NYU

Courant Institute of Mathematical Sciences
Department of Computer Science
CS101 Introduction to Computer Science

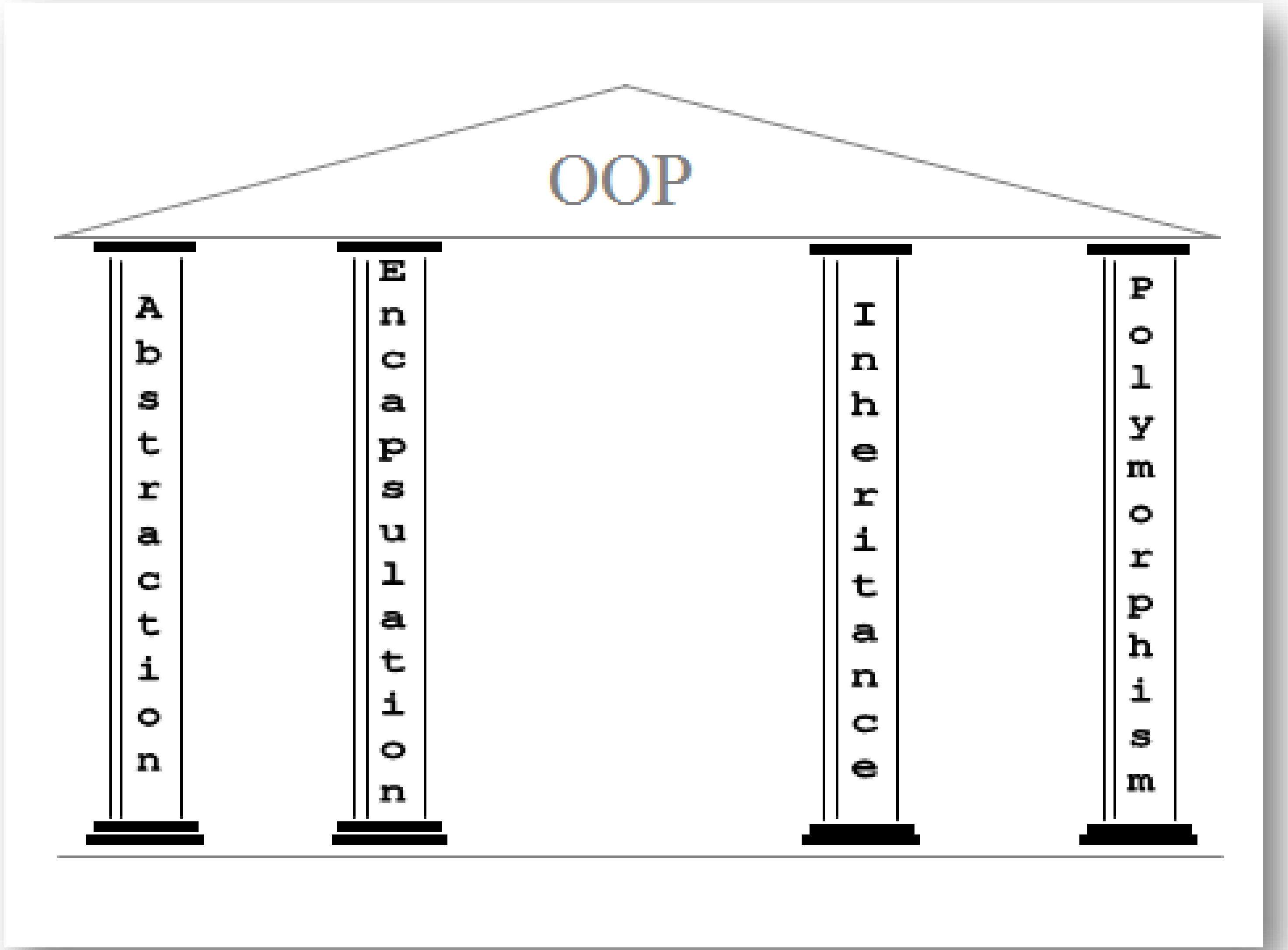
Anasse Bari, Ph.D.

Chapter#11: Main Pillars of the Object Oriented Programming Paradigm

Objectives

- ❖ Introducing the concepts of Encapsulation and Inheritance
- ❖ Learning the Super keyword, superclass methods and data fields
- ❖ Introducing the concept of polymorphism
- ❖ Learning how to use *interfaces* and *abstract classes* in Java
- ❖ Introducing the *comparable* java interface
- ❖ Introducing *casting* and *instance of*
- ❖ Understanding where to use an interface or an abstract class

Main Pillars of OOP



Review Summary on OOP

- ❖ Object-oriented Design: A problem-solving methodology that produces a solution to a problem in terms of self-contained entities called objects

- ❖ Object: A thing or entity that makes sense within the context of the problem

- ❖ Class: A class is a blueprint or template or set of instructions to build a specific type of object. Every object is built from a class.

Encapsulation

- ❖ Encapsulation means that all data members (fields) of a class are declared private. Some methods may be private, too.
- ❖ The class interacts with other classes (called the clients of this class) only through the class's constructors and public methods.
- ❖ Constructors and public methods of a class serve as the interface to class's clients.
- ❖ Hides the fine detail of the inner workings of the class
- ❖ The implementation is hidden
- ❖ Often called "information hiding"
- ❖ Part of the class is visible
- ❖ The necessary controls for the class are left visible
- ❖ The class interface is made visible
- ❖ The programmer is given only enough information to use the class

Encapsulation

- ❖Access to instance variables (aka member variables, class variable)
- ❖private does not expose the dot operator (object.variable)
- ❖Protect private instances variables (Getters and Setters)
- ❖Setters can validate parameters:

```
void setSize(int a){
    if (a > 0)
        size = a;
    else
        size = 0;
}
```

- ❖Getters can protect and modify data if necessary:

```
String getSocialSecurityNumber(){
    return “XXX-XX-” + ssn.substring(5); }
```

Encapsulation

- ❖ Changes in the code create software maintenance problems

- ❖ Usually, the structure of a class (as defined by its fields) changes more often than the class’s constructors and methods

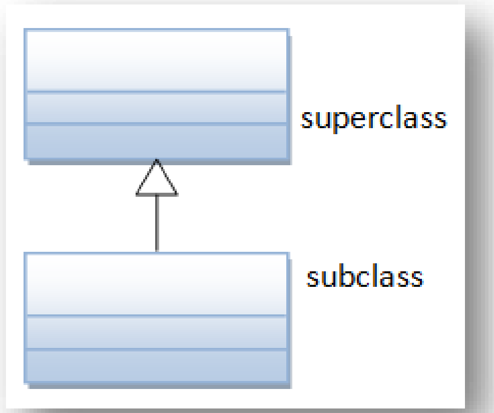
Inheritance

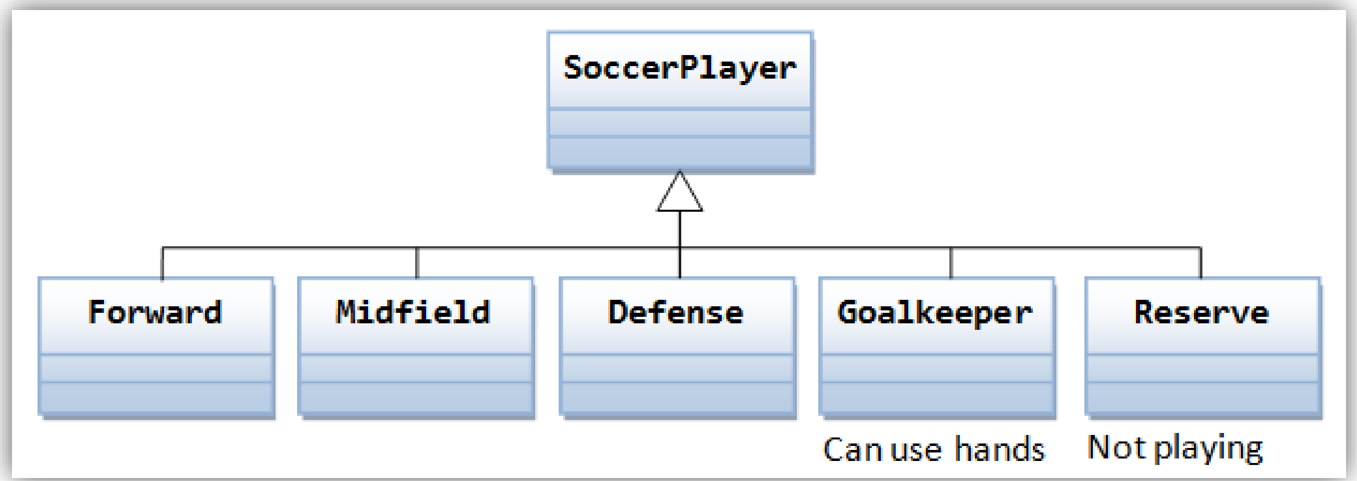
- ❖Inheritance allows a software developer to derive a new class from an existing one
- ❖The existing class is called the parent class, or superclass, or base class
- ❖The derived class is called the child class or subclass
- ❖As the name implies, the child inherits characteristics of the parent That is, the child class inherits the methods and data defined by the parent class
- ❖A class can extend another class, inheriting all its data members and methods while redefining some of them and/or adding its own.
- ❖Inheritance represents the is a relationship between data types

Inheritance

- ❖A programmer can tailor a derived class as needed by adding new variables or methods, or by modifying the inherited ones
- ❖Software reuse is a fundamental benefit of inheritance
- ❖By using existing software components to create new ones, we capitalize on all the effort that went into the design, implementation, and testing of the existing software

Inheritance – SoccerPlayer example

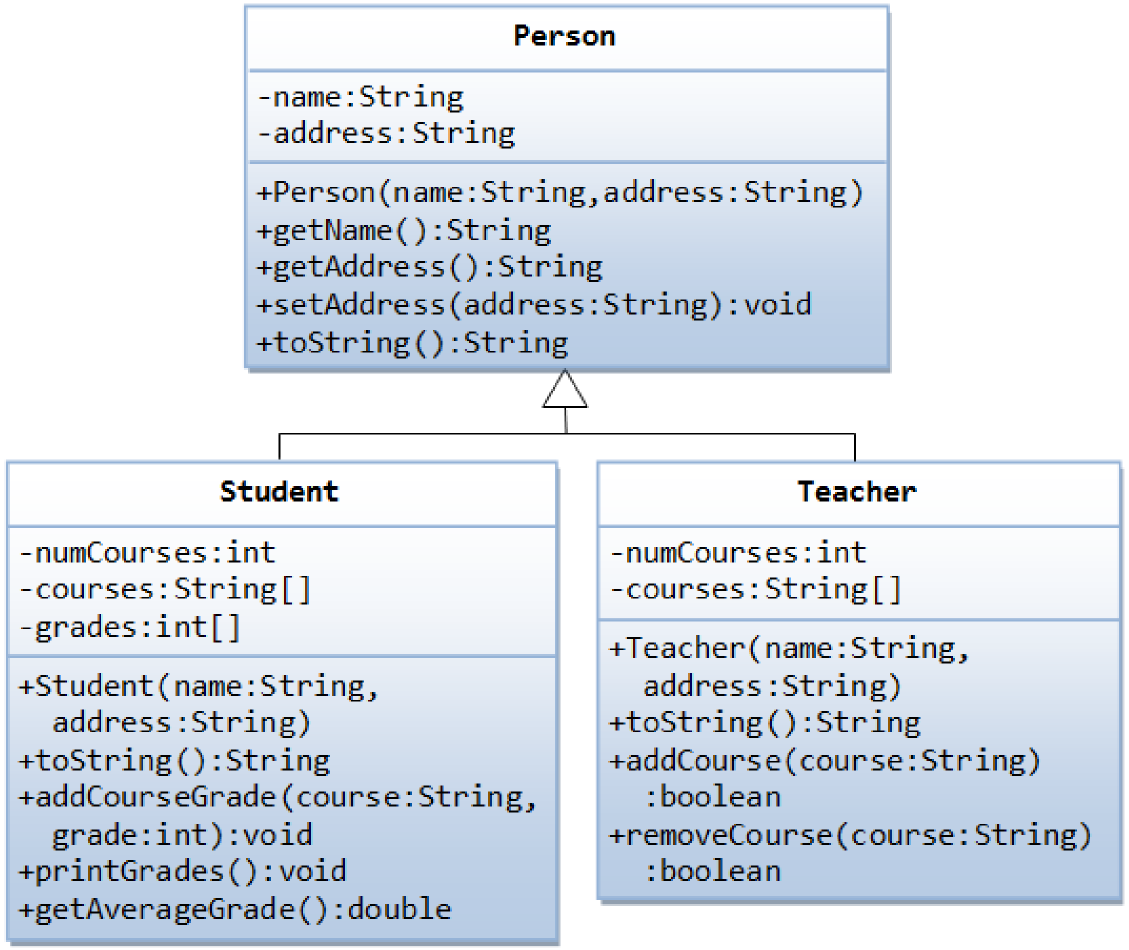




Class Goalkeeper **extends** SoccerPlayer {......}

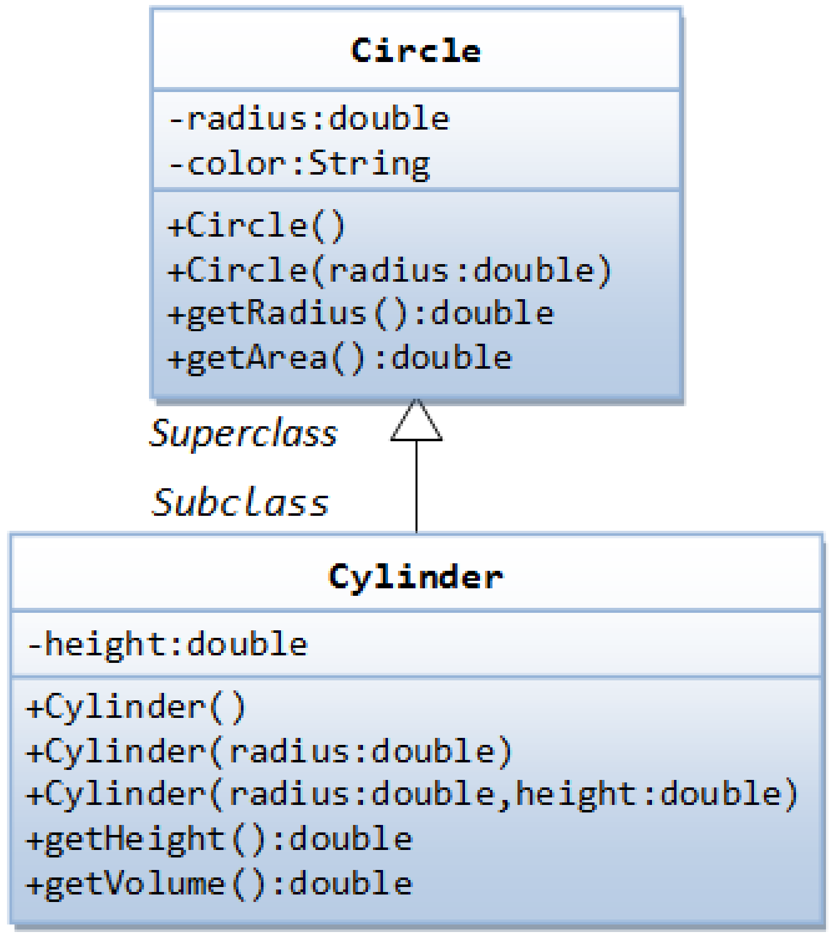
…

Inheritance – Person example

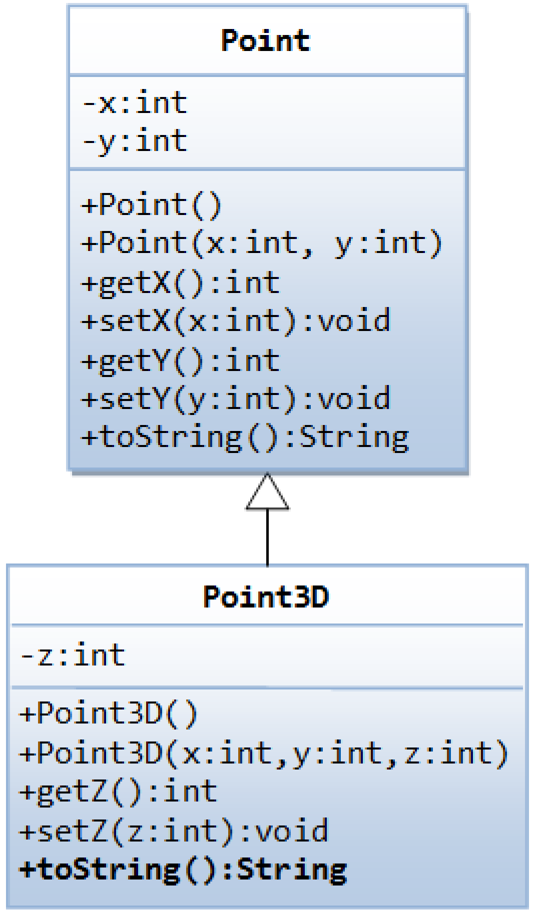


Person
-name:String
-address:String
+Person(name:String,address:String)
+getName():String
+getAddress():String
+setAddress(address:String):void
+toString():String
Student
-numCourses:int
-courses:String[]
-grades:int[]
+Student(name:String, address:String)
+toString():String
+addCourseGrade(course:String, grade:int):void
+printGrades():void
+getAverageGrade():double
Teacher
-numCourses:int
-courses:String[]
+Teacher(name:String, address:String)
+toString():String
+addCourse(course:String) :boolean
+removeCourse(course:String) :boolean

Inheritance – Circle-Cylinder example



Inheritance – 2D/3D point example



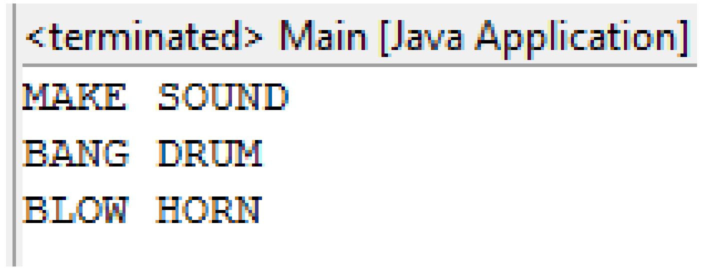
Inheritance - example

```
 public class Instrument {

     public void produceSound(){
         System.out.println("MAKE SOUND");
     }

     public void tuneInstrument(){
         System.out.println("Tuning Instrument");
     }
 }
```

```
public class Drum extends Instrument {
    public void produceSound(){
        System.out.println("BANG DRUM");
    }
}
```

```
public class Horn extends Instrument {
    public void produceSound(){
        System.out.println("BLOW HORN");
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Instrument one, two, three;

        one = new Instrument();
        two = new Drum();
        three = new Horn();

        one.produceSound();
        two.produceSound();
        three.produceSound();
    }
}
```



```
<terminated> Main [Java Application]
MAKE SOUND
BANG DRUM
BLOW HORN
```

Inheritance – Circle-Cylinder example

The *super* Reference

- Constructors are not inherited, even though they have public visibility
- Yet we often want to use the parent's constructor to set up the "parent's part" of the object
- The super reference can be used to refer to the parent class, and often is used to invoke the parent's constructor

Inheritance: *Super* keyword

- ❖The keyword "super" is used to access superclass constructors, methods, and data fields
- ❖Examples:

  super(); invokes the default constructor of the superclass.

  This call can be made only from within a constructor of a subclass
- ❖super(paramList);

  Invokes the constructor with parameters of the superclass (assuming it exists).
  This call can only be made from within a constructor of a subclass.

Inheritance: *Super* keyword

- ❖ super.methodName(paramList);

  Invokes a method of the superclass. This call can be made from anywhere in the subclass.

- ❖ super.dataField;

  Accesses a data Field of the superclass. This call can be made from anywhere in the subclass.

Inheritance: Superclass methods and data fields

A subclass inherits all public and protected methods and data fields of its superclass.

As a result, a subclass can use the methods and data fields it inherits without defining them.

Remember that there are 4 types of java access modifiers:

- **Private:** The private access modifier is accessible only within class.
- **Default:** If you don't use any modifier, it is treated as default bydefault. The default modifier is accessible only within package.
- **Protected:** The protected access modifier is accessible within package and outside the package but through inheritance only. The protected access modifier can be applied on the data member, method and constructor. It can't be applied on the class.
- **Public:** The public access modifier is accessible everywhere. It has the widest scope among all other modifiers.

Inheritance: Superclass Example 1

```
public class C1{
        protected int x;
        public void setX (int x) {
                this.x = x;
        }
        //the rest of the class definition
    }

public class C2 extends C1{
        protected float c;
        public C2 (float c, int x) {

                this.c = c;
                setX(x);
        }
        //the rest of the class definition
    }
```

Inheritance: Superclass Example 1 (cont’d)

In this example, the class C2 can access the data field “x” and the method “setX()” without defining them because they are inherited from C1.

Inheritance: Superclass Example 2

In the following example, the keyword “protected” is used to indicate data fields and methods that are accessible only from within the class and those that inherit from it.

One can use the “super” keyword to indicate the inherited methods but it isn’t necessary, unless the methods are overridden. The class C2 from Example 1 can be rewritten as shown in Example 2.

Superclass Example 2

```
public class C2 extends C1 {
        protected float c;
        public C2 (float c, int x) {
                this.c=c;
                super.setX(x);
                }
                //the rest of class definition
        }
```

Inheritance: Constructors and Inheritance

Constructors of the superclass are not inherited by its subclass. They can be invoked using the *super* keyword within the subclass constructors.

Inheritance:
Constructors and Inheritance Example

```
public class C1 {
        public C1 (int c) [ //do something ]
        //class definition
}

public class C2 extends C1{
        public C2 (int c, int x) {
                super(c);
                //do something
        }
        //class definition
}
```

Inheritance: Constructors and Inheritance Example (cont'd)

In this example, an object of the C2 class can be used with two parameter constructors, **BUT NOT** one parameter constructor or the default constructor.

C2 object1 = **new C2(3,5); //is valid**

C2 object2 = **new C2(3);   //is NOT valid**

C2 object3 = **new C2(); //is NOT valid**

Inheritance: Constructors and Inheritance

Constructor Chaining: constructing an instance of a class invokes the constructors of **ALL** the superclasses along the inheritance chain.

A constructor may:

- Invoke and overloaded constructor (of its own class), or
- Invoke its superclass constructor (this has to be done in the first line of the constructor).

If neither of these happens, the java compiler adds super() as the first statement in the constructor.

Inheritance: Overriding methods of the superclass

❖ **Overriding** is redefining a method of a superclass in a subclass.

❖ The **overridden method** has to have the same name, parameter list, and return type as the method in the superclass.

Inheritance: Overriding methods of the superclass Example

```
public class C1 {
        public void sayHi() {
                System.out.println("C1 says hi");
        }
        //rest of class definition
}
public class C2 extends C1 {
        @Override
        public void sayHi() {
                super.sayHi();
                System.out.println("C2 says hi");
        }
        //rest of class definition
}
```

Inheritance: Overriding methods of the superclass

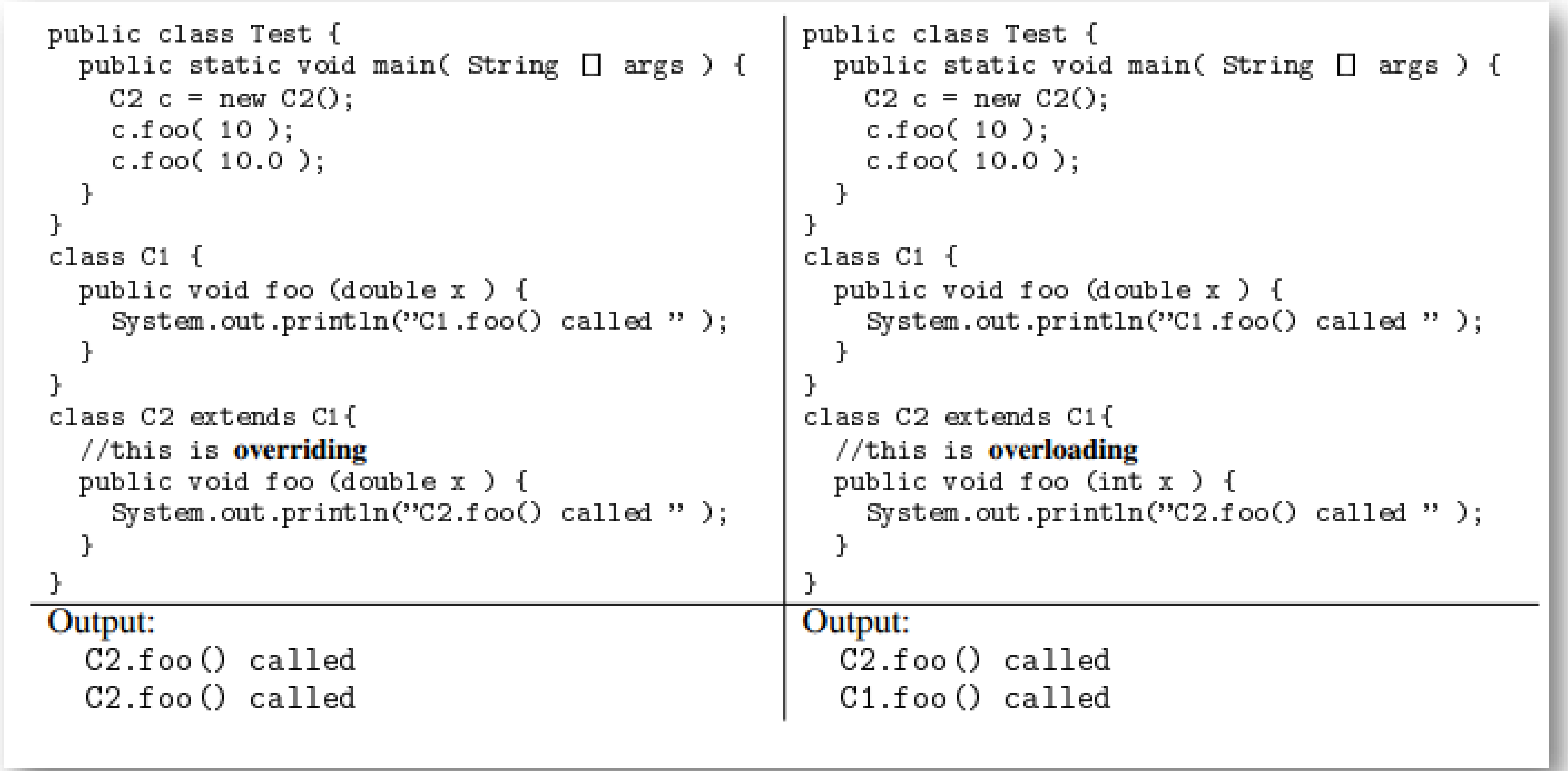
- ❖The class C2 overrides the inherited method **sayHi()** and uses the super keyword to access the overridden method (in this case the keyword super is not optional).
- ❖An overridden method of a superclass can be accessed using the super keyword.
- ❖Static methods of the superclass are not overridden.
- ❖Use the override annotation, @Override, when declaring methods that override methods of the superclass. This way, the java compiler double checks that our method truly overrides the method of the superclass.

Inheritance: Overriding vs Overloading

The method in a subclass does not match the method of its superclass exactly in its parameter list, then it only overloads the method of the superclass.

- See Example on next slide

Inheritance: Overriding vs Overloading



```
public class Test {
  public static void main( String [] args ) {
    C2 c = new C2();
    c.foo( 10 );
    c.foo( 10.0 );
  }
}
class C1 {
  public void foo (double x ) {
    System.out.println(”C1.foo() called ” );
  }
}
class C2 extends C1{
  //this is overriding
  public void foo (double x ) {
    System.out.println(”C2.foo() called ” );
  }
}
```

Output:

```
  C2.foo() called
  C2.foo() called
```

```
public class Test {
  public static void main( String [] args ) {
    C2 c = new C2();
    c.foo( 10 );
    c.foo( 10.0 );
  }
}
class C1 {
  public void foo (double x ) {
    System.out.println(”C1.foo() called ” );
  }
}
class C2 extends C1{
  //this is overloading
  public void foo (int x ) {
    System.out.println(”C2.foo() called ” );
  }
}
```

Output:

```
  C2.foo() called
  C1.foo() called
```

Object Class and its Methods

- Every class in Java inherits automatically from the **Object** class.
- This class has several methods that every other class inherits, most of which are not interesting. For more information, see the documentation from Oracle.

  https://docs.oracle.com/javase/tutorial/

- toString(), a commonly used method, is inherited from the Object class. When you write your own version of this method, you are overriding the one in the Object class.

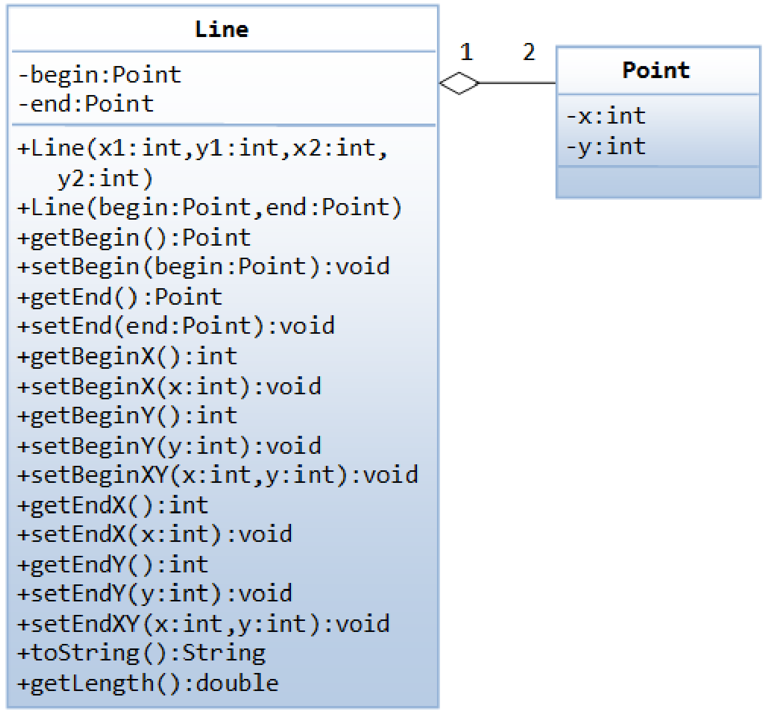
Inheritance vs. Composition

- There are two ways of reusing existing classes: *composition* and *inheritance*
- Composition exhibits a "*has-a*" relationship
- Inheritance represents the "***is a***" relationship between data types
- Example: Course (to previous example), a course has a teacher and students

Inheritance vs. Composition - example

we can re-use the Point class via *composition*.

We say that "A line is *composed* of two points", or "A line *has* two points"



Polymorphism

- ❖The term polymorphism means “having many forms”
- ❖A polymorphic reference is a variable that can refer to different types of objects at different points in time
- ❖The method invoked through a polymorphic reference can change from one invocation to the next
- ❖All object references in Java are potentially polymorphic
- ❖Polymorphism ensures that the appropriate method is called for an object of a specific type when the object is disguised as a more general type.
- ❖Polymorphism is already supported in Java — all you have to do is use it properly.

Polymorphism

- ❖Refers to the ability to process objects differently depending on their data type or class. More specifically, it is the ability to redefine methods for derived classes.
- ❖**Dynamic binding:** A method that can be implemented in several classes along the inheritance chain. The JVM (Java Virtual Machine) decides which method will run at runtime.
- ❖A variable has two types associated with it:

  Declared type – the type listed in the variable declaration

  Actual type - the type of object that variable references
- ❖The method invoked by a variable at runtime is determined by its actual type. However, the compiler only determines the appropriateness of method calls based on the declared type.

Polymorphism: Casting and instanceof Operator

Person p = new Student (…);

In this example, we are seeing **implicit casting**. It is done automatically because an instance of the class **Student** is an instance of the class **Person**.

On the other hand:

```
    Person s = new Student ( … );
    Student s = p;
```

This example will cause a compiler error because it only uses the declared type of variable p. Since Person is not a student, it cannot perform the assignment.

Polymorphism: Casting and *instanceof* Operator

In this case, p references a **Student** object, but it might not always be the case (and the compiler can't know that). The way around it is through explicit casting:

```
Person s = new Student ( … );
Student s = (Student) p;
```

The instanceof operator allows you to make sure that the particular variable references the object that is of a particular type:

refVariable instanceof ClassName

This evaluates to true if refVariable references an instance of class ClassName, and false otherwise.

Interfaces in Java

- An interface is a reference type, similar to a class, that can contain only constants, method signatures, and nested types.
- There are no method bodies.
- Interfaces cannot be instantiated
- They can only be implemented by classes or extended by other interfaces.

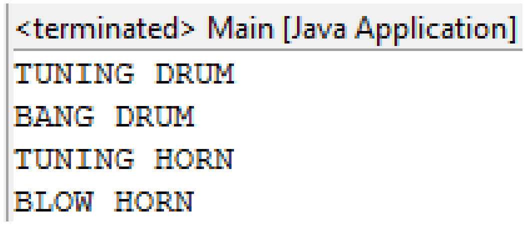
Example

```java
public interface Instrument {
    public void produceSound();
    public void tuneInstrument();
}
```

```java
public class Horn implements Instrument {
    public void produceSound(){
        System.out.println("BLOW HORN");
    }

    public void tuneInstrument() {
        System.out.println("TUNING HORN");
    }
}
```

```java
public class Drum implements Instrument {
    public void produceSound(){
        System.out.println("BANG DRUM");
    }

    public void tuneInstrument(){
        System.out.println("TUNING DRUM");
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Instrument two, three;

        two = new Drum();
        three = new Horn();

        two.tuneInstrument();
        two.produceSound();
        three.tuneInstrument();
        three.produceSound();
    }
}
```



```
<terminated> Main [Java Application]
TUNING DRUM
BANG DRUM
TUNING HORN
BLOW HORN
```

Comparable Interface in Java

- ❖An interface is a class-like construct that contains only constants and abstract methods.
- ❖Note that all data fields have to be public static final if you have any in your interface.
- ❖All methods have to be public abstract.
- ❖ Note that you can omit the access modifiers in definitions of interfaces.
- ❖ Example: Comparable interface provided by Java [1]

  A class that implements Comparable interface has to provide *compareTo()*. Note that this is the only requirement of the Comparable interface.

  Required Reading:
  [1] https://docs.oracle.com/javase/tutorial/collections/interfaces/order.html

Comparable Interface in Java

```
// This interface is defined in
// java.lang package
package java.lang;

public interface Comparable<E> {
  public int compareTo(E o);
}
```

Integer and BigInteger Classes

```java
public class Integer extends Number
    implements Comparable<Integer> {
  // class body omitted

  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}
```

```java
public class BigInteger extends Number
    implements Comparable<BigInteger> {
  // class body omitted

  @Override
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}
```

String and Date Classes

```java
public class String extends Object
    implements Comparable<String> {
  // class body omitted

  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}
```

```java
public class Date extends Object
    implements Comparable<Date> {
  // class body omitted

  @Override
  public int compareTo(Date o) {
    // Implementation omitted
  }
}
```

Integer and BigInteger Classes

```
System.out.println(new Integer(3).compareTo(new Integer(5)));
System.out.println("ABC".compareTo("ABE"));
java.util.Date date1 = new java.util.Date(2013, 1, 1);
java.util.Date date2 = new java.util.Date(2012, 1, 1);
System.out.println(date1.compareTo(date2));
```

Example Credits: Liang, Introduction to Java Programming, 10th Edition, Pearson Education

Abstract Classes in Java

- ❖An abstract class contains abstract methods
- ❖You can think of these as method-placeholders
- ❖Abstract methods are implemented by concrete subclasses
- ❖ Reason for using abstract classes:
  - ▪ Provide base/superclass that guarantees that all subclasses provide certain methods (subclasses) have to implement abstract methods of its superclass
  - ▪ Provides the ability to write more “generic”

Abstract Classes vs. Interfaces in Java

- An interface is a class-like construct that contains only constants and abstract methods.
- Note that all data fields have to be public static final if you have any in your interface.
- All methods have to be public abstract.
- Note that you can omit the access modifiers in definitions of interfaces. All data fields are public final static and all methods are public abstract in an interface. For this reason, these modifiers can be omitted, as shown below

```
public interface T1 {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

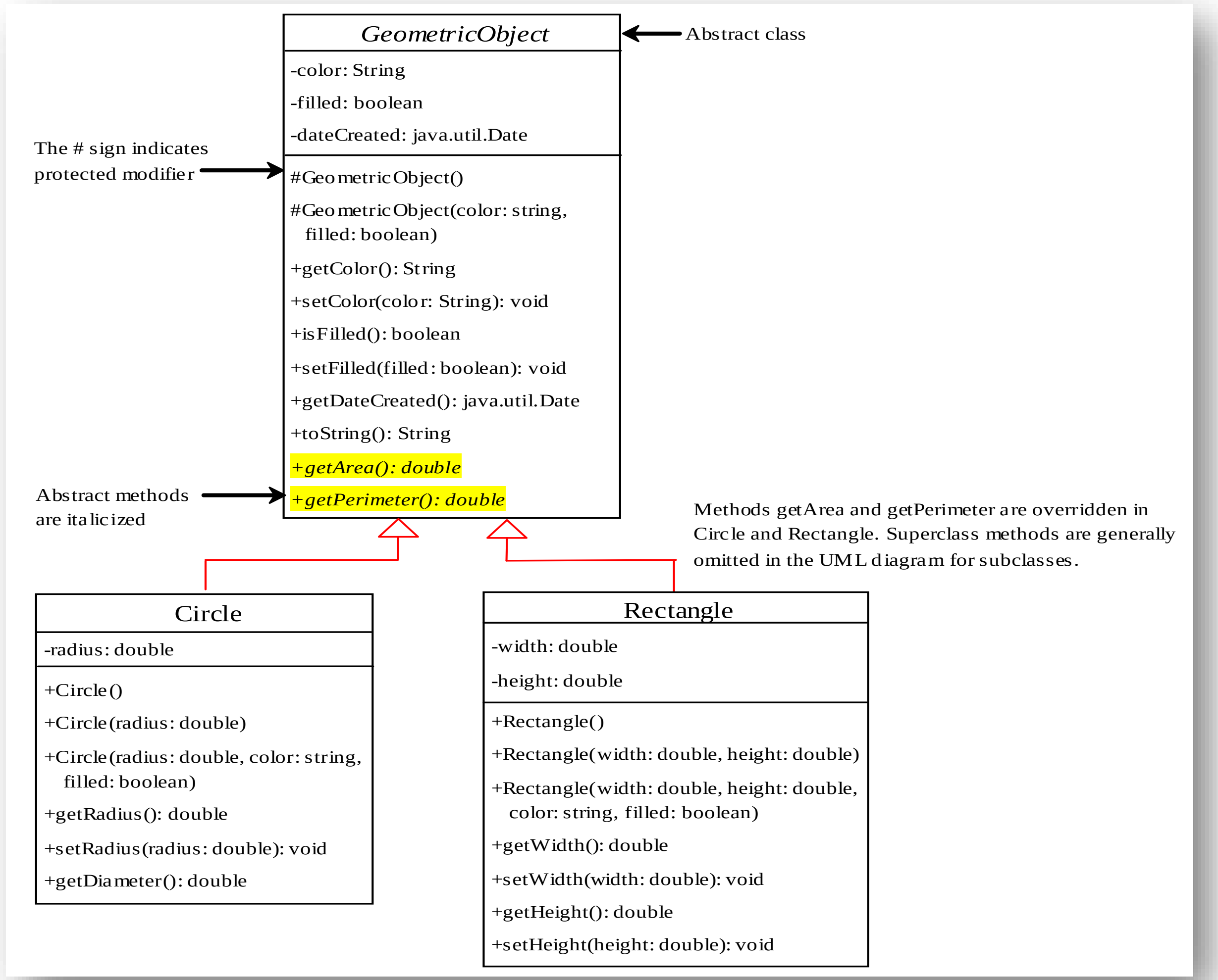
```
public interface T1 {
  int K = 1;

  void p();
}
```

Abstract Classes and Abstract Methods

- ❖ An abstract method cannot be contained in a nonabstract 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 nonabstract subclass extended from an abstract class, all the abstract methods must be implemented, even if they are not used in the subclass.
- ❖ 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. (example on next slide)

Abstract class and constructors

The constructors of Geometric Object (abstract class) are invoked in the Circle class and the Rectangle class.



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
+getArea(): double
+getPerimeter(): double
Abstract class
The # sign indicates protected modifier
Abstract methods are italicized
Methods getArea and getPerimeter are overridden in Circle and Rectangle. Superclass methods are generally omitted in the UML diagram for subclasses.
Circle
-radius: double
+Circle()
+Circle(radius: double)
+Circle(radius: double, color: string, filled: boolean)
+getRadius(): double
+setRadius(radius: double): void
+getDiameter(): double
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

Example Credits: Liang, Introduction to Java Programming, 10th Edition, Pearson Education

Abstract Class as a Data Type

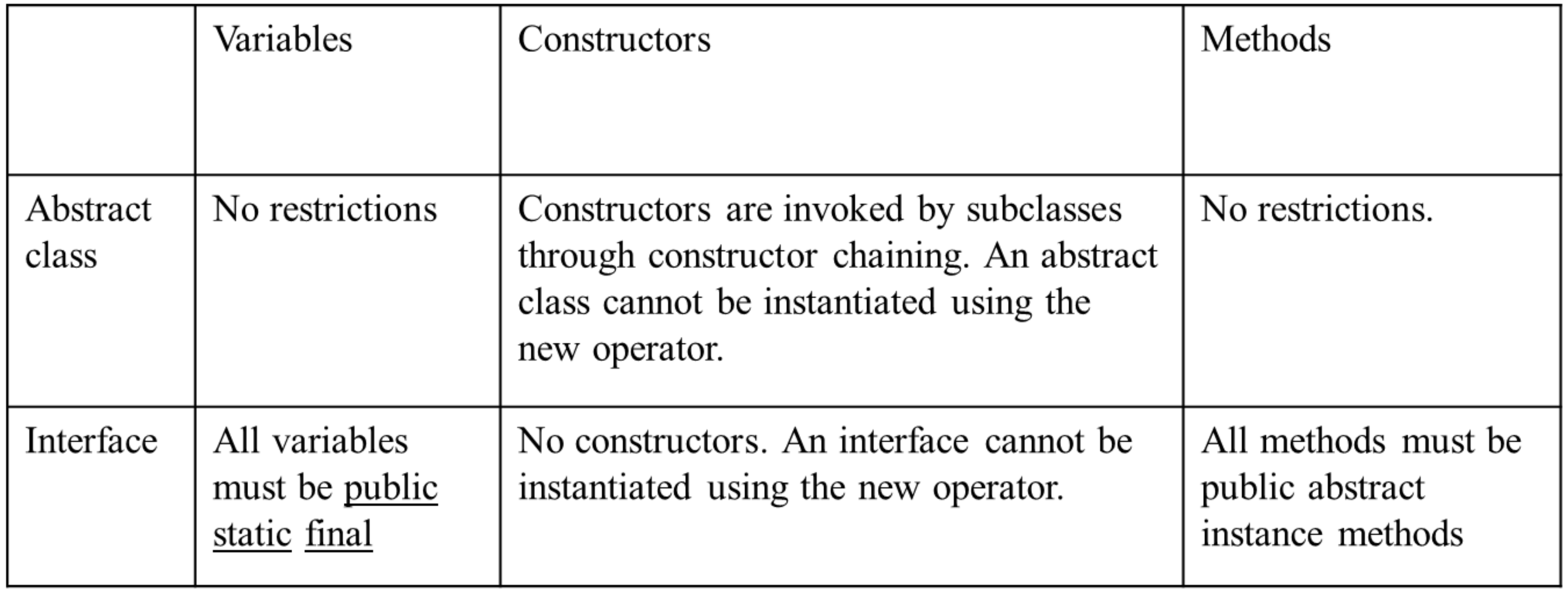
- You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type. Therefore, the following statement, which creates an array whose elements are of GeometricObject type, is correct.

```
GeometricObject[] geo = new    GeometricObject[10];
```

Example Credits: Liang, Introduction to Java Programming, 10th Edition, Pearson Education

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 public static final	No constructors. An interface cannot be instantiated using the new operator.	All methods must be public abstract instance methods

Example Credits: Liang, Introduction to Java Programming, 10th Edition, Pearson Education

Whether to use an interface or a class?

- Abstract classes and interfaces can both be used to model common features.
- How do you decide whether to use an interface or a class?
- In general, a strong is-a relationship that clearly describes a parent-child relationship should be modeled using classes. For example, a staff member is a person.
- So their relationship should be modeled using class inheritance.
- A weak is-a relationship, also known as an is-kind-of relationship, indicates that an object possesses a certain property. A weak is-a relationship can be modeled using interfaces. For example, all strings are comparable, so the String class implements the Comparable interface. You can also use interfaces to circumvent single inheritance restriction if multiple inheritance is desired. In the case of multiple inheritance, you have to design one as a superclass, and others as interface (Java does not implement multiple inheritance)

Polymorphism Example

```java
public interface Instrument {
    public String getName();
    public void setName(String name);
    public void produceSound();
    public void tuneInstrument();
}
```

```java
public abstract class BrassInstrument implements Instrument {
    private String name;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;

    }
    public BrassInstrument(String name){
        this.name = name;
    }

    public abstract void produceSound();
    public abstract void tuneInstrument();
}
```

```java
public class Horn extends BrassInstrument {
    public Horn() {
        super("Horn");
    }

    public void produceSound() {
        System.out.println("Blow Horn");
    }

    public void tuneInstrument() {
        System.out.println("Tighten Brass");
    }
}
```

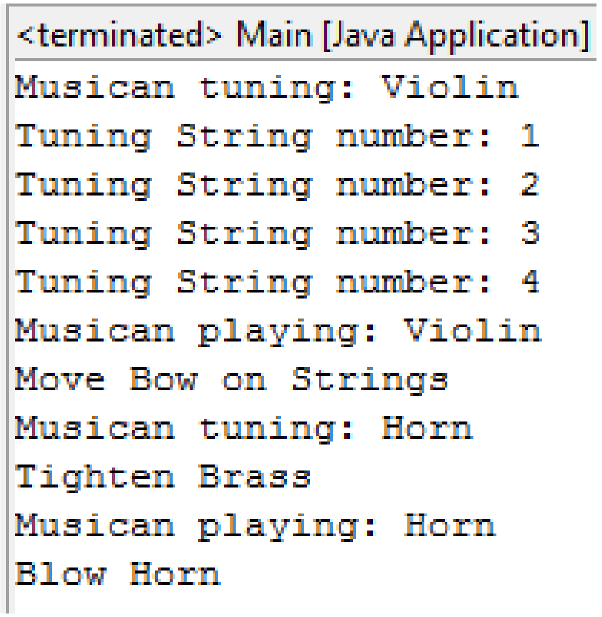
```java
public abstract class StringInstrument implements Instrument {
    private String name;
    private int numberOfStrings;

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;

    }

    public void tuneInstrument() {
        for (int x = 1; x <= numberOfStrings; x++) {
            System.out.println("Tuning String number: " + x);
        }
    }

    public StringInstrument(String name, int numberOfStrings) {
        this.name = name;
        this.numberOfStrings = numberOfStrings;
    }

    public abstract void produceSound();
}
```

```java
public class Violin extends StringInstrument {

    public Violin() {
        super("Violin", 4);
    }

    public void produceSound() {
        System.out.println("Move Bow on Strings");
    }
}
```

Polymorphism Example (cont'd)

```java
public class Musician {

    void tuneInstrument(Instrument instrument){
        System.out.println("Musican tuning: " + instrument.getName());
        instrument.tuneInstrument();
    }
    void playInstrument(Instrument instrument){
        System.out.println("Musican playing: " + instrument.getName());
        instrument.produceSound();
    }
}
```

```java
public class Main {
    public static void main(String[] args) {
        Instrument violin, horn;
        violin = new Violin();
        horn = new Horn();

        Musician musician = new Musician();
        musician.tuneInstrument(violin);
        musician.playInstrument(violin);

        musician.tuneInstrument(horn);
        musician.playInstrument(horn);
    }
}
```



```
<terminated> Main [Java Application]
Musican tuning: Violin
Tuning String number: 1
Tuning String number: 2
Tuning String number: 3
Tuning String number: 4
Musican playing: Violin
Move Bow on Strings
Musican tuning: Horn
Tighten Brass
Musican playing: Horn
Blow Horn
```

NYU

Courant Institute of Mathematical Sciences
Department of Computer Science
CS101 Introduction to Computer Science

Anasse Bari, Ph.D.

End of Chapter#11: Main Pillars of the Object Oriented Programming Paradigm