3 to reference different object types.

Recall: Square Class

```java
// Superclass
class Rectangle {
    private int length;
    private int width;

    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }

    public void displayType() {
        System.out.println("I am a Rectangle");
    }
}

// Subclass
class Square extends Rectangle {
    public Square(int side) {
        super(side, side);
    }

    @Override
    public void displayType() {
        System.out.println("I am a Square");
    }
}
```

Dynamic Binding

When is it known which method should be invoked?

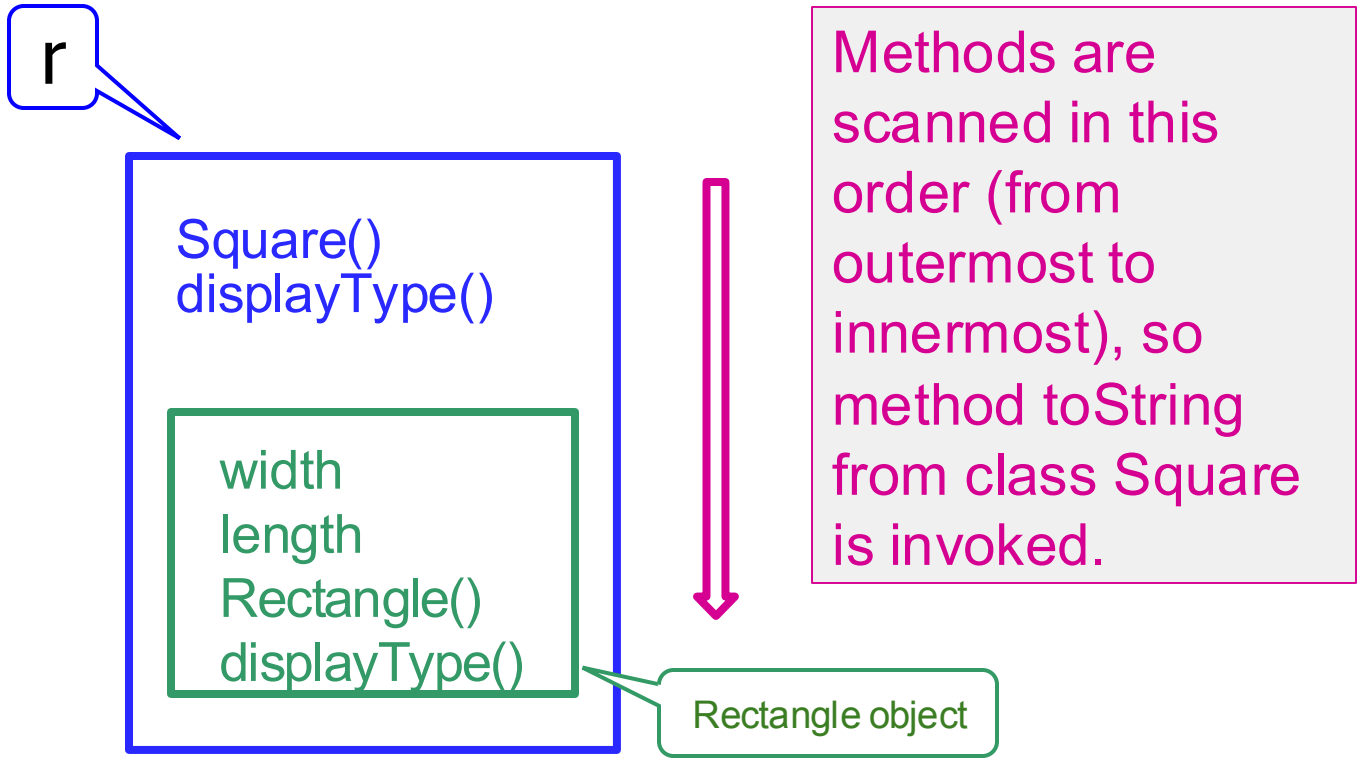
At runtime.

```
28  public class Main {
29      public static void main(String[] args) {
30          // Creating a Rectangle object
31          Rectangle r = new Rectangle(5, 7);
32
33          // Outputs: I am a Rectangle
34          r.displayType();
35
36          // Creating a Square object but
37          // referencing it as a Rectangle
38          r = new Square(4);
39
40          // Outputs: I am a Square
41          r.displayType();
42
43      }
```

- This is called dynamic binding or late binding of a variable to the object type.
- The decision to call **displayType**() from Rectangle or Square happens at runtime.
- **Why is this not known at compile time?**
- Since r can reference any subclass of Rectangle, the exact method cannot be determined until the program runs.

Dynamic Binding

- When a method is present in both the superclass and the subclass, the version from the subclass is executed.
- The method that is called must be defined in the superclass (or one of its parent classes); otherwise, a **compiler error** will occur.



Type Casting

Review: Casting Primitive Types

- We can use casting to convert some primitive types to other primitive types.
- Example:

```
int i, j, n;
n = (int) Math.random( );
double q = (double) i / (double) j;
```

- Note that this actually changes the representation from double to integer (second statement) or from integer to double (last statement).

Casting Reference Variables: **Upcasting**

- Recall:

```
//Here, x is created as a Square object
Rectangle x = new Square(5);
```

- Upcasting: Assigning a subclass object to a superclass reference.
- This is done implicitly.

Casting Reference Variables: **Downcasting**

- **Downcasting**: Converting a superclass reference back to a subclass reference.
- This must be done explicitly.
- Example:

```
Square s = (Square) r;
```

Downcasting

The Need for Downcasting

- Go back to the example:

```
Rectangle r = new Square(5);
System.out.println(r.getSide());
```

- This will generate a compiler error (why?)
  - The compiler error occurs because **r** is declared a Rectangle and lacks a **getSide**() method.
  - Although **r** references a Square object, the compiler only recognizes **r** as a Rectangle and restricts access to Square-specific methods.
- So, how do we overcome this? The answer lies in downcasting.
  - `System.out.println(((Square) r).getSide());`
- We can let the compiler know that we intend variable **r** to reference a Square object, by casting it to type Square.

Question!

Why might the following code result in a runtime error?

```
Animal a = new Animal();
Dog d = (Dog) a;
```

A) Because **a** is not an instance of **Dog**

B) Because **a** must be declared as a **Dog** for the cast to work

C) Because downcasting is only valid within the same class

D) Because **a** and **d** must be of the same type

Square Class

With modification

```java
// Superclass
class Rectangle {
    protected int length;
    protected int width;

    public Rectangle(int length, int width) {
        this.length = length;
        this.width = width;
    }

    public void displayType() {
        System.out.println("I am a Rectangle");
    }
}

// Subclass
class Square extends Rectangle {
    public Square(int side) {
        super(side, side);
    }

    public int getSide() {
        return this.length;
    }

    @Override
    public void displayType() {
        System.out.println("I am a Square");
    }
}
```

instanceof Operator

- We can also cast from one class type to another within an inheritance hierarchy.
- The compiler is now happy.
- Casting does not change the object being referenced!
- However, what if **r** was not actually referencing a Square object at the time of casting?
- The compiler would accept it, but a **runtime error** would occur.

```
32  public class Main {
33      public static void main(String[] args) {
34          // Creating a Rectangle object
35          Rectangle r = new Rectangle(5, 7);
36
37          // Outputs: I am a Rectangle
38          r.displayType();
39
40          // Creating a Square object but
41          // referencing it as a Rectangle
42          r = new Square(4);
43
44          // Outputs: I am a Square
45          r.displayType();
46
47          // Cast r to Square to access getSide()
48          if (r instanceof Square) {
49              int side = ((Square) r).getSide();
50              System.out.println("The side length of" +
51              "the square is: " + side);
52          }
53      }
54  }
```

instanceof Operator (cont.)

- A safer fix: use the instanceof operator

```
if (r instanceof Square) {
   System.out.println(((Square)r).getSide( ));
}
```

- Note that instanceof is an operator, not a method
  - An operator is a built-in language feature used to perform a specific operation. In this case, instanceof checks whether an object is an instance of a particular class or interface.
- It tests whether the referenced object is an instance of a particular class and gives the expression the value TRUE or FALSE.

Question!

Given the following three Java classes:

```
public class Animal
public class Dog extends Animal
public class Cat extends Animal
```

And the following variables:

```
Animal animalVar;
Dog dogVar;
Cat catVar;
```

Determine which of the following statements are correct, which generate compilation errors, and which cause runtime errors:

```
1. animalVar = new Dog();
2. animalVar = new Cat();
3. dogVar = new Cat();
4. dogVar = new Animal();
5. catVar = new Dog();
```


thank
you