GET SMART: WITH JAVA PROGRAMMING

YAMAN OMAR ALASHQAR

SYSTEM.OUT.PRINTLN(“WELCOME TO THIS COURSE”);

BANK ACCOUNT + SAVINGS BANK ACCOUNT + ACCOUNT HOLDER

- Deposit (returns a Boolean)
- Withdraw (returns a Boolean)
- Display Account Info (JoptionPane)
- Transfer to other account
- Log of all transactions (FOR A SINGLE ACCOUNT)
- NUMBER OF transactions (FOR ALL ACCOUNTS)

USING **THIS** TO INVOKE A CONSTRUCTOR

```
public class Rectangle {
    private int x, y;
    private int width, height;

    public Rectangle() {
        this(1, 1);
    }
    public Rectangle(int width, int height) {
        this( 0,0,width, height);
    }
    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }

}
```

Use **this()** to call a constructor from another overloaded constructor in the same class.

The call to **this()** can be used only in a constructor, and it must be the first statement in a constructor. It's used with constructor chaining, in other words when one constructor calls another constructor, and helps to reduce duplicated code.

```java
class Rectangle {
    private int x;
    private int y;
    private int width;
    private int height;

    public Rectangle() {
        this.x = 0;
        this.y = 0;
        this.width = 0;
        this.height = 0;
    }

    public Rectangle(int width, int height) {
        this.x = 0;
        this.y = 0;
        this.width = width;
        this.height = height;
    }

    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

- In this example we have three constructors.
- All three constructors initialize variables.
- There is repeated code in every constructor. We are initializing variables in each constructor with some default values.
- **You should never write constructors like this.**
- Let's look at the right way to do this by using a this() call.

```java
class Rectangle {
    private int x;
    private int y;
    private int width;
    private int height;

    // 1st constructor
    public Rectangle() {
        this(0, 0); // calls 2nd constructor
    }

    // 2nd constructor
    public Rectangle(int width, int height) {
        this(0, 0, width, height); // calls 3rd constructor
    }

    // 3rd constructor
    public Rectangle(int x, int y, int width, int height) {
        // initialize variables
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

- In this example we have three constructors.
- The 1st constructor calls the 2nd, the 2nd constructor calls the 3rd constructor, and the 3rd constructor initializes the instance variables.
- The 3rd constructor does all the work
- No matter what constructor we call, the variables will always be initialized in 3rd constructor
- This is known as constructor chaining, the last constructor has the “responsibility” to initialize the variables.

BAD

```java
class Rectangle {
    private int x;
    private int y;
    private int width;
    private int height;

    public Rectangle() {
        this.x = 0;
        this.y = 0;
        this.width = 0;
        this.height = 0;
    }

    public Rectangle(int width, int height) {
        this.x = 0;
        this.y = 0;
        this.width = width;
        this.height = height;
    }

    public Rectangle(int x, int y, int width, int height) {
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

GOOD

```java
class Rectangle {
    private int x;
    private int y;
    private int width;
    private int height;

    // 1st constructor
    public Rectangle() {
        this(0, 0); // calls 2nd constructor
    }

    // 2nd constructor
    public Rectangle(int width, int height) {
        this(0, 0, width, height); // calls 3rd constructor
    }

    // 3rd constructor
    public Rectangle(int x, int y, int width, int height) {
        // initialize variables
        this.x = x;
        this.y = y;
        this.width = width;
        this.height = height;
    }
}
```

INHERITANCE

INHERITANCE

- Inheritance in Java is a mechanism in which one object acquires all the properties and behaviors of a parent object.
- The idea behind inheritance in Java is that you can create new classes that are built upon existing classes. When you inherit from an existing class, you can reuse methods and fields of the parent class. Moreover, you can add new methods and fields in your current class also.
- Inheritance represents the **IS-A relationship** which is also known as a *parent-child* relationship.

TERMS USED IN INHERITANCE

- **Class:** A class is a group of objects which have common properties. It is a template or blueprint from which objects are created.
- **Sub Class/Child Class:** Subclass is a class which inherits the other class. It is also called a derived class, extended class, or child class.
- **Super Class/Parent Class:** Superclass is the class from where a subclass inherits the features. It is also called a base class or a parent class.
- **Reusability:** As the name specifies, reusability is a mechanism which facilitates you to reuse the fields and methods of the existing class when you create a new class. You can use the same fields and methods already defined in the previous class.

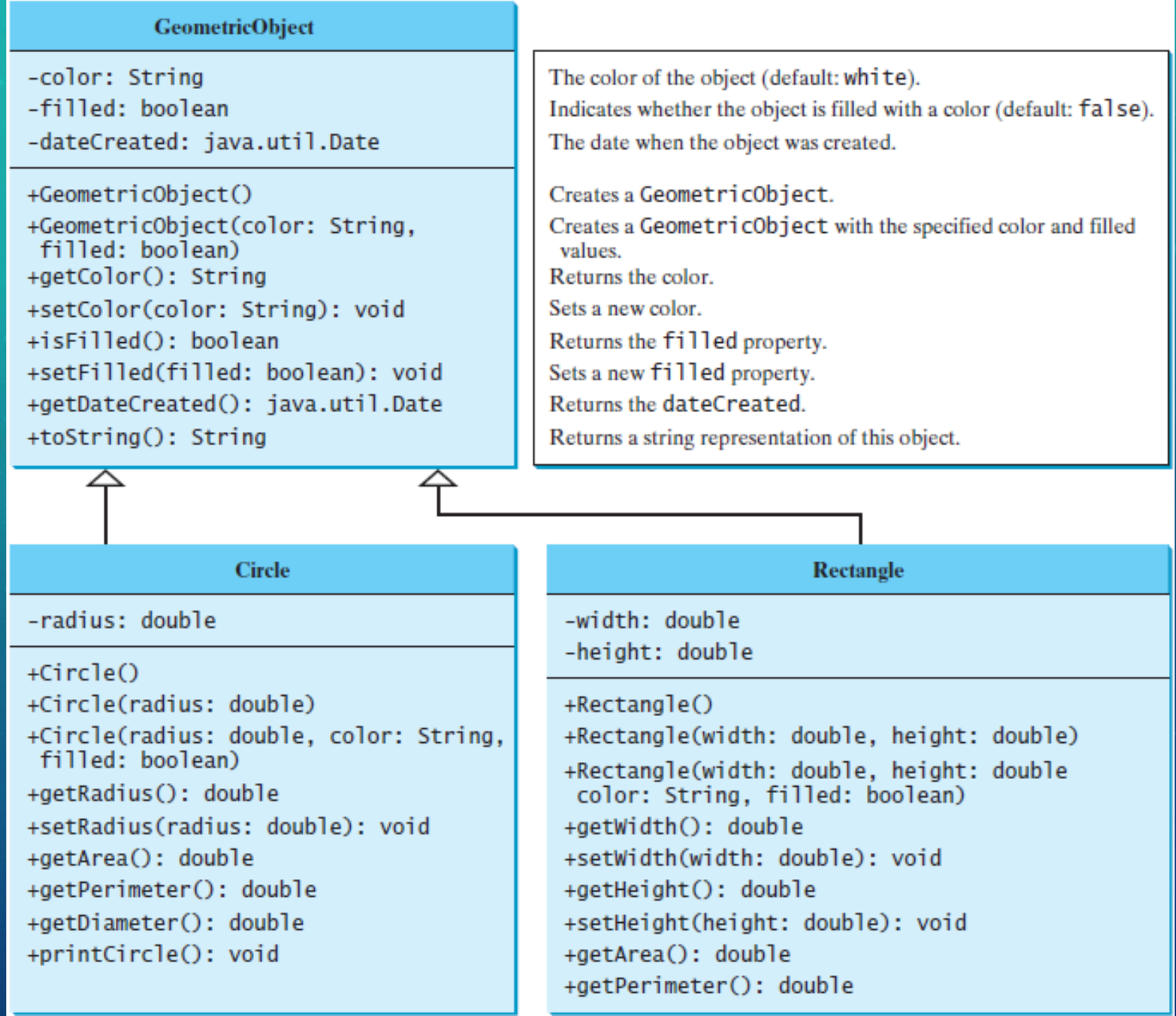
THE SYNTAX OF JAVA INHERITANCE

```
class Subclass-name extends Superclass-name
{
  //methods and fields
}
```

- The extends keyword indicates that you are making a new class that derives from an existing class.
- The meaning of "extends" is to increase the functionality.

EXAMPLE: GEOMETRIC OBJECTS

- Geometric objects as circles and rectangles have many common properties and behaviors.
- A general class GeometricObject class contains the properties as color and filled and their appropriate getter and setter methods, dateCreated property and the getDateCreated() and toString() methods.
- A Circle is a special type (i.e. a subclass) of GeometricObject
- Likewise, Rectangle can also be defined as a subclass of GeometricObject.
- A subclass inherits accessible data fields and methods from its superclass and may also add new data fields and methods.


GeometricObject
-color: String
-filled: boolean
-dateCreated: java.util.Date
+GeometricObject()
+GeometricObject(color: String, filled: boolean)
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
The color of the object (default: white).
Indicates whether the object is filled with a color (default: false).
The date when the object was created.
Creates a GeometricObject.
Creates a GeometricObject with the specified color and filled values.
Returns the color.
Sets a new color.
Returns the filled property.
Sets a new filled property.
Returns the dateCreated.
Returns a string representation of this object.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: String, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double
+printCircle(): void
Rectangle
-width: double
-height: double
+Rectangle()
+Rectangle(width: double, height: double)
+Rectangle(width: double, height: double color: String, filled: boolean)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

METHOD OVERRIDING

- If subclass (child class) has the same method as declared in the parent class, it is known as method overriding in Java.
- Method overriding is used to provide the specific implementation of a method which is already provided by its superclass.
- Method overriding is used for runtime polymorphism

Rules for Java Method Overriding

STEP 01

Method must have same name as in the parent class

STEP 02

Method must have same parameter as in the parent class.

STEP 03

There must be IS-A relationship (inheritance).

Can we override static method?

No, a static method cannot be overridden. It can be proved by runtime polymorphism, so we will learn it later.

Why can we not override static method?

It is because the static method is bound with class whereas instance method is bound with an object. Static belongs to the class area, and an instance belongs to the heap area.

Can we override java main method?

No, because the main is a static method.

No.	Method Overloading	Method Overriding
1)	Method overloading is used *to increase the readability* of the program.	Method overriding is used *to provide the specific implementation* of the method that is already provided by its super class.
2)	Method overloading is performed *within class*.	Method overriding occurs *in two classes* that have IS-A (inheritance) relationship.
3)	In case of method overloading, *parameter must be different*.	In case of method overriding, *parameter must be same*.
4)	Method overloading is the example of *compile time polymorphism*.	Method overriding is the example of *run time polymorphism*.
5)	In java, method overloading can't be performed by changing return type of the method only. *Return type can be same or different* in method overloading. But you must have to change the parameter.	*Return type must be same or covariant* in method overriding.

- The keyword **super** is used to access/call the parent class members (variables and methods).
- The keyword **this** is used to call the current class members (variables and methods). This is required when we have a parameter with the same name as an instance variable (field).

- NOTE: We can use both of them anywhere in a class except static areas(the static block or a static method). Any attempt to do so will lead to compile-time errors (more on static later in the course).

The keyword **super** is commonly used with **method overriding**, when we call a method with the same name from the parent class. In the example below we have a method printMethod that calls super.printMethod.

```
class SuperClass { // parent class aka super class
    public void printMethod() {
        System.out.println("Printed in Superclass.");
    }
}

class SubClass extends SuperClass { // subclass aka child class
    // overrides method from parent
    @Override
    public void printMethod() {
        super.printMethod(); // calls method in SuperClass (parent)
        System.out.println("Printed in Subclass");
    }
}

class MainClass {
    public static void main(String[] args) {
        SubClass s = new SubClass();
        s.printMethod();
    }
}
```

In other words it's calling the method with the same name from the parent class. Without the keyword **super** in this case it would be recursive call. Meaning that the method would call it self forever (or until memory is fully used). That is why the **super** keyword is needed.

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

    public Shape(int x, int y) {
        this.x = x;
        this.y = y;
    }
}

class Rectangle extends Shape {
    private int width;
    private int height;

    // 1st constructor
    public Rectangle(int x, int y) {
        this(x, y, 0, 0); // calls 2nd constructor
    }

    // 2nd constructor
    public Rectangle(int x, int y, int width, int height) {
        super(x, y); // calls constructor from parent (Shape)
        this.width = width;
        this.height = height;
    }
}
```

- In this example we have a class Shape with x,y variables and class Rectangle that extends Shape with variables width and height
- In Rectangle, the 1st constructor we are calling the 2nd constructor
- The 2nd constructor calls the parent constructor with parameters x and y
- The parent constructor will initialize x,y variables while the 2nd Rectangle constructor will initialize the width and height variables
- Here we have both **super()** and **this()** calls

Association - I have a relationship with an object. `Foo` uses `Bar`

```
public class Foo {
    void Baz(Bar bar) {
    }
};
```

Composition - I own an object and I am responsible for its lifetime. When `Foo` dies, so does `Bar`

```
public class Foo {
    private Bar bar = new Bar();
}
```

Aggregation - I have an object which I've borrowed from someone else. When `Foo` dies, `Bar` may live on.

```
public class Foo {
    private Bar bar;
    Foo(Bar bar) {
       this.bar = bar;
    }
}
```

AGGREGATION

- If a class have an entity reference, it is known as Aggregation.
- Aggregation represents HAS-A relationship.

AGGREGATION

Consider a situation, Employee object contains many informations such as id, name, email etc.

- It contains one more object named address, which contains its own informations such as city, state, country, zipcode etc. as given below.

```
class Employee{
int id;
String name;
Address address;//Address is a class
...
}
```

POLYMORPHISM

- 3 Types of vehicles (Car, Boat, Motorcycle)
- Different Behavior (.move())

POLYMORPHISM

- A method can be defined in a superclass and overridden in its subclass.
- The JVM decides which method is invoked at runtime.

ACCESS MODIFIERS

- A subclass cannot weaken the accessibility of a method defined in the superclass.
- For example, if a method is defined as public in the superclass, it must be defined as public in the subclass.

POLYMORPHISM EXAMPLE

```
public class PolymorphismDemo {

public static void main(String[] args) {
SimpleGeometricObject x = new Circle (1, "red", false);
SimpleGeometricObject y = new Rectangle (1, 1, "black", true));
    displayObject(x);
    displayObject(y);
  }
  public static void displayObject(SimpleGeometricObject object) {
    System.out.println("Created on " + object.getDateCreated() +  ". Color is "
+ object.getColor());
  }
}
```

Polymorphism promotes extensibility: *Software written to invoke polymorphic behavior is written independently of the specific types of the objects to which messages are sent. Thus, new types of objects that can respond to existing messages can be incorporated into such a system without modifying the base system. Only client code that instantiates new objects must be modified to accommodate new types.*