Advanced Programming Language

MSc. Nguyen Cao Dat
dat@hcmut.edu.vn

Module II

JAVA OBJECT ORIENTED PROGRAMMING

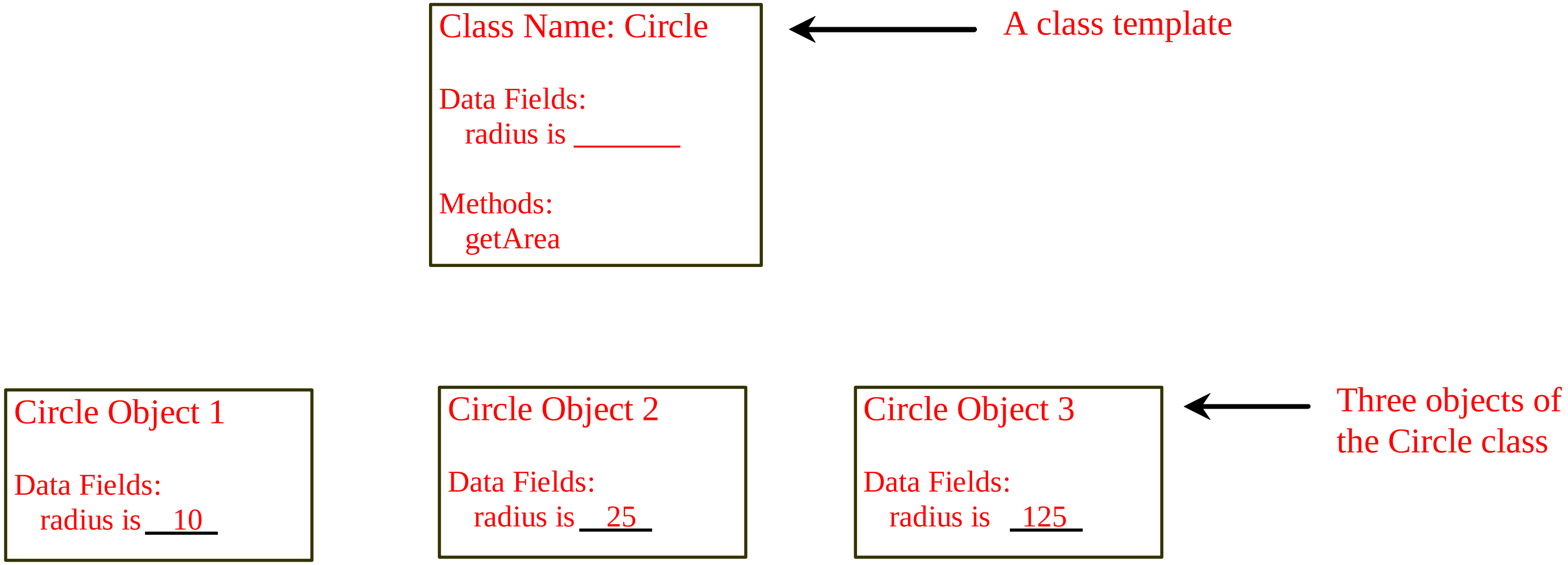
Content

- Introduction
- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

Introduction

- Object Oriented Programming (OOP) involves programming using objects
- An *object* represents an entity in the real world that can be distinctly identified. For example, a student, a desk, a circle, a button, ..
- An object has a unique identity, state, and behaviors.

Objects



Class Name: Circle

Data Fields:
radius is _______

Methods:
getArea

A class template

Circle Object 1

Data Fields:
radius is 10

Circle Object 2

Data Fields:
radius is 25

Circle Object 3

Data Fields:
radius is 125

Three objects of the Circle class

- An object has both states and behaviors.
- The state defines the object, and the behavior defines what the object does.

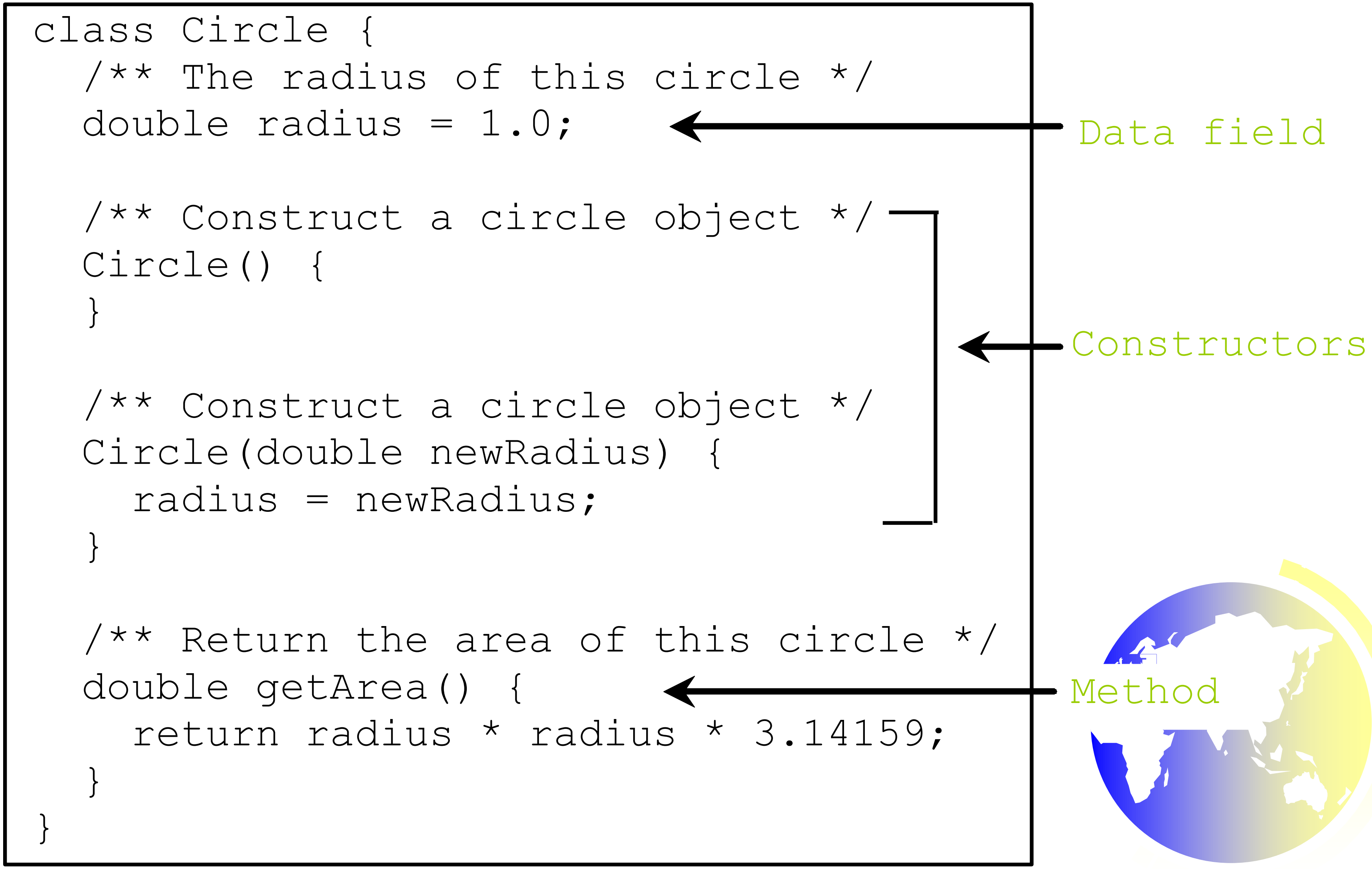
Classes

- *Classes* are constructs that define objects of the same type.

- A Java class uses variables to define data fields and methods to define behaviors.

- A class provides a special type of methods, known as constructors, which are invoked to construct objects from the class.

Classes



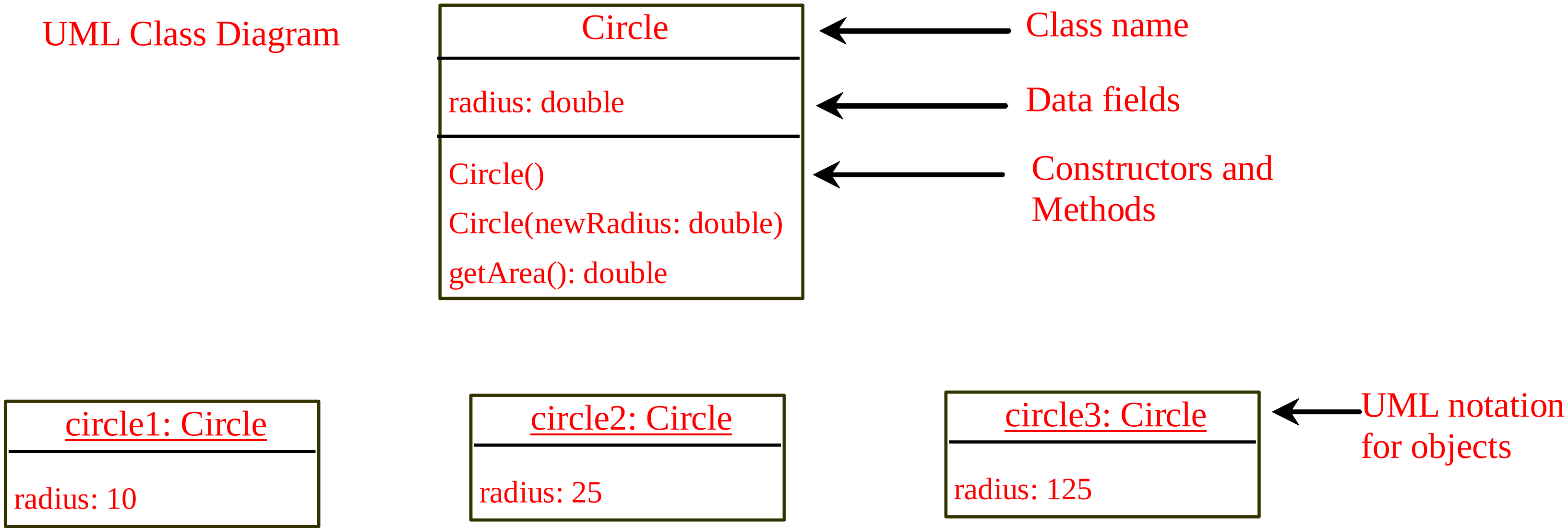
```
class Circle {
  /** The radius of this circle */
  double radius = 1.0;

  /** Construct a circle object */
  Circle() {
  }

  /** Construct a circle object */
  Circle(double newRadius) {
    radius = newRadius;
  }

  /** Return the area of this circle */
  double getArea() {
    return radius * radius * 3.14159;
  }
}
```

Data field

Constructors

Method

UML Class Diagram



UML Class Diagram

Circle

radius: double

Circle()

Circle(newRadius: double)

getArea(): double

Class name

Data fields

Constructors and Methods

circle1: Circle

radius: 10

circle2: Circle

radius: 25

circle3: Circle

radius: 125

UML notation for objects

Constructors

```
Circle() {
}
```

Constructors are a special kind of methods that are invoked to construct objects.

```
Circle(double newRadius) {
  radius = newRadius;
}
```

Constructors, cont.

- A constructor with no parameters is referred to as a *no-arg constructor*.
- Constructors must have the same name as the class itself.
- Constructors do not have a return type—not even void.
- Constructors are invoked using the new operator when an object is created.
- Constructors play the role of initializing objects.

Creating Objects Using Constructors

```
new ClassName();
```

Example:

```
new Circle();

new Circle(5.0);
```

Default Constructor

A class may be declared without constructors. In this case, a no-arg constructor with an empty body is implicitly declared in the class. This constructor, called *a default constructor*, is provided automatically *only if no constructors are explicitly declared in the class*.

Declaring Object Reference Variables

To reference an object, assign the object to a reference variable.

To declare a reference variable, use the syntax:

```
ClassName objectRefVar;
```

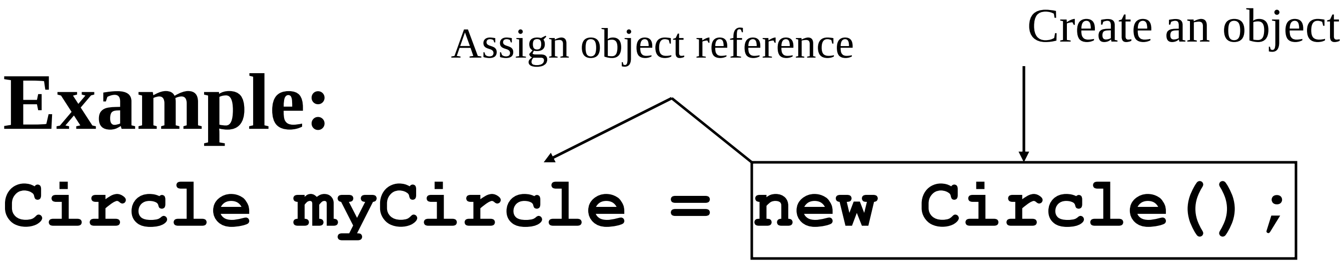
Example:

```
Circle myCircle;
```

Declaring/Creating Objects in a Single Step

```
ClassName objectRefVar = new ClassName();
```

Example:



Assign object reference

Create an object

```
Circle myCircle = new Circle();
```

Accessing Objects

- ☞ **Referencing the object's data:**

  `objectRefVar.data`

  e.g., **`myCircle.radius`**

- ☞ **Invoking the object's method:**

  `objectRefVar.methodName(arguments)`

  e.g., **`myCircle.getArea()`**

Accessibility Options

Four accessibility options:

- public
- protected *
- (default) = “package” **
- private

* protected is also accessible by package

** called also “package-private” or “package-friendly”

Example:

```
public class Person {
    private String name;
    protected java.util.Date birthDate;
    String id; // default accessibility = package
    public Person() {}
}
```

Static

Static member can be accessed without an instance

Called sometimes “class variable” as opposed to “instance variable”

Example:

```
public class Widget {
    static private int counter;
    static public getCounter() {return counter;}
}

int number = Widget.getCounter();
```

The ‘this’ keyword

In Java ‘this’ is a reference to myself
(in C++ it is a pointer…)

Example:

```
public class Point {
    private int x, y;
    public Point(int x, int y) {
       this.x = x;
       this.y = y;
    }
}
```

Defining constants

Though **const** is a reserved word in Java it's actually not in use!

However the **final** keyword let's you define constants and const variables

Example:

```
public class Thingy {
   public final static doodad = 6;  // constant
   public final id;  // constant variable
   public Thingy(int id) {this.id = id;} // OK
   // public set(int id) {this.id = id;} // error!
}
```

Questions?

Content

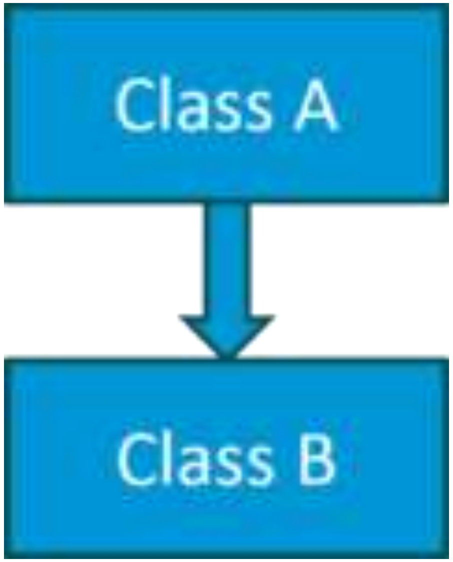
- Introduction
- Inheritance
- Encapsulation
- Polymorphism
- Abstraction

Inheritance

- Inheritance is one concept where the properties of one class can be inherited by the other.
- It helps to reuse the code and establish a relationship between different classes.

Example

```
Class A
{
---
}
Class B extends A {
---
}
```

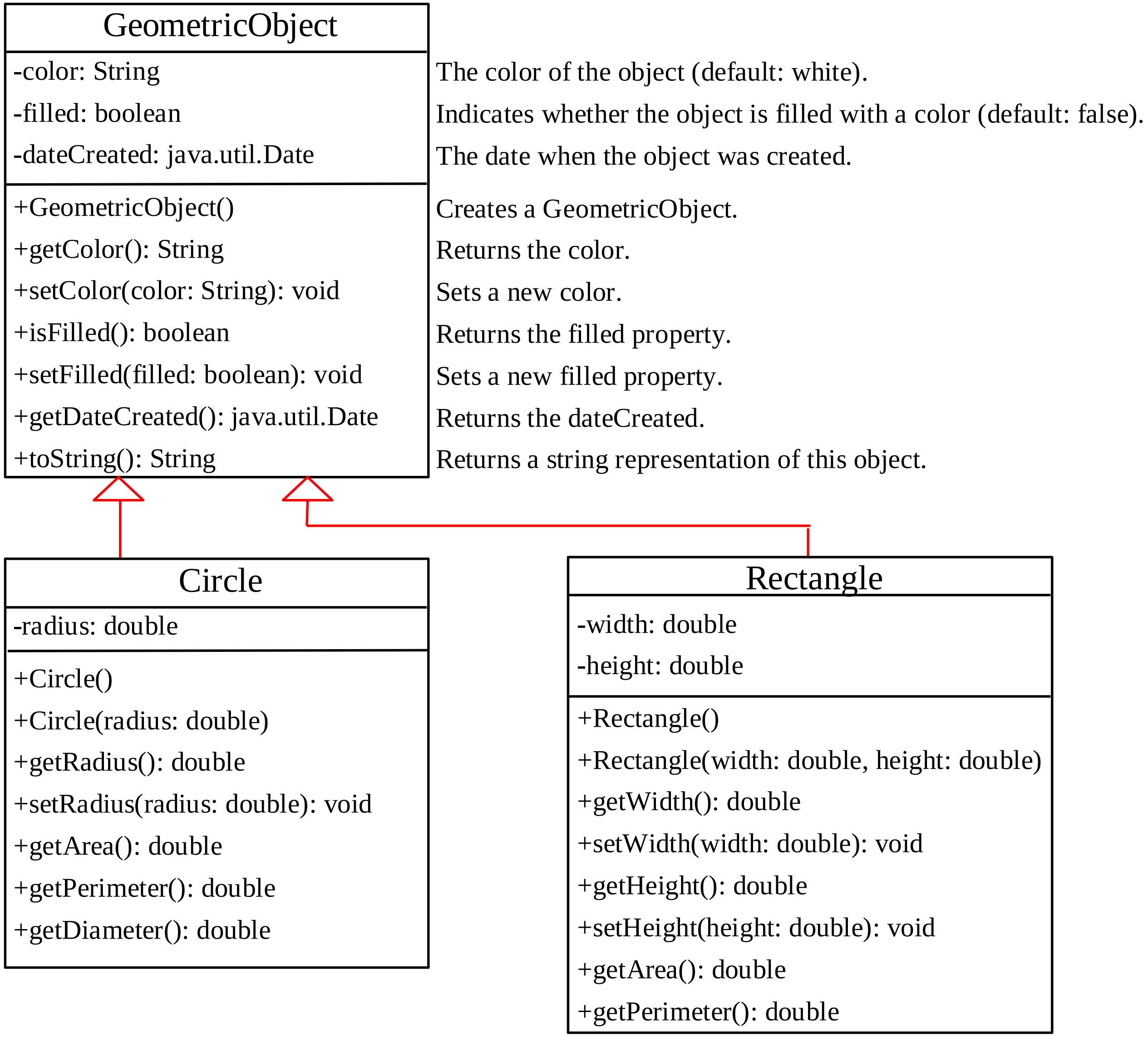


Declaring a Subclass

A subclass extends properties and methods from the superclass. You can also:

- Add new properties
- Add new methods
- Override the methods of the superclass

Superclasses and Subclasses



GeometricObject

-color: String
-filled: boolean
-dateCreated: java.util.Date

+GeometricObject()
+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.
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)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double

Rectangle

-width: double
-height: double

+Rectangle()
+Rectangle(width: double, height: double)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

Calling Superclass Methods

You could rewrite the printCircle() method in the Circle class as follows:

```
public void printCircle() {
  System.out.println("The circle is created " +
    super.getDateCreated() + " and the radius is " + radius);
}
```

Overriding Methods in the Superclass

A subclass inherits methods from a superclass. Sometimes it is necessary for the subclass to modify the implementation of a method defined in the superclass. This is referred to as *method overriding*.

```
public class Circle extends GeometricObject {
  // Other methods are omitted


  /** Override the toString method defined in GeometricObject */
  public String toString() {
    return super.toString() + "\nradius is " + radius;
  }
}
```

NOTE

- An instance method can be overridden only if it is accessible.
- Thus a private method cannot be overridden, because it is not accessible outside its own class.
- If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

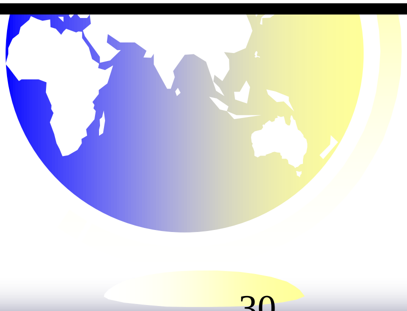
NOTE

- Like an instance method, a static method can be inherited.
- However, a static method cannot be overridden.
- If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden.

Overriding vs. Overloading

```java
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
  }
}

class B {
  public void p(int i) {
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```

```java
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
  }
}

class B {
  public void p(int i) {
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```



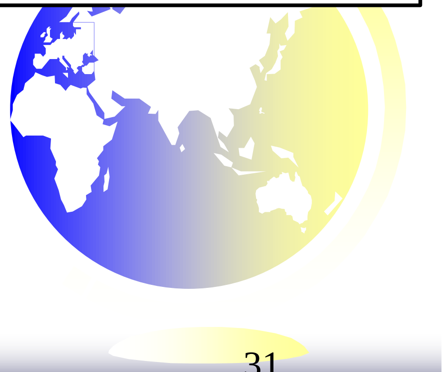
The Object Class

- Every class in Java is descended from the java.lang.Object class.
- If no inheritance is specified when a class is defined, the superclass of the class is Object.

```
public class Circle {
  ...
}
```

Equivalent

```
public class Circle extends Object {
  ...
}
```



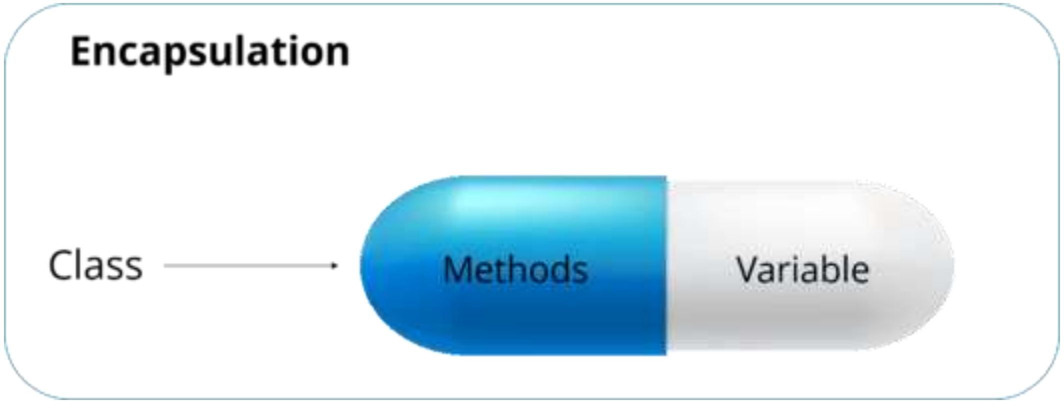
Questions?

Content

- Introduction
- Inheritance
- Encapsulation
- Polymorphism
- Abstraction

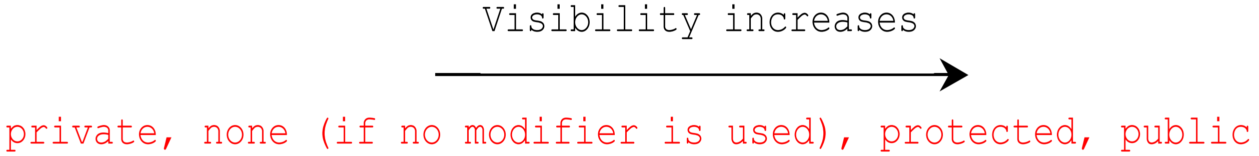
Encapsulation

- Bind data and code together as a single unit.
- Hide your data in order to make it safe from any modification.
- Encapsulation the methods and variables of a class are well hidden and safe.



The protected Modifier

- The protected modifier can be applied on data and methods in a class. A protected data or a protected method in a public class can be accessed by any class in the same package or its subclasses, even if the subclasses are in a different package.
- private, default, protected, public



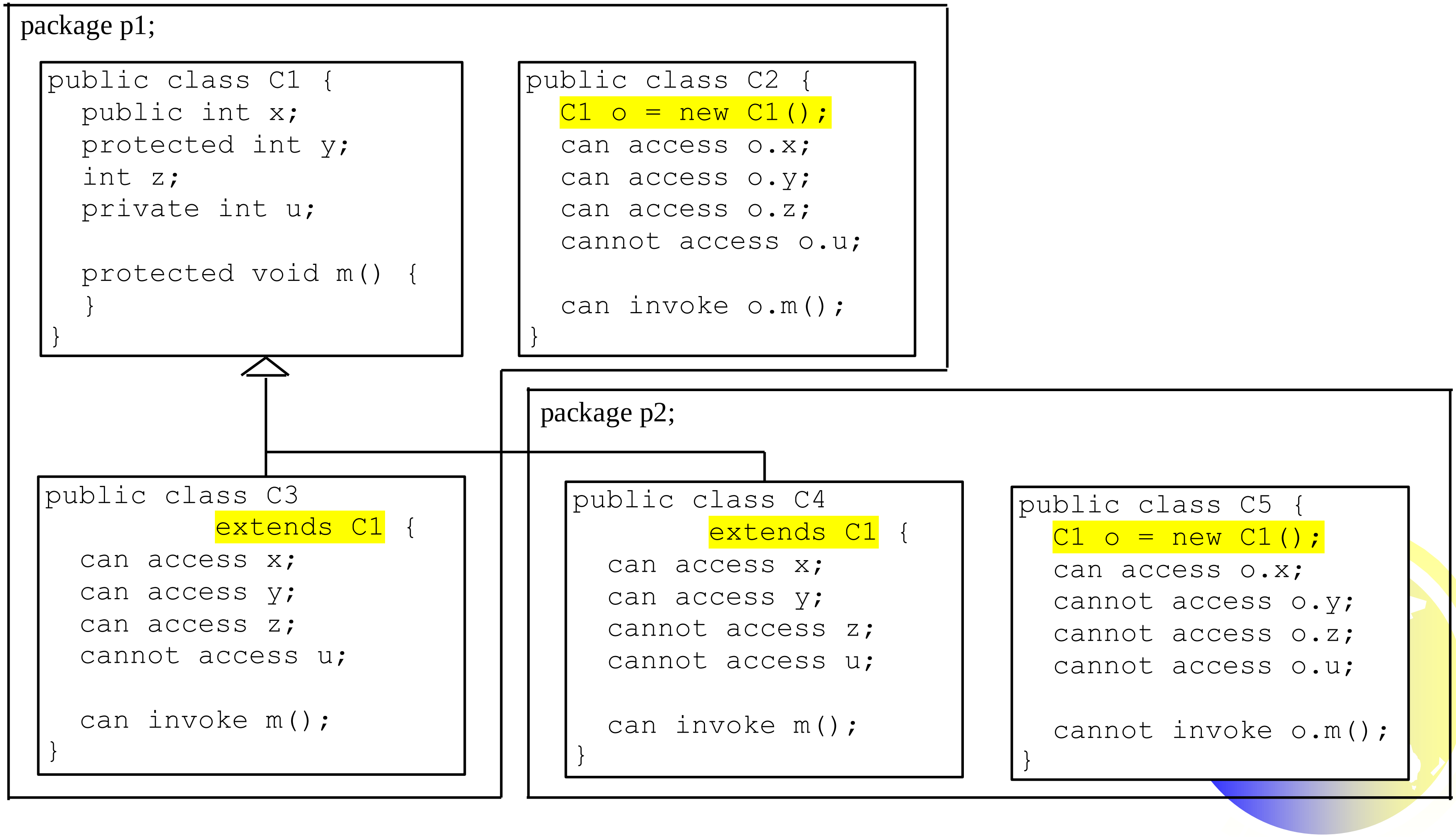
Visibility increases

private, none (if no modifier is used), protected, public

Accessibility Summary

Modifier on members in a class	Accessed from the same class	Accessed from the same package	Accessed from a subclass	Accessed from a different package
public	✓	✓	✓	✓
protected	✓	✓	✓	–
default	✓	✓	–	–
private	✓	–	–	–

Visibility Modifiers



package p1;

```
public class C1 {
  public int x;
  protected int y;
  int z;
  private int u;

  protected void m() {
  }
}
```

```
public class C2 {
  C1 o = new C1();
  can access o.x;
  can access o.y;
  can access o.z;
  cannot access o.u;

  can invoke o.m();
}
```

```
public class C3
          extends C1 {
  can access x;
  can access y;
  can access z;
  cannot access u;

  can invoke m();
}
```

package p2;

```
public class C4
          extends C1 {
  can access x;
  can access y;
  cannot access z;
  cannot access u;

  can invoke m();
}
```

```
public class C5 {
  C1 o = new C1();
  can access o.x;
  cannot access o.y;
  cannot access o.z;
  cannot access o.u;

  cannot invoke o.m();
}
```

A Subclass Cannot Weaken the Accessibility

- A subclass may override a protected method in its superclass and change its visibility to public.

- However, 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.

NOTE

- The modifiers are used on classes and class members (data and methods), except that the final modifier can also be used on local variables in a method.

- A final local variable is a constant inside a method.

The `final` Modifier

- The `final` class cannot be extended:

```
final class Math {
    ...
  }
```

- The `final` variable is a constant:

```
final static double PI = 3.14159;
```

- The `final` method cannot be overridden by its subclasses.

Questions?

Content

- **Introduction**
- **Encapsulation**
- **Inheritance**
- **Polymorphism**
- **Abstraction**

Polymorphism

- Taking many forms
- It is the ability of a variable, function or object to take on multiple forms.
- Allows define one interface or method and have multiple implementations.

Example

```
public class PolymorphismDemo {
  public static void main(String[]
     args) {
     m(new GraduateStudent());
     m(new Student());
     m(new Person());
     m(new Object());
  }
public static void m(Object x) {

    System.out.println(x.toString());
  }
}
class GraduateStudent extends
    Student {
}
class Student extends Person {
  public String toString() {
    return "Student";
  }
}
class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Method m takes a parameter of the Object type. You can invoke it with any object.

Dynamic binding

- When the method m(Object x) is executed, the argument x's toString method is invoked. x may be an instance of GraduateStudent, Student, Person, or Object. Classes GraduateStudent, Student, Person, and Object have their own implementation of the toString method.
- Which implementation is used will be determined dynamically by the Java Virtual Machine at runtime.
- This capability is known as *dynamic binding*.

Casting Objects

You have already used the casting operator to convert variables of one primitive type to another. *Casting* can also be used to convert an object of one class type to another within an inheritance hierarchy. In the preceding section, the statement

```
m(new Student());
```

assigns the object new Student() to a parameter of the Object type. This statement is equivalent to:

```
Object o = new Student(); // Implicit casting
m(o);
```

The statement Object o = new Student(), known as implicit casting, is legal because an instance of Student is automatically an instance of Object.

Casting from Superclass to Subclass

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Fruit fruit = new Fruit();

Apple x = (Apple)fruit;

Orange x = (Orange)fruit;
```

Example: Demonstrating Polymorphism and Casting

- `TestPolymorphismCasting.java`
- This example creates two geometric objects: a circle, and a rectangle, invokes the displayGeometricObject method to display the objects.
- The displayGeometricObject displays the area and diameter if the object is a circle, and displays area if the object is a rectangle.

Questions?

Content

- Introduction
- Encapsulation
- Inheritance
- Polymorphism
- Abstraction

Abstraction

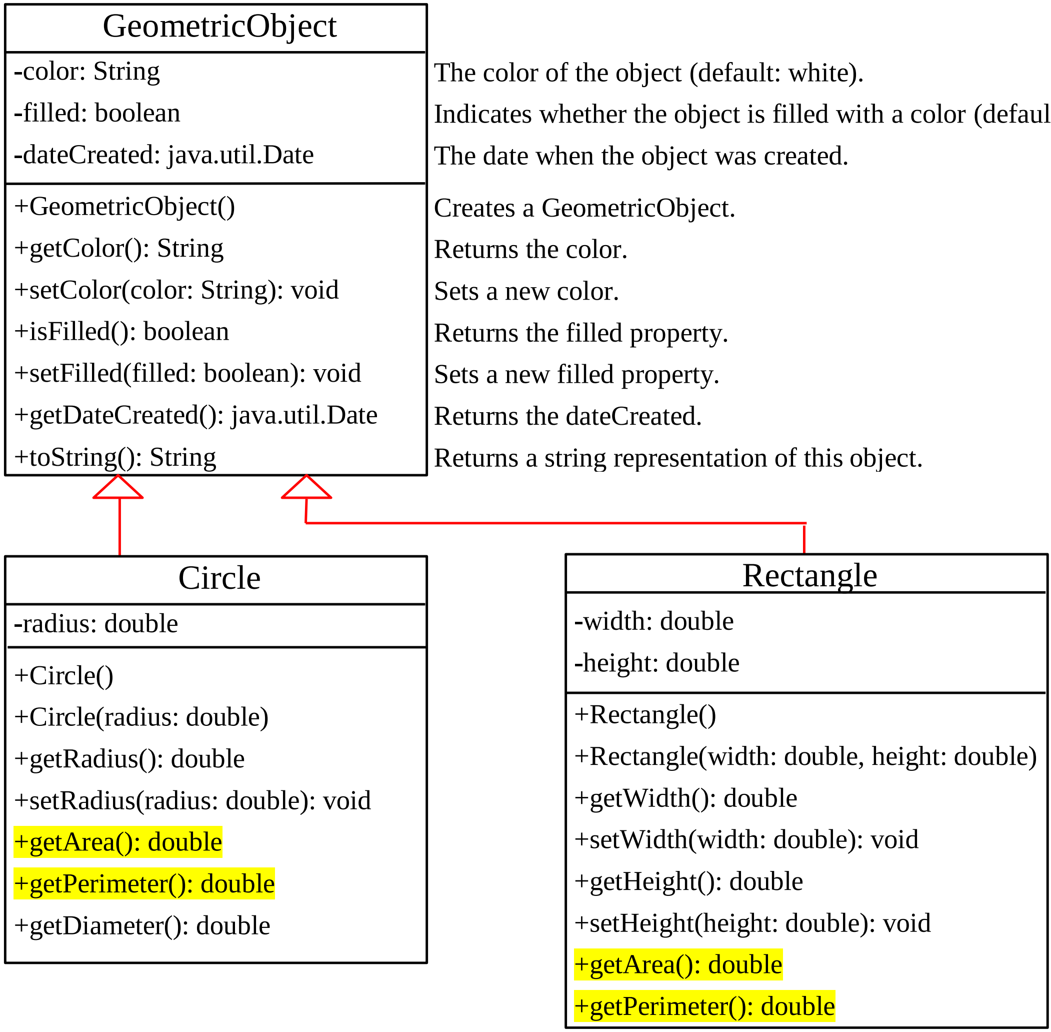
- Deals with ideas rather than events.
- Helps to reduce complexity.
- Achieves abstraction in two ways:
  - Abstract Class and Abstract Method
  - Interface

The `abstract` Modifier

- **The `abstract` class**
  - Cannot be instantiated
  - Should be extended and implemented in subclasses

- **The `abstract` method**
  - Method signature without implementation

Example



GeometricObject

-color: String	The color of the object (default: white).
-filled: boolean	Indicates whether the object is filled with a color (default: false).
-dateCreated: java.util.Date	The date when the object was created.
+GeometricObject()	Creates a GeometricObject.
+getColor(): String	Returns the color.
+setColor(color: String): void	Sets a new color.
+isFilled(): boolean	Returns the filled property.
+setFilled(filled: boolean): void	Sets a new filled property.
+getDateCreated(): java.util.Date	Returns the dateCreated.
+toString(): String	Returns a string representation of this object.

Circle

-radius: double

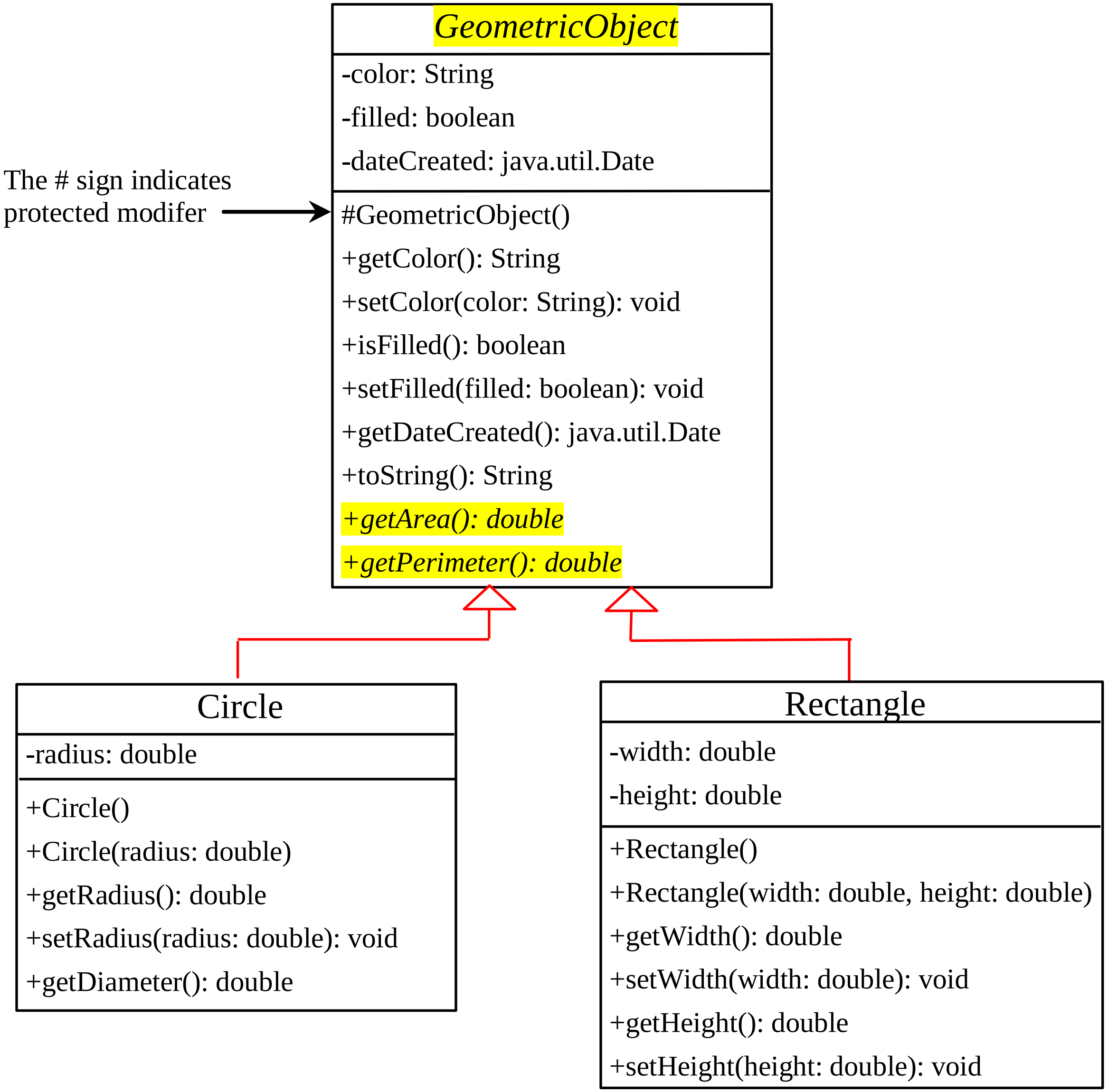
+Circle()
+Circle(radius: double)
+getRadius(): double
+setRadius(radius: double): void
+getArea(): double
+getPerimeter(): double
+getDiameter(): double

Rectangle

-width: double
-height: double

+Rectangle()
+Rectangle(width: double, height: double)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void
+getArea(): double
+getPerimeter(): double

Abstract Classes



The # sign indicates protected modifer

GeometricObject

-color: String
-filled: boolean
-dateCreated: java.util.Date

#GeometricObject()
+getColor(): String
+setColor(color: String): void
+isFilled(): boolean
+setFilled(filled: boolean): void
+getDateCreated(): java.util.Date
+toString(): String
+getArea(): double
+getPerimeter(): double

Circle

-radius: double

+Circle()
+Circle(radius: double)
+getRadius(): double
+setRadius(radius: double): void
+getDiameter(): double

Rectangle

-width: double
-height: double

+Rectangle()
+Rectangle(width: double, height: double)
+getWidth(): double
+setWidth(width: double): void
+getHeight(): double
+setHeight(height: double): void

NOTE

- An abstract method cannot be contained in a none abstract class.

- If a subclass of an abstract superclass does not implement all the abstract methods, the subclass must be declared 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.

NOTE

- 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.

Interfaces

- An *interface* is a classlike construct that contains only constants and abstract methods.
- In many ways, an interface is similar to an abstract class, but an abstract class can contain variables and concrete methods as well as constants and abstract methods.
- To distinguish an interface from a class, Java uses the following syntax to declare an interface:

```
public interface InterfaceName {
  constant declarations;
  method signatures;
}
```

Example

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

public interface Comparable {
  public int compareTo(Object o);
}
```

String and Date Classes

Many classes (e.g., String and Date) in the Java library implement Comparable to define a natural order for the objects. If you examine the source code of these classes, you will see the keyword implements used in the classes, as shown below:

```
public class String extends Object
    implements Comparable {
  // class body omitted
}
```

```
public class Date extends Object
    implements Comparable {
  // class body omitted
}
```

```
new String() instanceof String
new String() instanceof Comparable
new java.util.Date() instanceof java.util.Date
new java.util.Date() instanceof Comparable
```

Generic max Method

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum of two objects */
  public static Comparable max
      (Comparable o1, Comparable o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

(a)

```
// Max.java: Find a maximum object
public class Max {
  /** Return the maximum of two objects */
  public static Object max
      (Object o1, Object o2) {
    if (((Comparable)o1).compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}
```

(b)

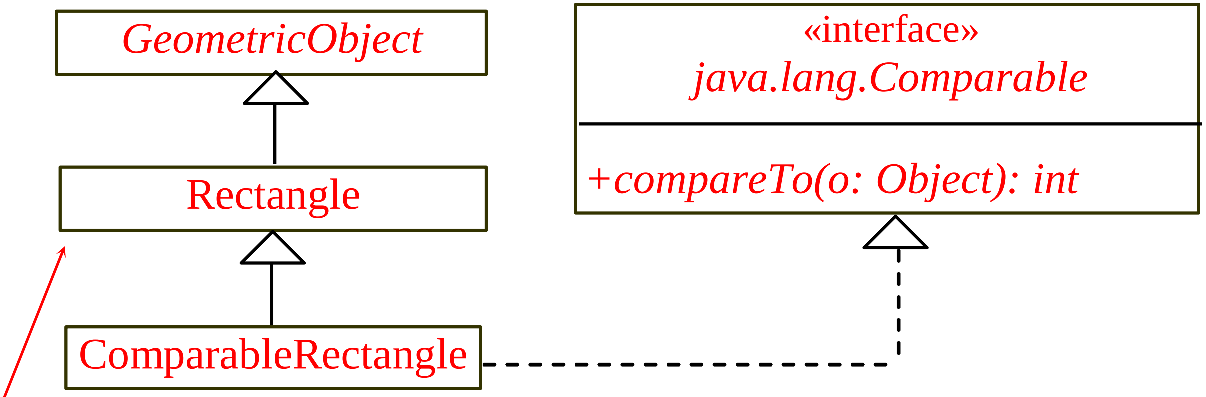
```
String s1 = "abcdef";
String s2 = "abcdee";
String s3 = (String)Max.max(s1, s2);
```

```
Date d1 = new Date();
Date d2 = new Date();
Date d3 = (Date)Max.max(d1, d2);
```

The return value from the max method is of the Comparable type. So, you need to cast it to String or Date explicitly.

Declaring Classes to Implement Comparable

Notation:
The interface name and the method names are italicized. The dashed lines and hollow triangles are used to point to the interface.



GeometricObject

Rectangle

ComparableRectangle

«interface»
java.lang.Comparable

+compareTo(o: Object): int

ComparableRectangle

You cannot use the max method to find the larger of two instances of Rectangle, because Rectangle does not implement Comparable. However, you can declare a new rectangle class that implements Comparable. The instances of this new class are comparable. Let this new class be named ComparableRectangle.

```
ComparableRectangle rectangle1 = new ComparableRectangle(4, 5);
ComparableRectangle rectangle2 = new ComparableRectangle(3, 6);
System.out.println(Max.max(rectangle1, rectangle2));
```

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

Creating Custom Interfaces

```
public interface Edible {
  /** Describe how to eat */
  public String howToEat();
}
```

```
class Animal {
}

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

class Tiger extends Animal {
}
```

```
class abstract Fruit
    implements Edible {
}

class Apple extends Fruit {
  public String howToEat() {
    return "Make apple cider";
  }
}

class Orange extends Fruit {
  public String howToEat() {
    return "Make orange juice";
  }
}
```

Implements Multiple Interfaces

```
class Chicken extends Animal implements Edible, Comparable {
  int weight;
  public Chicken(int weight) {
    this.weight = weight;
  }
  public String howToEat() {
    return "Fry it";
  }
  public int compareTo(Object o) {
    return weight – ((Chicken)o).weight;
  }
}
```

Creating Custom Interfaces, cont.

```
public interface Edible {
  /** Describe how to eat */
  public String howToEat();
}
```

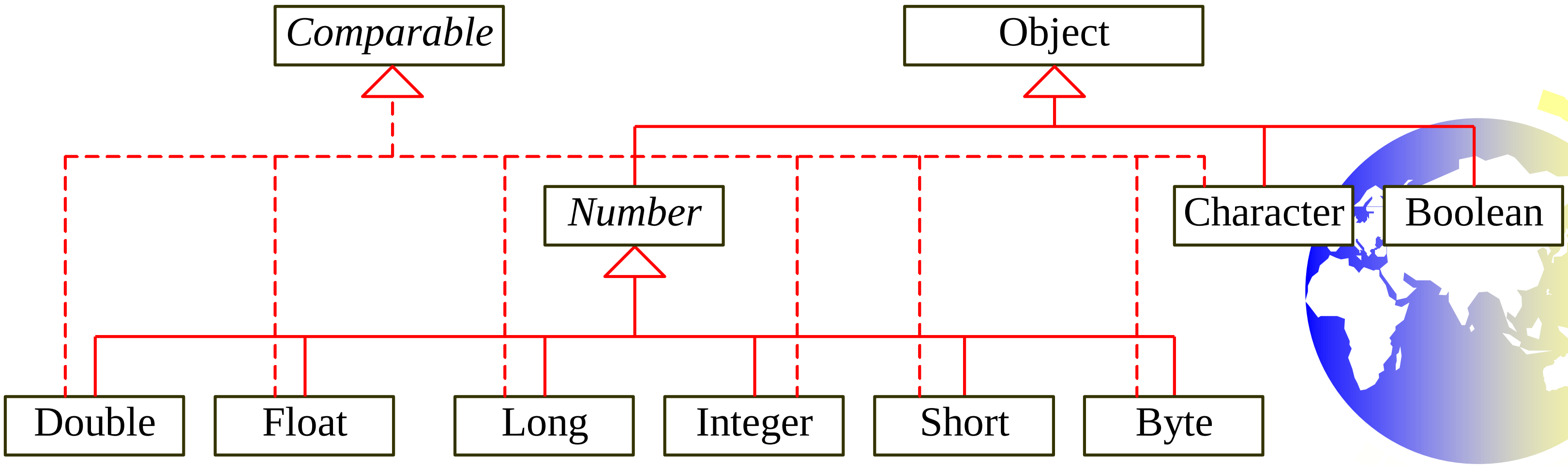
```
public class TestEdible {
  public static void main(String[] args) {
    Object[] objects = {new Tiger(), new Chicken(), new Apple()};
    for (int i = 0; i < objects.length; i++)
      showObject(objects[i]);
  }

  public static void showObject(Object object) {
    if (object instanceof Edible)
      System.out.println(((Edible)object).howToEat());
  }
}
```

Wrapper Classes

- **Boolean**
- **Character**
- **Short**
- **Byte**

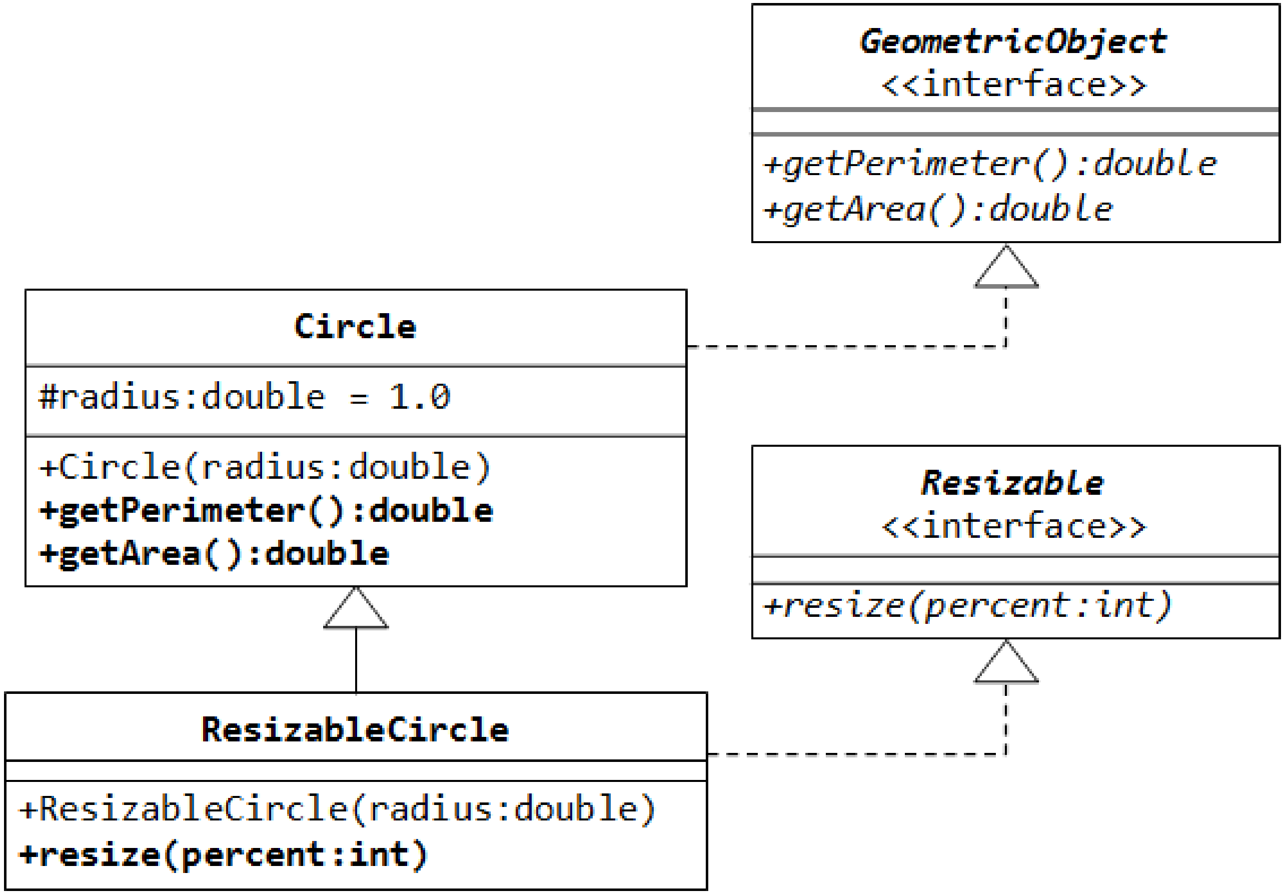
- Integer
- Long
- Float
- Double

NOTE: (1) The wrapper classes do not have no-arg constructors. (2) The instances of all wrapper classes are immutable, i.e., their internal values cannot be changed once the objects are created.



Practice Problem

- **Interfaces GeometricObject and Resizable**



GeometricObject
<<interface>>

+getPerimeter():double
+getArea():double

Circle

#radius:double = 1.0

+Circle(radius:double)
+getPerimeter():double
+getArea():double

Resizable
<<interface>>

+resize(percent:int)

ResizableCircle

+ResizableCircle(radius:double)
+resize(percent:int)

Practice Problem (cont'd)

- Write the interface called `GeometricObject`

```
public interface GeometricObject {
     public double getPerimeter();
     ......
     }
```

Practice Problem (cont’d)

☞ **Write the implementation class `Circle`**

```
public class Circle implements GeometricObject {
   // Private variable
   ......
   // Constructor
   ......
   // Implement methods defined in interface GeometricObject
   @Override
   public double getPerimeter() { ...... }
   ......
}
```

Practice Problem (cont’d)

- Write a test program called `TestCircle` to test the methods defined in `Circle`

Practice Problem (cont'd)

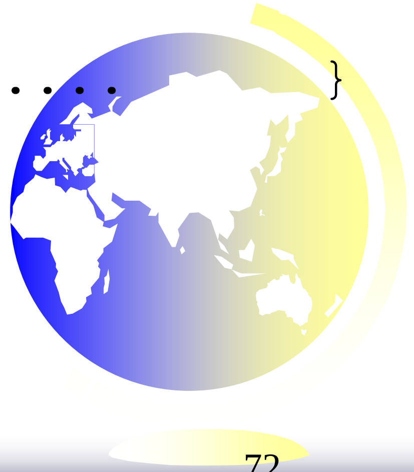
- **Write the interface `Resizable`**

```
public interface Resizable {
   public double resize(...);
}
```

Practice Problem (cont’d)

- **Write the class `ResizableCircle`**

```
public class ResizableCircle extends Circle implements
Resizeable {
   // Constructor
   public ResizableCircle(double radius) {
      super(...);
   }
   // Implement methods defined in interface Resizable
   @Override
   public double resize(int percent) { ......    }
}
```



Practice Problem (cont'd)

☞ **Write a test program called `TestResizableCircle` to test the methods defined in `ResizableCircle`**

Questions?