

Introduction to Java for C++ Programmers

Segment - 1

JAC 444

Professor: Mahboob Ali

Abstraction

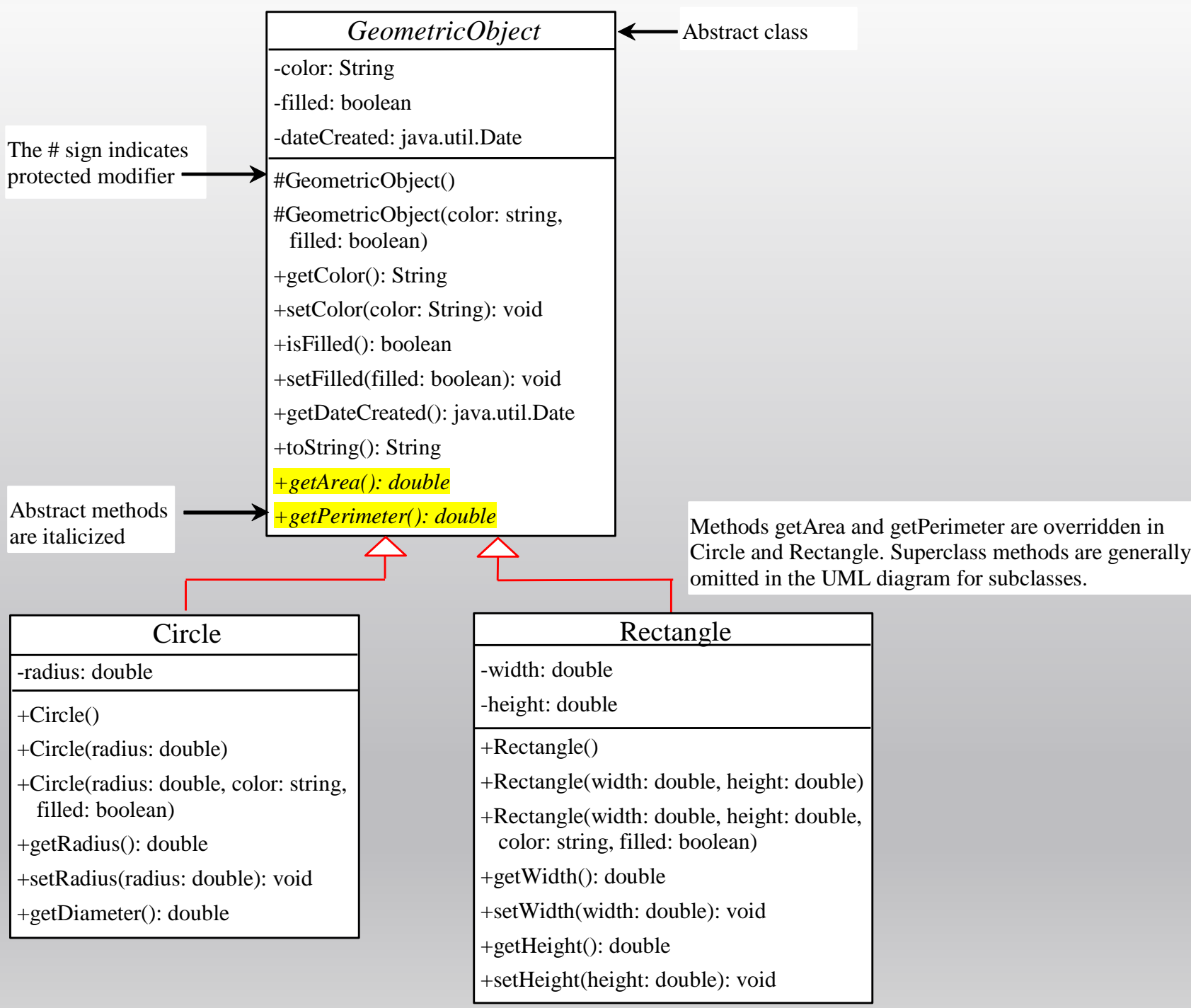
- As abstraction says hide the implementation from the user by showing only functionality.
- How to achieve Abstraction?
 - Two ways to achieve abstraction in java:
 1. Abstract classes.
 2. Interfaces.

Abstract classes and Abstract Methods

- Class declared with a keyword **abstract**, known as an abstract class.

```
abstract class Bookmark{  
    ...  
}
```

- Needs to be extended.
- Cannot be instantiated.
- An abstract class can contain abstract methods that are implemented in concrete subclasses.



An Abstract method

- Declared with the keyword **abstract** and does not contain any implementation

- **abstract void** someMethod();

No
Body

Can't be
static

Must be
overridden

- An abstract method cannot be contained in a *non-abstract* class.
- If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be defined abstract. In other words, in a non-abstract subclass extended from an abstract class, all the abstract methods must be implemented, even if they are not used in the subclass.

object cannot be created from abstract class

An abstract class cannot be instantiated using the new operator, but you can still define its constructors, which are invoked in the constructors of its subclasses.

For instance, the constructors of GeometricObject are invoked in the Circle class and the Rectangle class.

abstract class without abstract method

- A class that contains abstract methods must be abstract.
- However, it is possible to define an abstract class that contains no abstract methods.
- Still cannot create instances of the class using the new operator. This class is used as a base class for defining a new subclass.

superclass of abstract class may be concrete

- A subclass can be abstract even if its superclass is concrete.
- For example, the Object class is concrete, but its subclasses, such as GeometricObject, may be abstract.

concrete method overridden to be abstract

- A subclass can override a method from its superclass to define it abstract.
- This is rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be defined abstract.

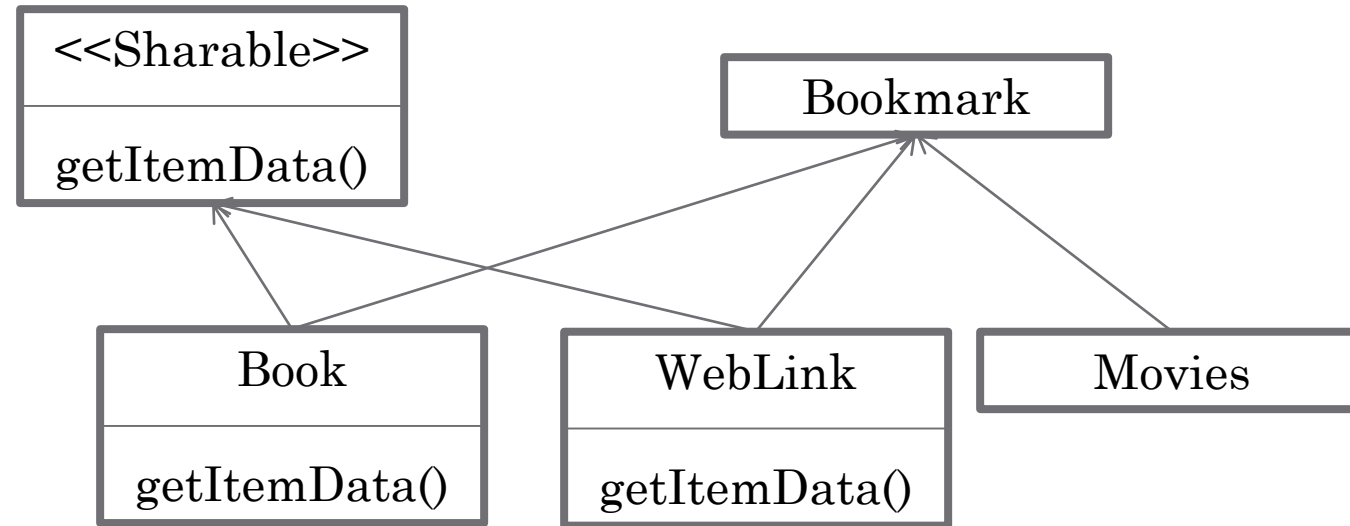
abstract class as type

- You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type.
- Therefore, the following statement, which creates an array whose elements are of GeometricObject type, is correct.
- GeometricObject[] geo = new GeometricObject[10];

Concrete subclass

- Must override *unimplemented* abstract methods.

Multiple Inheritance



- Java does not support multiple inheritance.
- Why?

Diamond Problem

Interfaces

- What is an interface?
- Why is an interface useful?
- How do you define an interface?
- How do you use an interface?

- An interface is a class like construct that contains only constants and abstract methods.
- In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify common behavior for objects.
- For example, you can specify that the objects are comparable, edible, cloneable using appropriate interfaces.

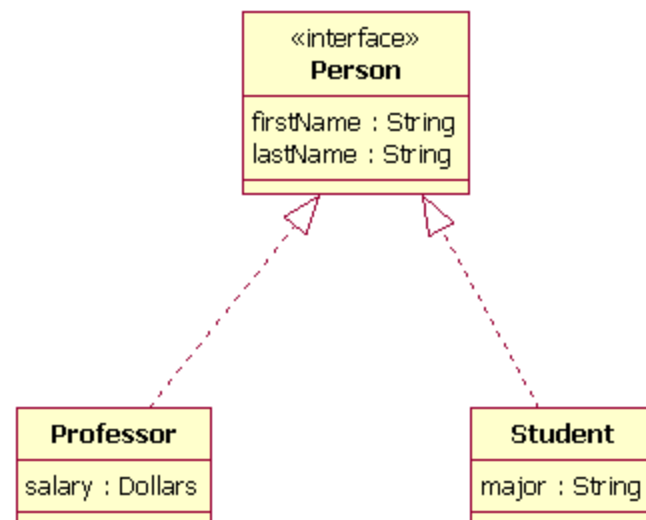
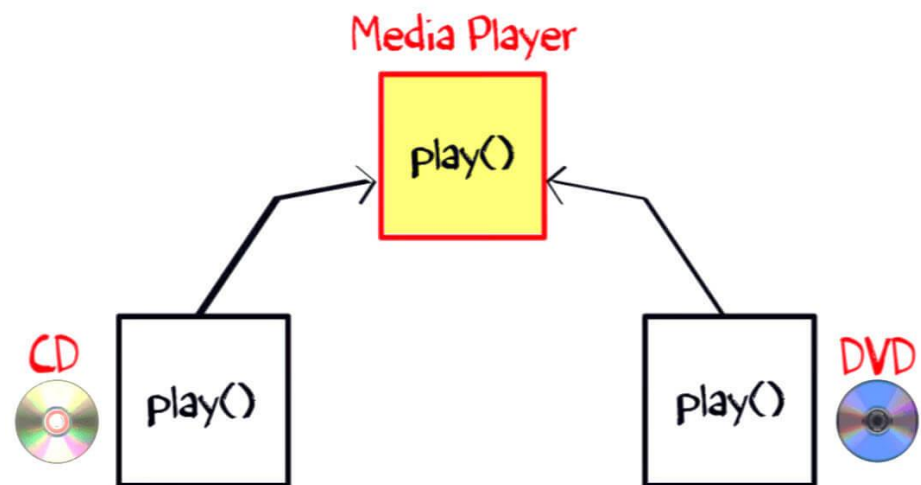
Interface Definition

- Interface is a reference type in Java, similar to a class.
- Interfaces cannot be instantiated, they can only be *implemented* by a class or *extended* by other interfaces.

```
interface InterfaceName {  
    constant(s) - final static fields  
    abstract method declaration(s)  
    default method(s)  
    static method(s)  
    nested types  
}
```

An interface creates a new reference data type, just as class definition

```
InterfaceName refVariable;
```



Interface Structure

```
public interface Sharable{  
    String getItemData();  
}
```


- abstract method is *public and static* by default (modifiers omitted)
- fields are *public, static and final* by default (modifiers omitted)
- All members are *public* by default
- Member's can't be *private & protected*.

Demo time

Interface Example

```
public interface Conversion { double  
    INCH_TO_MM = 25.4; double  
    inchToMM(double inches);  
}
```

```
public interface ConversionVersion2 {  
    double INCH_TO_MM = 25.4;  
    double inchToMM(double inches);  
    default public void defaultMethod() {  
        System.out.println("Special implementation");  
    }  
}
```



default method

```
public interface Edible {
```

```
    /** Describe how to eat */
```

```
    public abstract String howToEat();
```

```
}
```

```
    public class TestEdible {
```

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

```
            Object[] objects = {new Tiger(), new Chicken(), new Apple()};
```

```
            for (int i = 0; i < objects.length; i++) {
```

```
                if (objects[i] instanceof Edible)
```

```
                    System.out.println(((Edible)objects[i]).howToEat());
```

```
                if (objects[i] instanceof Animal) {
```

```
                    System.out.println(((Animal)objects[i]).sound());
```

```
                }            }        }
```

```
abstract class Animal {

    /** Return animal sound */

    public abstract String sound();

}

class Tiger extends Animal {
    @Override
    public String sound() {
        return "Tiger: RROOAARR";
    }
}
```

```
class Chicken extends Animal implements Edible {
    @Override
    public String howToEat() {
        return "Chicken: Fry it";
    }

    @Override
    public String sound() {
        return "Chicken: cock-a-doodle-doo";
    }
}
```

```
abstract class Fruit implements Edible {  
    // Data fields, constructors, and methods omitted here  
}  
    class Apple extends Fruit {  
        @Override  
        public String howToEat() {  
            return "Apple: Make apple cider";  
        }  
    }  
    class Orange extends Fruit {  
        @Override  
        public String howToEat() {  
            return "Orange: Make orange juice";  
        }  
    }
```

Comparing

- How do you compare two values in java?
 - By using the == operator

- How do you compare two string values in java?

- By using == operator

← Compares the reference not values

- By using compareTo()

← Compares lexicographically i.e. <, >, == and returns an int

- By using equals()

← Compares the original contents

The equals Method

The `equals()` method compares the contents of two objects. The default implementation of the equals method in the Object class is as follows:

```
public boolean equals(Object obj) {  
    return (this == obj);  
}
```

For example, the equals method is overridden in the `Circle` class.

```
public boolean equals(Object o) {  
    if (o instanceof Circle) {  
        return radius == ((Circle)o).radius;  
    }  
    else  
        return false;  
}
```

Interfaces vs. Abstract Classes

In an interface, the data must be constants; an abstract class can have all types of data.

Each method in an interface has only a signature without implementation; an abstract class can have concrete methods.

	Variables	Constructors	Methods
Abstract class	No restrictions	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be <u>public static final</u>	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

Some Common Interfaces of Java API

- ***Comparable:***

- Interface Comparable is used to allow objects of the class that implements the interface to be compared to another.
- Common use for ordering the objects in collections like ArrayList.

- ***Serializable:***

- Used to identify classes whose objects can be written to (serialized) or read from (deserialized) some type of storage (file, disk, database etc.) or transmitted across a network.

- ***Runnable:***

- Used by any class that represents a task to perform, Multithreading.

- ***AutoCloseable:***

- Used for closing resources of the current object. Prevents resource leaks.
- Invoked automatically on objects managed by the *try-with-resources* statement.

Java 8: Default and Static methods

- Only abstract methods allowed ~ prior to Java 8
- Why default and static methods?
- To allow the developers to add new methods to the interface without affecting the classes that implements these classes.
- Place the default keyword in front of the method in an interface.
- Must provide the body as well in the interface.

Java 9: Interface changes

- Java 9 introduces the
 - private interface methods
 - private static interface methods
 - Only accessible inside the interface.
 - Not allowed to inherit into another interface or class.
 - Now allowed to have private default methods.
 - Not allowed to have private abstract methods.
- Benefits:
 - Code reusability inside interface.