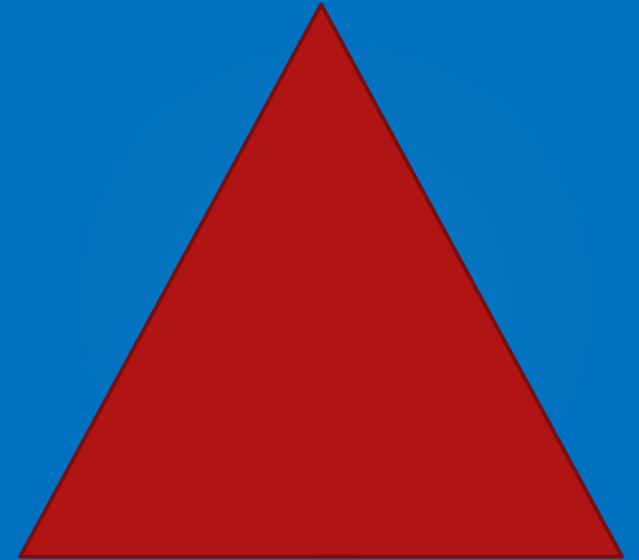
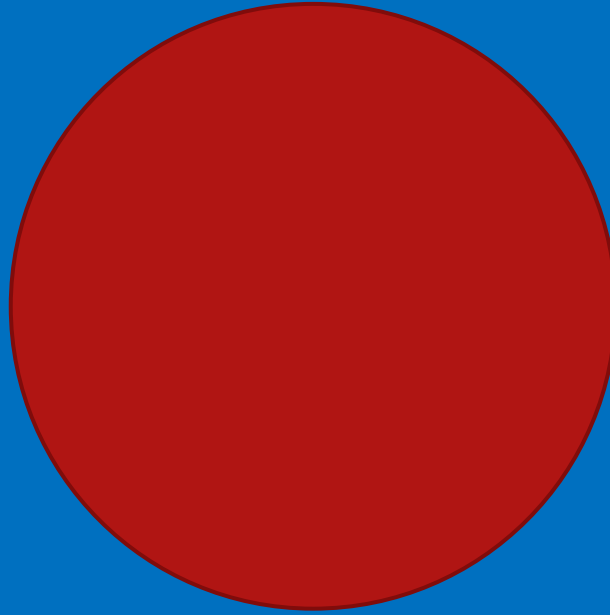


# Chapter 08 Abstract Classes and Interfaces

Case: Compare the area of two geometric objects

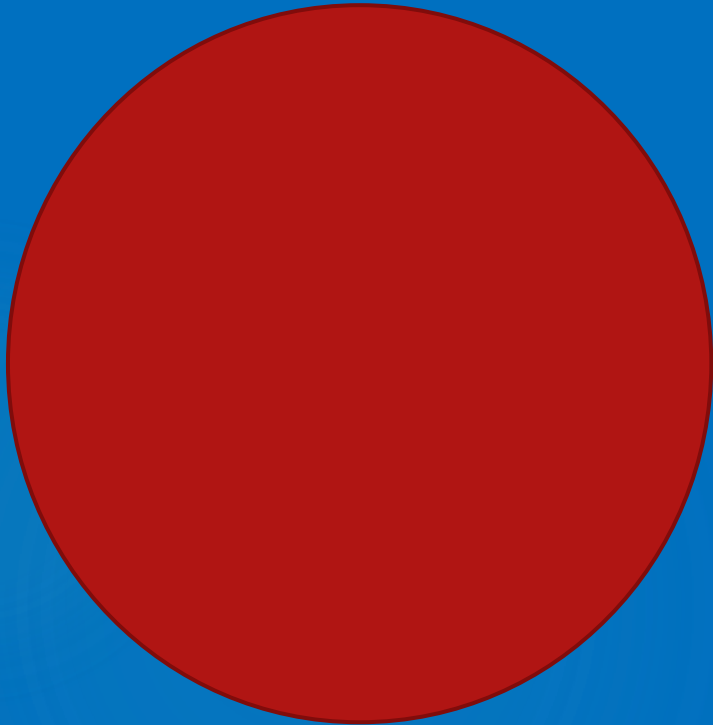
2



Which polygon is the bigger one?

# Circle Class


3



```
public class Circle{  
    private double radius;  
    public Circle(double r){  
        radius = r;  
    }  
    public void getArea(){  
        return Math.PI * radius * radius;  
    }  
}
```

# Rectangle Class

4



```
public class Rectangle{  
    private double width, height;  
    public Rectangle(double w, double h){  
        width = w;  
        height = h;  
    }  
    public void getArea(){  
        return w * h;  
    }  
}
```

# Define Comparison Method

5

```
public static void main(String[] args){
    Circle c1 = new Circle(2);
    Rectangle r1 = new Rectangle(2,2);
    System.out.println(compareArea(c1, r1));
}
public static int compareArea(Circle c, Rectangle r){
    if(c.getArea() > r.getArea()){
        return 1;
    }else if(c.getArea() < r.getArea()){
        return -1;
    }else{
        return 0;
    }
}
```

The method can only compare circle  
and rectangle

# How to Define a General Comparison Method

6

In order to define a method which can compare all kinds of polygons, a *abstract method* is needed.

```
public abstract class GeomObject{  
    public abstract double getArea();  
}  
  
public static int compareArea(GeomObject o1, GeomObject o2){}
```

# New Circle and Rectangle Classes

```
public class Circle extends GeomObject{
    public double getArea(){
        return Math.PI * radius * radius;
    }
}
public class Rectangle extends GeomObject{
    public double getArea(){
        return width * height;
    }
}
```

# New Client Program

8

```
public static void main(String[] args){
    GeomObject g1 = new Circle(2);
    GeomObject g2 = new Rectangle(2,2);
    System.out.println(compareArea(g1, g2));
}
public static int compareArea(GeomObject g1, GeomObject g2){
    if(g1.getArea() > g2.getArea()){
        return 1;
    }else if(g1.getArea() < g2.getArea()){
        return -1;
    }else{
        return 0;
    }
}
```

The method can compare arbitrary polygons



## Another Method Design

9

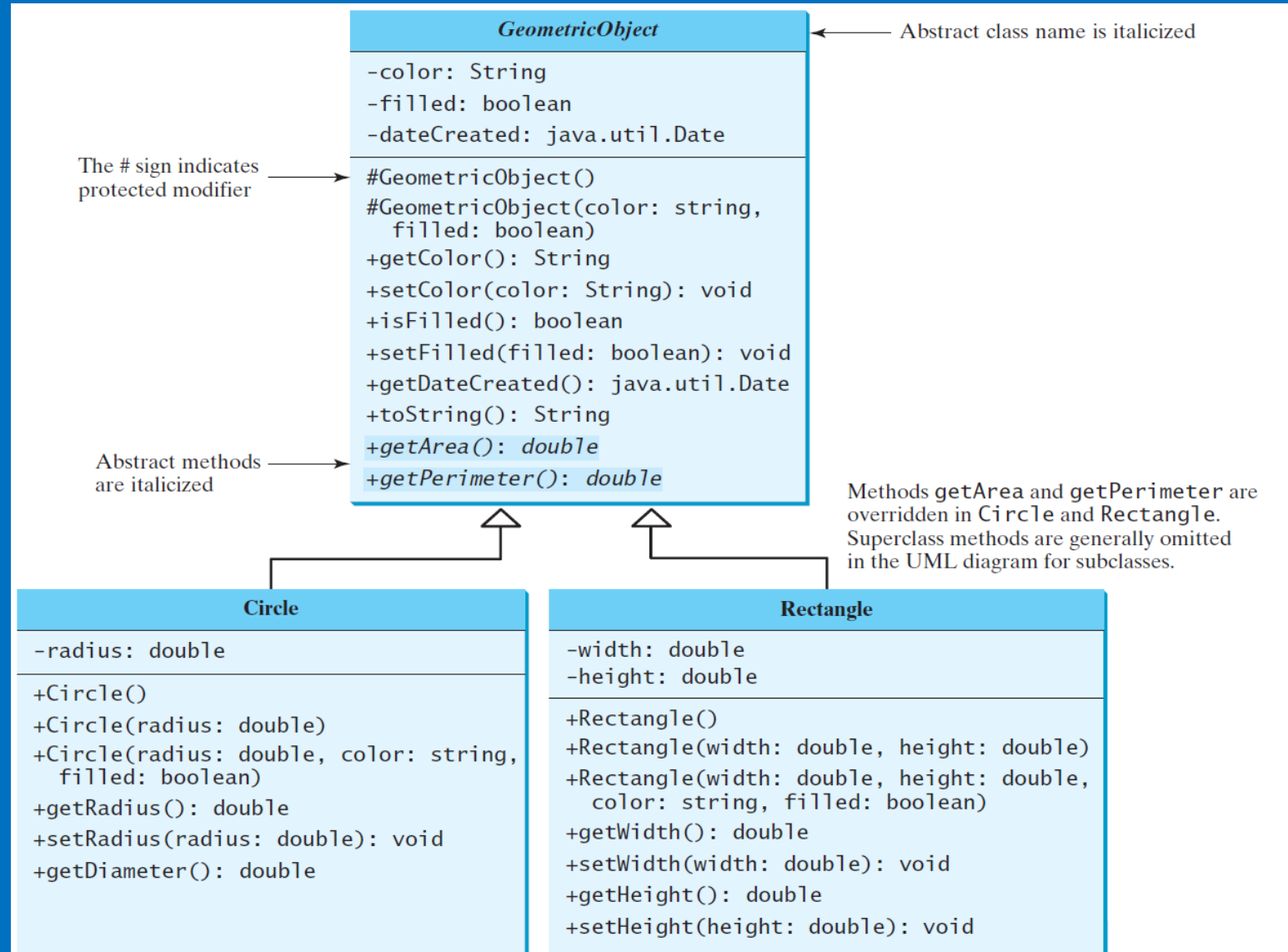
Define the comparison function inside the class.

```
public abstract class GeomObject{  
    public abstract double getArea();  
    public int compareTo(GeomObject o);  
}
```

# Abstract Classes

10

An abstract class cannot be used to create objects. An abstract class can contain abstract methods, which are implemented in concrete subclasses.



# Abstract Method in Abstract Class

11

An *abstract method cannot* be contained in a *non-abstract (concrete) 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

12

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 GeomObject are invoked in the Circle class and the Rectangle class.

## Abstract class without abstract method

13

A class that contains abstract methods must be abstract. However, it is possible to define an *abstract class* that *contains no abstract methods*. In this case, you 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

14

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

java.lang

**Class Object**

java.lang.Object

---

public class Object

Class Object is the root of the class hierarchy

## Concrete method overridden to be abstract

15

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.

```
public class A{
    public void method(){}
}
public abstract class B extends A{
    public abstract void method();
}
```

Which of the following classes defines a legal abstract class?

```
class A {  
    abstract void unfinished() {  
    }  
}
```

(a)

```
public class abstract A {  
    abstract void unfinished();  
}
```

(b)

```
class A {  
    abstract void unfinished();  
}
```

(c)

```
abstract class A {  
    protected void unfinished();  
}
```

(d)

```
abstract class A {  
    abstract void unfinished();  
}
```

(e)

```
abstract class A {  
    abstract int unfinished();  
}
```

(f)



## True or false?

- a. An abstract class can be used just like a nonabstract class except that you cannot use the *new* operator to create an instance from the abstract class.
- b. An abstract class can be extended.
- c. A subclass of a nonabstract superclass cannot be abstract.
- d. A subclass cannot override a concrete method in a superclass to define it as abstract.
- e. An abstract method must be non-static.

## Abstract class as type

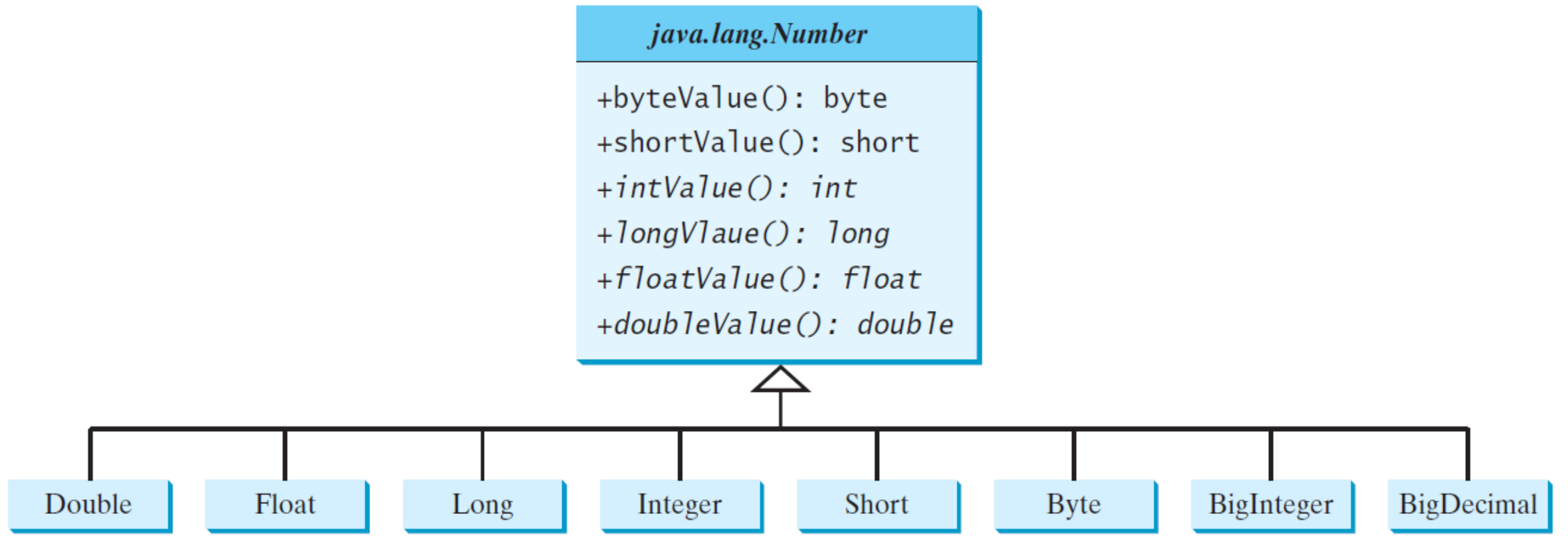
18

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.

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

# Case Study: the Abstract Number Class

19



Why do the following two lines of code compile but cause a runtime error?

```
Number numberRef = new Integer(0);  
Double doubleRef = (Double)numberRef;
```

Why do the following two lines of code compile but cause a runtime error?

```
Number[] numberArray = new Integer[2];  
numberArray[0] = new Double(1.5);
```

Show the output of the following code.

```
public class Test {  
    public static void main(String[] args) {  
        Number x = 3;  
        System.out.println(x.intValue());  
        System.out.println(x.doubleValue());  
    }  
}
```

What is wrong in the following code?

```
public class Test {  
    public static void main(String[] args) {  
        Number x = new Integer(3);  
        System.out.println(x.intValue());  
        System.out.println(x.compareTo(new Integer(4)));  
    }  
}
```

What is wrong in the following code?

```
public class Test {  
    public static void main(String[] args) {  
        Number x = new Integer(3);  
        System.out.println(x.intValue());  
        System.out.println((Integer)x.compareTo(new Integer(4)));  
    }  
}
```

# Case Study: Calendar and GregorianCalendar in JDK

24

## *java.util.Calendar*

```
#Calendar()  
+get(field: int): int  
+set(field: int, value: int): void  
+set(year: int, month: int,  
    dayOfMonth: int): void  
+getActualMaximum(field: int): int  
+add(field: int, amount: int): void  
+getTime(): java.util.Date  
  
+setTime(date: java.util.Date): void
```



## *java.util.GregorianCalendar*

```
+GregorianCalendar()  
+GregorianCalendar(year: int,  
    month: int, dayOfMonth: int)  
+GregorianCalendar(year: int,  
    month: int, dayOfMonth: int,  
    hour: int, minute: int, second: int)
```

Constructs a default calendar.

Returns the value of the given calendar field.

Sets the given calendar to the specified value.

Sets the calendar with the specified year, month, and date. The month parameter is 0-based; that is, 0 is for January.

Returns the maximum value that the specified calendar field could have.

Adds or subtracts the specified amount of time to the given calendar field.

Returns a **Date** object representing this calendar's time value (million second offset from the UNIX epoch).

Sets this calendar's time with the given **Date** object.

Constructs a **GregorianCalendar** for the current time.

Constructs a **GregorianCalendar** for the specified year, month, and date.

Constructs a **GregorianCalendar** for the specified year, month, date, hour, minute, and second. The month parameter is 0-based, that is, 0 is for January.



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 behavior for objects*. For example, you can specify that the objects are *comparable*, *edible*, *cloneable* using appropriate interfaces.

# Define an Interface

26

To distinguish an *interface* from a class, Java uses the following syntax to define an interface:

```
/*  
public interface InterfaceName {  
    constant declarations;  
    method signatures;  
} */  
public interface C{  
    public static final int NUM = 20;  
    public void m();  
}
```

# Interface is a Special Class

27

An interface is treated like a *special class* in Java. Each interface is compiled into a *separate bytecode file*, just like a regular class. Like an abstract class, you *cannot create* an *instance* from an interface using the new operator, but in most cases you can use an interface more or less the same way you use an abstract class. For example, you can *use an interface as a data type for a variable*, as the result of casting, and so on.

# Case Study: The Edible Interface

28



Chicken



Apple



plum candy





# Define and Implement Ediable

29

```
public interface Ediable{
    public void taste();
}
public class Chicken implements Ediable{
    public void taste(){
        System.out.println("Spicy Chicken!");
    }
}
public class Apple implements Ediable{
    public void taste(){
        System.out.println("Sweet Apple!");
    }
}
```

# Define and Implement Ediable

30

```
public class Person{
    public static void main(String[] args){
        Person p = new Person();
        Ediable[] foodBag = {new Chicken(), new Apple()};
        for(Ediable food:foodBag) {
            p.eat(food);
        }
    }
    public void eat(Ediable food){
        food.taste();
    }
}
```

# Omitting Modifiers in Interfaces

31

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();  
}
```

How to write a sorting algorithm for any object?



# Define an Interface for Comparison

33

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

# General Sorting Method using Comparison Interface

34

```
class MyUtils{
    public static void sort(IComparison[] list) {
        for(int i = 1; i < list.length; i++) {
            IComparison curObj = list[i];
            int k;
            for(k = i - 1; k >= 0 && curObj.compareTo(list[k]) < 0; k--) {
                list[k + 1] = list[k];
            }
            list[k + 1] = curObj;
        }
    }
}
```

# Implements Comparison Interface (MyNumber)

35

```
class MyNumber implements IComparison{
    @Override
    public int compareTo(Object o) {
        MyNumber number = (MyNumber) o;
        if(this.getValue() > number.getValue()) {
            return 1;
        }else if(this.getValue() < number.getValue()) {
            return -1;
        }else {
            return 0;
        }
    }
}
```

# Implements Comparison Interface (Circle)

36

```
class Circle implements IComparison{
    @Override
    public int compareTo(Object o) {
        Circle c = (Circle) o;
        if(this.getRadius() > c.getRadius()) {
            return 1;
        }else if(this.getRadius() < c.getRadius()) {
            return -1;
        }else {
            return 0;
        }
    }
}
```

# Client Program to Test the Sorting Method

37

```
public static void main(String[] args) {  
    // TODO Auto-generated method stub  
    MyNumber[] myList = {new MyNumber(3), new MyNumber(2), new MyNumber(1)};  
    MyUtils.sort(myList);  
    for(MyNumber n:myList) {  
        System.out.printf("%4d", n.getValue());  
    }  
    Circle[] circleList = {new Circle(4.2), new Circle(3.2), new Circle(1.2)};  
    System.out.println();  
    MyUtils.sort(circleList);  
    for(Circle c:circleList) {  
        System.out.printf("%5.1f", c.getRadius());  
    }  
}
```

# Program Updating Using Comparable Defined in JDK

38

```
class MyNumber implements Comparable<MyNumber>{
    @Override
    public int compareTo(MyNumber o) {
        if(this.getValue() > o.getValue()) {
            return 1;
        }else if(this.getValue() < o.getValue()) {
            return -1;
        }else {
            return 0;
        }
    }
}
```

# Program Updating Using Comparable Defined in JDK

39

```
class Circle implements Comparable<Circle>{
    @Override
    public int compareTo(Circle o) {
        Circle c = (Circle) o;
        if(this.getRadius() > c.getRadius()) {
            return 1;
        }else if(this.getRadius() < c.getRadius()) {
            return -1;
        }else {
            return 0;
        }
    }
}
```

# Program Updating Using Comparable Defined in JDK

40

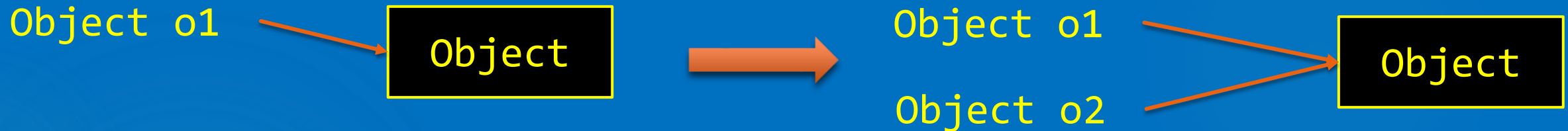
```
public static void main(String[] args) {
    List<MyNumber> myList = Arrays.asList(new MyNumber[] {new MyNumber(
3),new MyNumber(2),new MyNumber(1)});
    Collections.sort(myList);
    for(MyNumber n:myList) {
        System.out.printf("%4d", n.getValue());
    }
    List<Circle> circleList = Arrays.asList(new Circle[] {new Circle(4.
2), new Circle(3.2), new Circle(1.2)});
    Collections.sort(circleList);
    for(Circle c:circleList) {
        System.out.printf("%5.1f", c.getRadius());
    }
}
```



# How to copy an Object?

41

```
Object o1 = new Object();  
Object o2 = o1;
```



The Object class provide a clone() method to copy objects

```
public class Object {  
    protected native Object clone() throws CloneNotSupportedException;  
}
```

# Mark a Cloneable Class using a Interface

42

A *marker interface* does not contain constants or methods. It is used to denote that a class possesses certain desirable properties. A class that implements the Cloneable interface is marked cloneable, and its objects can be cloned using the clone() method defined in the Object class.

A null interface used for cloneable marking

```
public interface Cloneable { }
```

```
java.lang.CloneNotSupportedException: test.CloneableObject  
    at java.lang.Object.clone(Native Method)  
    at test.CloneableObject.clone(Test.java:21)  
    at test.Test.main(Test.java:12)
```

# Overriding The clone() method

43

```
public class CloneableObject implements Cloneable{
    public int num = 1;
    public Object clone(){
        CloneableObject co = null;
        try{
            co = (CloneableObject)super.clone();
        }catch(CloneNotSupportedException e){
            System.out.println(e.toString());
        }
        return co;
    }
}
```

# Object copying by the clone() method

44

```
public class CloneableTest{
    public static void main(String[] args){
        CloneableObject co1 = new CloneableObject();
        CloneableObject co2 = (CloneableObject)co1.clone();
        co2.num = 888;
        System.out.println(co1.num);
        System.out.println(co2.num);
    }
}
```

# Shallow Copy

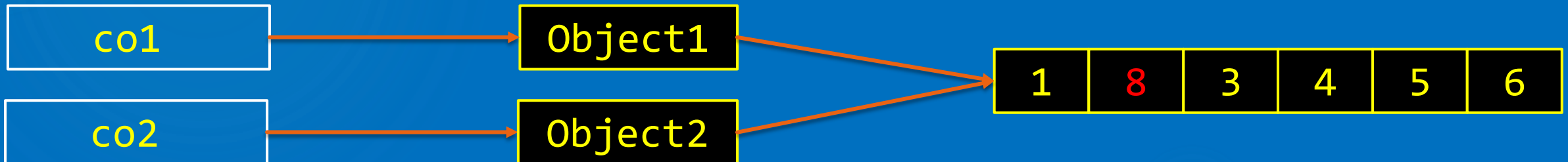
45

```
public class CloneableObject implements Cloneable{
    public int[] array = {1,2,3,4,5,6};
    public Object clone(){
        CloneableObject co = null;
        try{
            co = (CloneableObject)super.clone();
        }catch(CloneNotSupportedException e){
            System.out.println(e.toString());
        }
        return co;
    }
}
```

# Shallow Copy

46

```
public class CloneableTest{  
    public static void main(String[] args){  
        CloneableObject co1 = new CloneableObject();  
        CloneableObject co2 = (CloneableObject)co1.clone();  
        co2.array[1] = 8;  
        System.out.println(co1.array[1]); //8  
        System.out.println(co2.array[1]); //8  
    }  
}
```



# Deep Copy

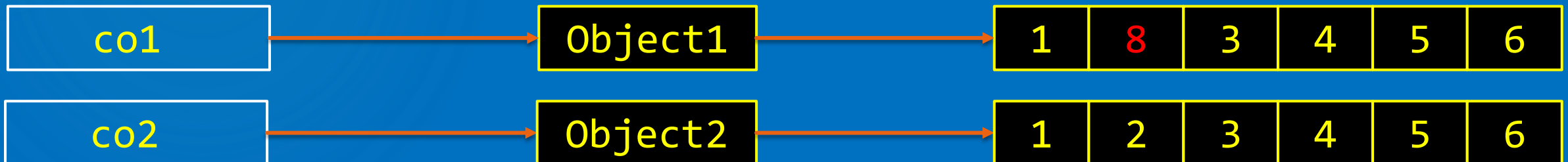
47

```
public class CloneableObject implements Cloneable{
    public int[] array = {1,2,3,4,5,6};
    public Object clone(){
        CloneableObject co = null;
        try{
            co = (CloneableObject)super.clone();
            co.array = array.clone();
        }catch(CloneNotSupportedException e){
            System.out.println(e.toString());
        }
        return co;
    }
}
```

# Deep Copy

48

```
public class CloneableTest{  
    public static void main(String[] args){  
        CloneableObject co1 = new CloneableObject();  
        CloneableObject co2 = (CloneableObject)co1.clone();  
        co1.array[1] = 8;  
        System.out.println(co1.array[1]); //8  
        System.out.println(co2.array[1]); //2  
    }  
}
```





# Interfaces vs. Abstract Classes

49

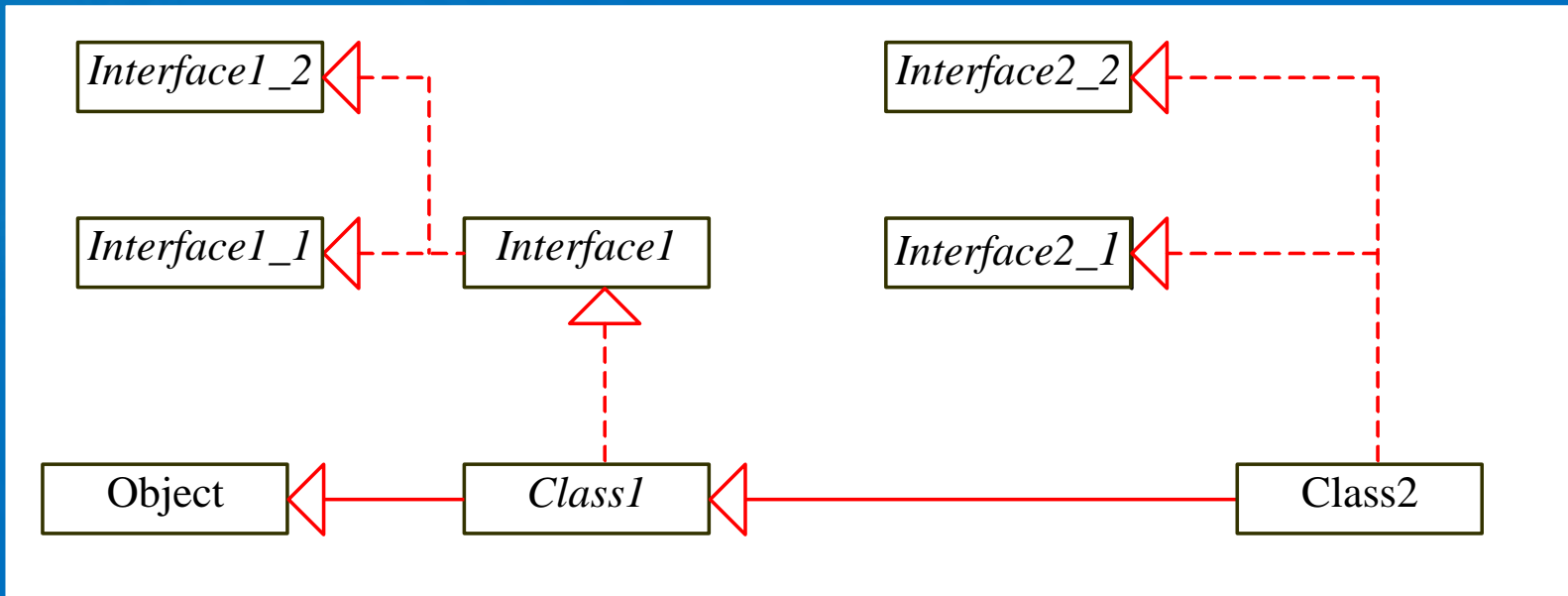
In an *interface*, the *data* must be *constants*; an *abstract class* can have *all types of data*. Each method in an interface has only a signature without implementation; an *abstract* class can have *concrete methods*.

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

# Interfaces vs. Abstract Classes

50

All classes share a single root, the Object class, **but there is no single root for interfaces**. Like a class, an interface also defines a type. **A variable of an interface type can reference any instance of the class that implements the interface.** If a class extends an interface, this interface plays the same role as a superclass. You can use an interface as a data type and cast a variable of an interface type to its subclass, and vice versa.



## Caution: conflict interfaces

51

In rare occasions, a class may implement two interfaces with conflict information (e.g., *two same constants with different values* or *two methods with same signature but different return type*). This type of errors will be detected by the compiler.

```
public interface InterfaceA{
    public static final int A = 20;
}
public interface InterfaceB{
    public static final int A = 60;
    public int method();
}
public class ClassA implements InterfaceA, InterfaceB{}
```

# Whether to use an interface or a class?

52

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 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.*

Can the following code be compiled? Why?

```
Integer n1 = new Integer(3);  
Object n2 = new Integer(4);  
System.out.println(n1.compareTo(n2));
```

You can define the *compareTo* method in a class without implementing the *Comparable* interface. What are the benefits of implementing the *Comparable* interface?

What is wrong in the following code?

```
public class Test {  
    public static void main(String[] args) {  
        Person[] persons = {new Person(3), new Person(4), new Person(1)};  
        java.util.Arrays.sort(persons);  
    }  
}  
  
class Person {  
    private int id;  
    Person(int id) {  
        this.id = id;  
    }  
}
```

Which of the following is a correct interface?

```
interface A {  
    void print() { };  
}
```

(a)

```
abstract interface A extends I1, I2 {  
    abstract void print() { };  
}
```

(b)

```
abstract interface A {  
    print();  
}
```

(c)

```
interface A {  
    void print();  
}
```

(d)