Chapter 13 Abstract Classes and Interfaces

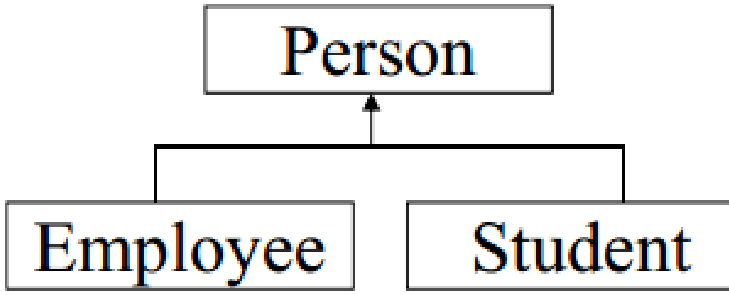
Abstract classes: Defining templates for subclasses

Interfaces: Defining common behavior for unrelated classes

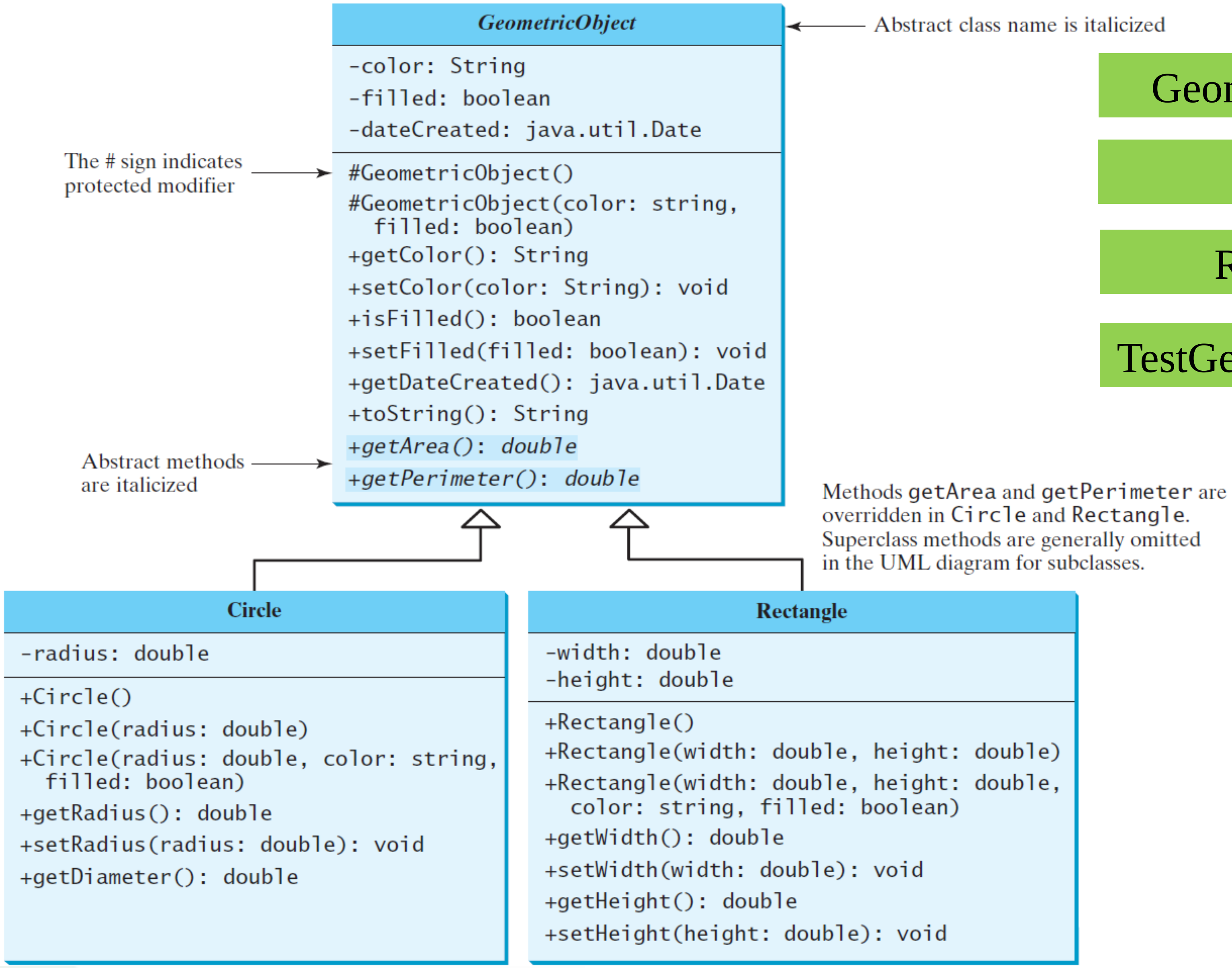
abstract Classes and Methods

- ☞ **Abstract classes**: some methods are **only** declared, but no **concrete** implementations are provided.
- ☞ Those methods called **abstract methods** and they need to be implemented by the extending classes.

```
abstract  class  Person {
   protected  String  name;
   . . .

   public  abstract  String  getDescription() ;
   . . .
}
Class  Student  extends  Person {
   private  String  major;
   . . .
   public  String  getDescription() {
           return  name + " a student major in " + major;
   }
}
Class  Employee  extends  Person {
   private  float  salary;
   . . .
   public  String  getDescription() {
           return  name + " an employee with a salary of $ " + salary;
   }
}
```



Abstract Classes and Abstract 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
+getArea(): double
+getPerimeter(): double
Abstract class name is italicized
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

GeometricObject

Circle

Rectangle

TestGeometricObject

Run

abstract method in abstract class

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 nonabstract subclass extended from an abstract class, all the abstract methods must be implemented, even if they are not used in the subclass.

☞ In UML graphic notation, the names of abstract classes and their abstract methods

☞are italicized

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.
  - ➢ In this case, you **cannot** create instances of the class using the **new** operator.
  - ➢ The constructor in the abstract class is defined as protected because it is used only by subclasses
  - ➢ 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];
```

Which is correct?

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

(a)

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

(b)

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

(c)

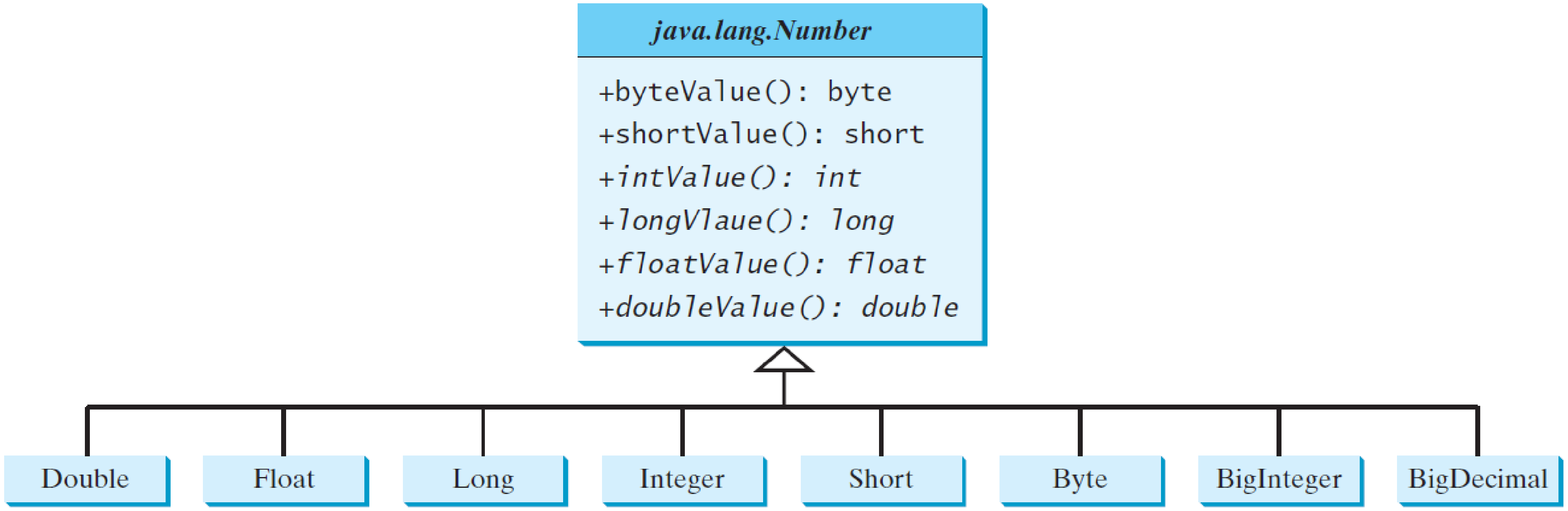
```java
abstract class A {
  protected void unfinished();
}
```

(d)

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

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

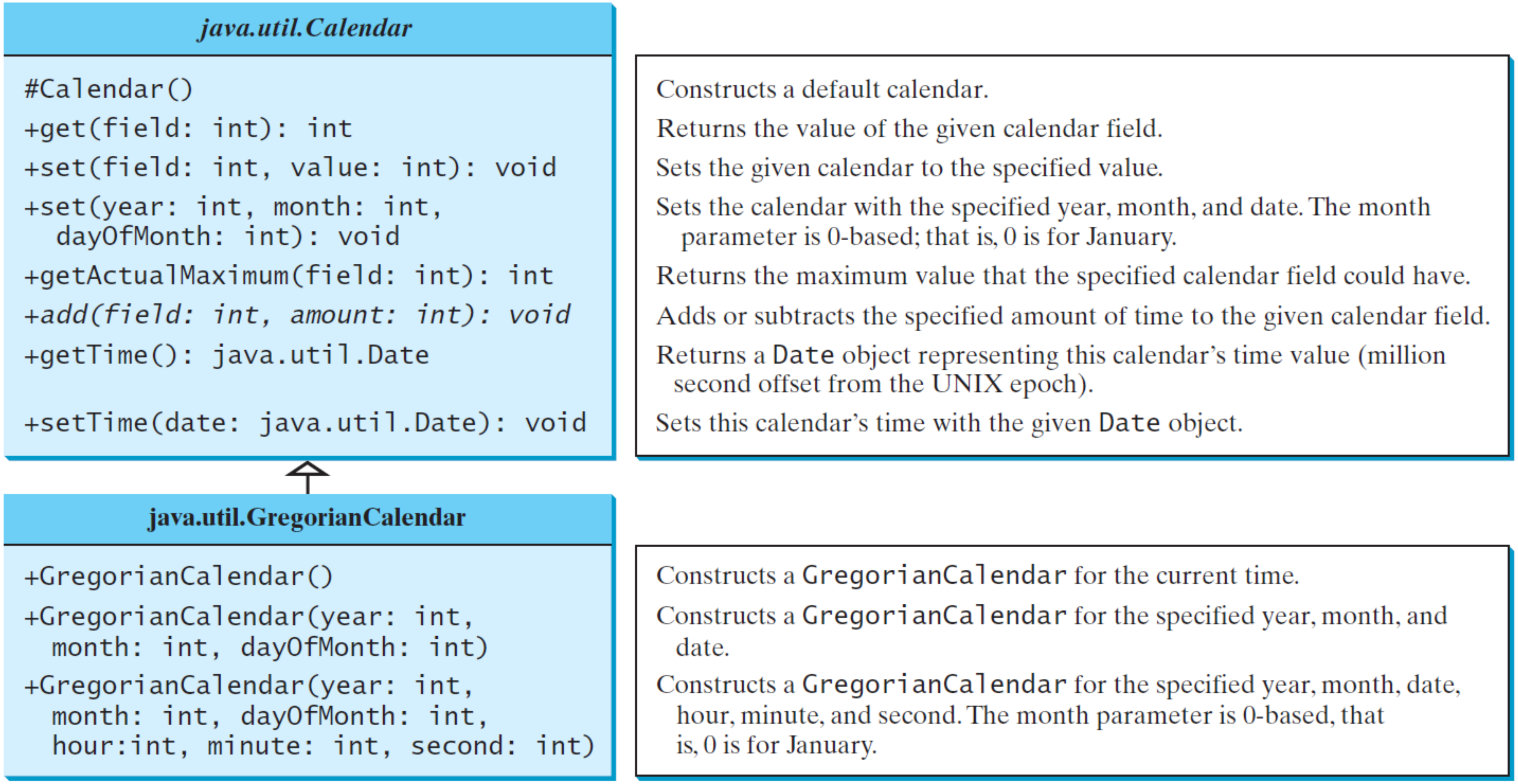
Case Study: the Abstract Number Class



LargestNumbers

Run

The Abstract Calendar Class and Its GregorianCalendar subclass



java.util.Calendar

`#Calendar()`	Constructs a default calendar.
`+get(field: int): int`	Returns the value of the given calendar field.
`+set(field: int, value: int): void`	Sets the given calendar to the specified value.
`+set(year: int, month: int, dayOfMonth: int): void`	Sets the calendar with the specified year, month, and date. The month parameter is 0-based; that is, 0 is for January.
`+getActualMaximum(field: int): int`	Returns the maximum value that the specified calendar field could have.
`+add(field: int, amount: int): void`	Adds or subtracts the specified amount of time to the given calendar field.
`+getTime(): java.util.Date`	Returns a `Date` object representing this calendar's time value (million second offset from the UNIX epoch).
`+setTime(date: java.util.Date): void`	Sets this calendar's time with the given `Date` object.

java.util.GregorianCalendar

`+GregorianCalendar()`	Constructs a `GregorianCalendar` for the current time.
`+GregorianCalendar(year: int, month: int, dayOfMonth: int)`	Constructs a `GregorianCalendar` for the specified year, month, and date.
`+GregorianCalendar(year: int, month: int, dayOfMonth: int, hour:int, minute: int, second: int)`	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.

GregorianCalendar subclass

- An instance of **java.util.Date** represents a specific instant in time with millisecond precision.
- **java.util.Calendar** is an abstract base class for extracting detailed information such as year, month, date, hour, minute and second from a Date object.
- Subclasses of Calendar can implement specific calendar systems such as **Gregorian calendar**, **Lunar Calendar** and **Jewish calendar**.
- Currently, **java.util.GregorianCalendar** for the Gregorian calendar is supported in the Java API.

The GregorianCalendar Class

You can use new GregorianCalendar() to construct a default GregorianCalendar with the current time.

Use new GregorianCalendar(year, month, date) to construct a GregorianCalendar with the specified year, month, and date.

The month parameter is 0-based, i.e., 0 is for January.

The get Method in Calendar Class

The get(int field) method defined in the Calendar class is useful to extract the date and time information from a Calendar object. The fields are defined as constants, as shown in the following.

Constant	*Description*
YEAR	The year of the calendar.
MONTH	The month of the calendar, with 0 for January.
DATE	The day of the calendar.
HOUR	The hour of the calendar (12-hour notation).
HOUR_OF_DAY	The hour of the calendar (24-hour notation).
MINUTE	The minute of the calendar.
SECOND	The second of the calendar.
DAY_OF_WEEK	The day number within the week, with 1 for Sunday.
DAY_OF_MONTH	Same as DATE.
DAY_OF_YEAR	The day number in the year, with 1 for the first day of the year.
WEEK_OF_MONTH	The week number within the month, with 1 for the first week.
WEEK_OF_YEAR	The week number within the year, with 1 for the first week.
AM_PM	Indicator for AM or PM (0 for AM and 1 for PM).

Getting Date/Time Information from Calendar

TestCalendar

Run

Interfaces

What is an interface?

Why is an interface useful?

How do you define an interface?

How do you use an interface?

What is an interface?
Why is an interface useful?

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 the intent of an interface is to specify common behavior for unrelated objects.

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

What is 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.

Define an Interface

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

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

Example:

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

Interface is a Special Class

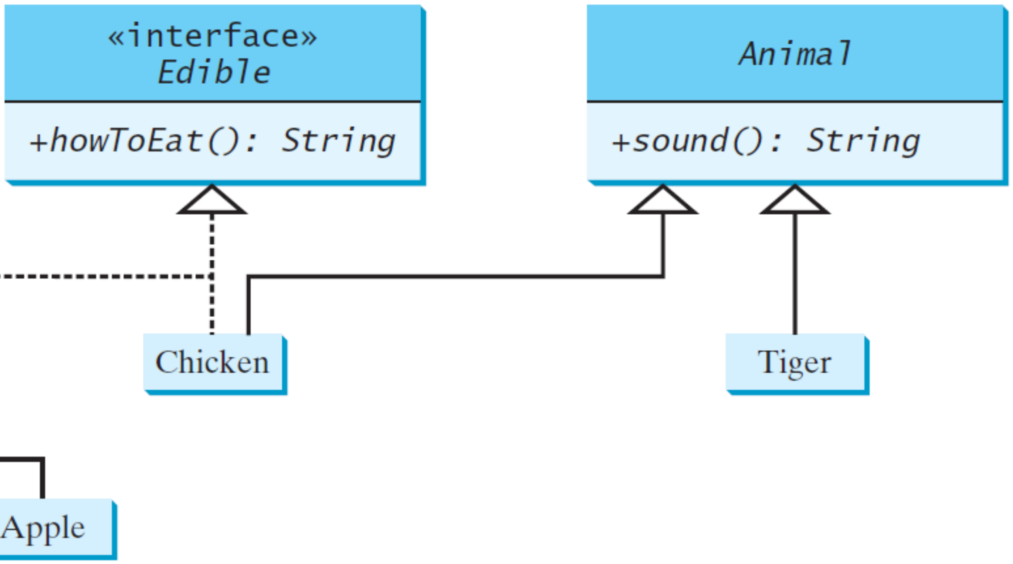
- An **interface** is treated like a special class in **Java**.
- Each **interface** is compiled into a separate **bytecode** file, just like a regular class.
- Like an **abstract** class, you **cannot** create an instance from an **interface** using the **new** operator, but in most cases you can use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

Example

You can now use the Edible interface to specify whether an object is edible. This is accomplished by letting the class for the object implement this interface using the implements keyword. For example, the classes Chicken and Fruit implement the Edible interface (See TestEdible).

Edible TestEdible Run

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



Omitting Modifiers in 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

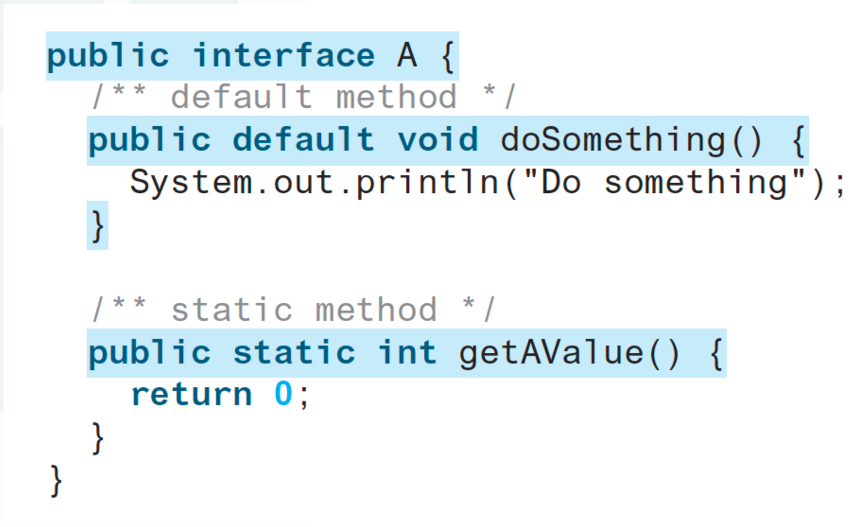
```
public interface T1 {
  int K = 1;

  void p();
}
```

A constant defined in an interface can be accessed using syntax InterfaceName.CONSTANT_NAME (e.g., T1.K).

YOU CAN also have public static methods!

Default keyword (J8)

- A default method provides a default implementation for the method in the interface. A class that implements the interface may simply use the default implementation for the method or override the method with a new implementation.
- Java 8 also permits public static methods in an interface. A public static method in an interface can be used just like a public static method in a class.



```
public interface A {
  /** default method */
  public default void doSomething() {
    System.out.println("Do something");
  }

  /** static method */
  public static int getAValue() {
    return 0;
  }
}
```

Which is correct?

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

(a)

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

(b)

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

(c)

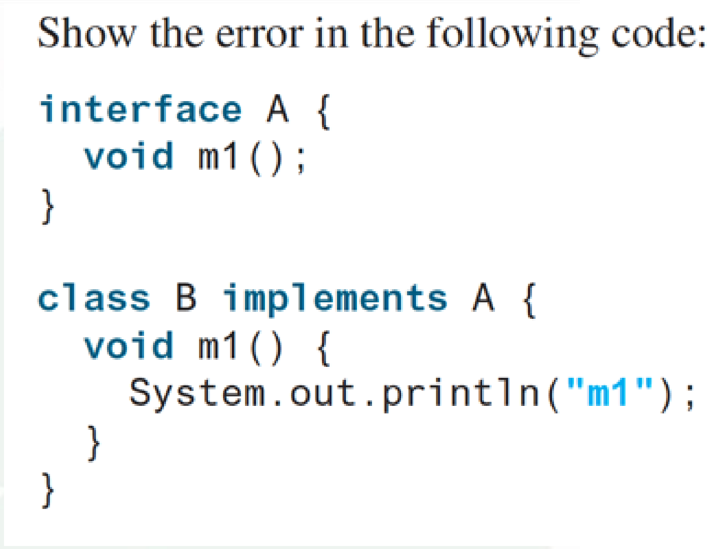
```
interface A {
  void print();
}
```

(d)

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

```
interface A {
  static int get() {
    return 0;
  }
}
```

Find and Explain the Error!



Show the error in the following code:

```
interface A {
  void m1();
}

class B implements A {
  void m1() {
    System.out.println("m1");
  }
}
```

Example: The Comparable Interface

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

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

Integer and BigInteger Classes

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

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

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

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

String and Date Classes

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

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

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

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

Example

```
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));
```

The toString, equals, and hashCode Methods

Each wrapper class overrides the toString, equals, and hashCode methods defined in the Object class. Since all the numeric wrapper classes and the Character class implement the Comparable interface, the compareTo method is implemented in these classes.

Generic `sort` Method

Let **n** be an **Integer** object, **s** be a **String** object, and **d** be a **Date** object. All the following expressions are **true**.

```
n instanceof Integer
n instanceof Object
n instanceof Comparable
```

```
s instanceof String
s instanceof Object
s instanceof Comparable
```

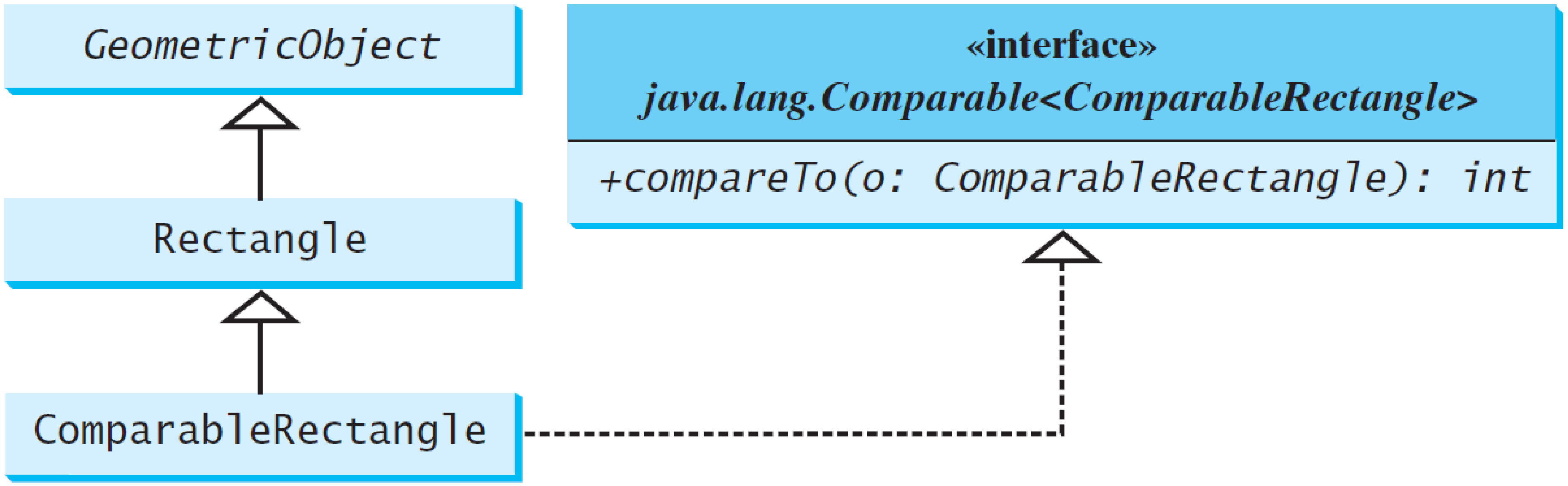
```
d instanceof java.util.Date
d instanceof Object
d instanceof Comparable
```

The java.util.Arrays.sort(array) method requires that the elements in an array are instances of Comparable<E>.

SortComparableObjects

Run

Defining Classes to Implement Comparable


GeometricObject
Rectangle
ComparableRectangle
«interface»
java.lang.Comparable<ComparableRectangle>
+compareTo(o: ComparableRectangle): int

ComparableRectangle SortRectangles Run

Extending Interfaces

- Interfaces support **multiple** inheritance: an **interface** can extend **more** than one **interface**.
- **Superinterfaces** and **subinterfaces**.
- Example:

```
public interface SerializableRunnable
extends
    java.io.Serializable , Runnable {
            …
}
```

Extending Interfaces – Constants

☞ If a **superinterface** and a **subinterface** contain two constants with the same name, then the one belonging to the superinterface is **hidden**:

```
interface X {
            int  val  = 1;
}
interface Y extends X {
            int  val  = 2;
            int sum = val  +  X.val;
}
```

Extending Interfaces – Methods

- If a declared method in a subinterface has the same signature as an inherited method **and** the same return type, then the new declaration **overrides** the inherited method in its superinterface.

- If the **only** difference is in the return type, then there will be a **compile-time error**.

The `Cloneable` Interfaces

Marker Interface: An empty interface.

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.

```
package java.lang;
public interface Cloneable {
}
```

Examples

Many classes (e.g., Date and Calendar) in the Java library implement Cloneable. Thus, the instances of these classes can be cloned. For example, the following code

```
Calendar calendar = new GregorianCalendar(2003, 2, 1);
Calendar calendarCopy = (Calendar)calendar.clone();
System.out.println("calendar == calendarCopy is " +
  (calendar == calendarCopy));
System.out.println("calendar.equals(calendarCopy) is " +
  calendar.equals(calendarCopy));
```

displays

```
calendar == calendarCopy is false
calendar.equals(calendarCopy) is true
```

Implementing Cloneable Interface

To define a custom class that implements the Cloneable interface, the class must override the clone() method in the Object class. The following code defines a class named House that implements Cloneable and Comparable.

House

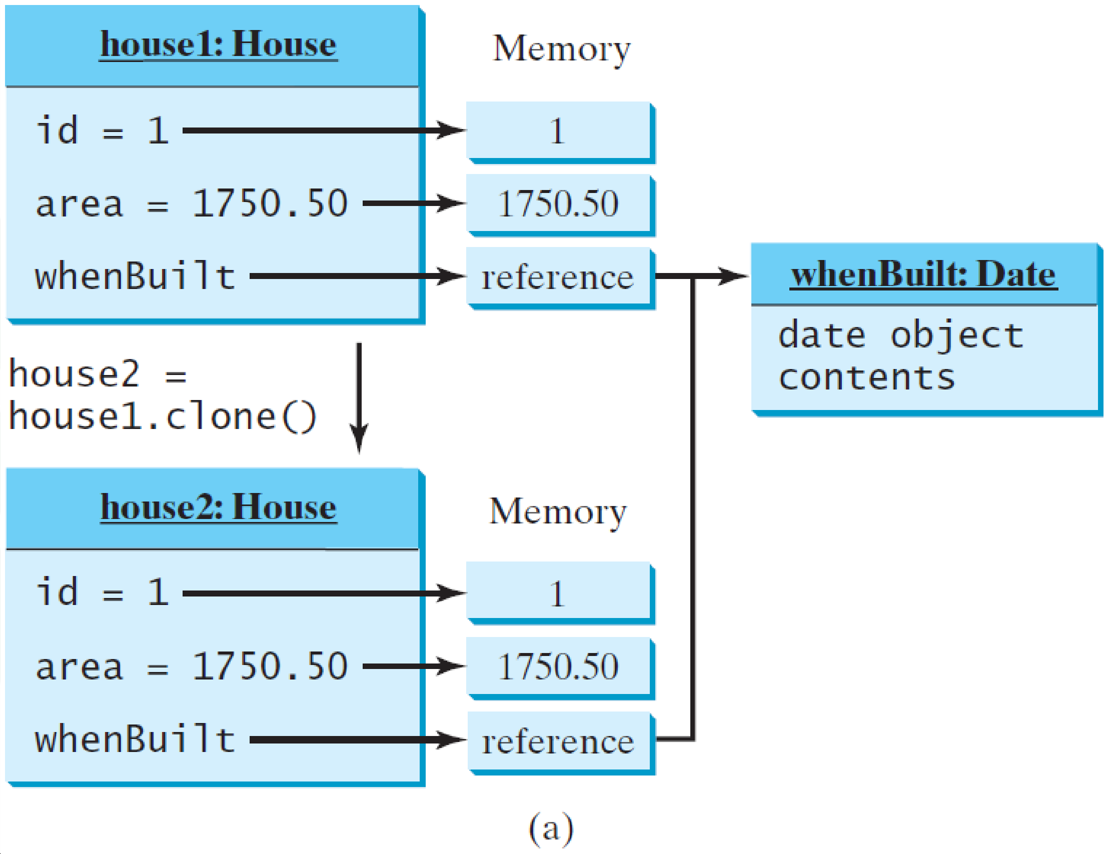
```
public  class House implements Cloneable, Comparable<House>
{
  private  int  id;
  private  double  area;
  private  java.util.Date   whenBuilt;

  public House(int id, double  area) {
    this.id = id;
    this.area = area;
    whenBuilt = new java.util.Date();
  }

  public int getId()      {        return id;      }

  public double getArea()       {     return area;  }

  public java.util.Date   getWhenBuilt()   {     return whenBuilt;     }
@Override // Override the clone method defined in the Object class
 public Object clone() {
         return  super.clone();
  }

 @Override // Implement the compareTo method defined in Comparable
 public int  compareTo(House  o) {
       if (area  >  o.area)
         return 1;
       else if (area < o.area)
         return -1;
       ~~else~~
         return 0;
 }
}
```

Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);

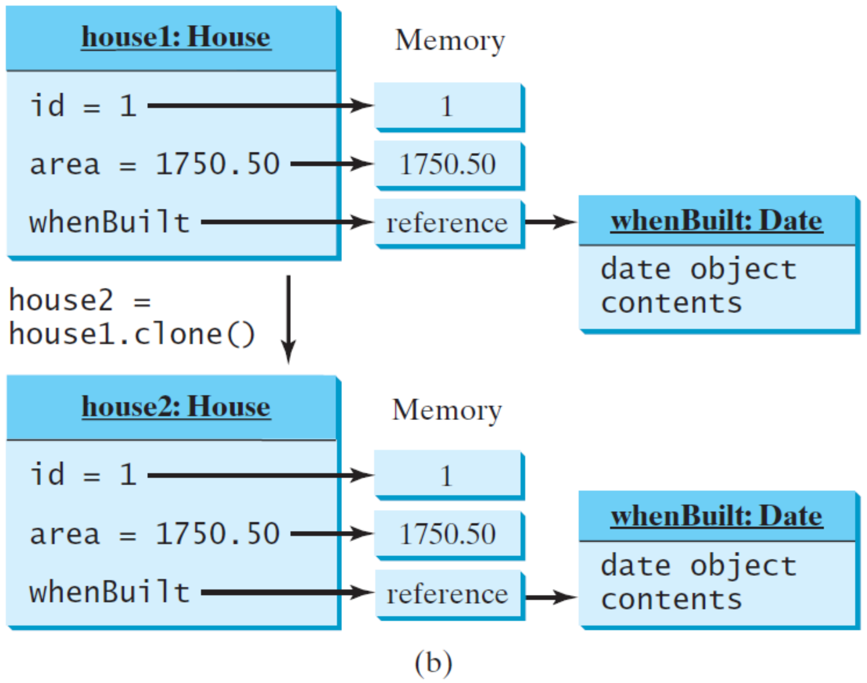
House house2 = (House)house1.clone();

Shallow Copy



Shallow vs. Deep Copy

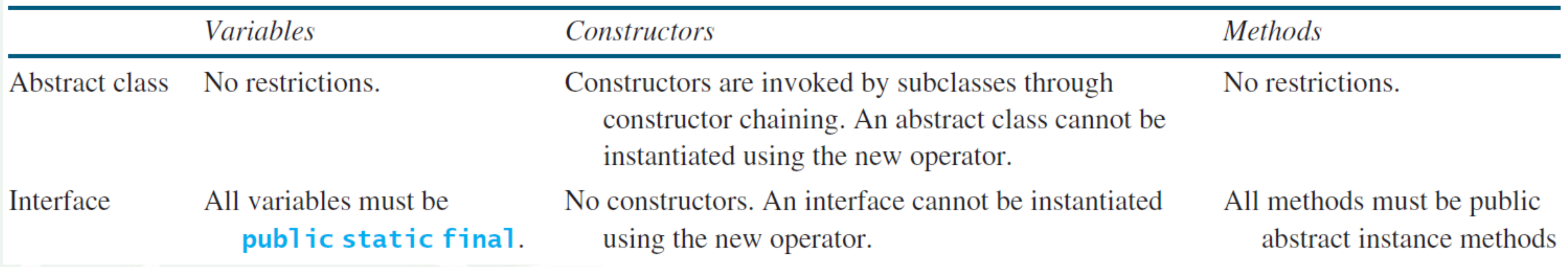
```
public Object clone() {
   try {
         // Perform a shallow copy
         House houseClone = (House)super.clone();
         // Deep copy on whenBuilt
         houseClone.whenBuilt = (java.util.Date)(whenBuilt.clone());
         return houseClone;
         }
   catch (CloneNotSupportedException ex) {
         return null;
   }
}
```



Interfaces vs. Abstract Classes

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

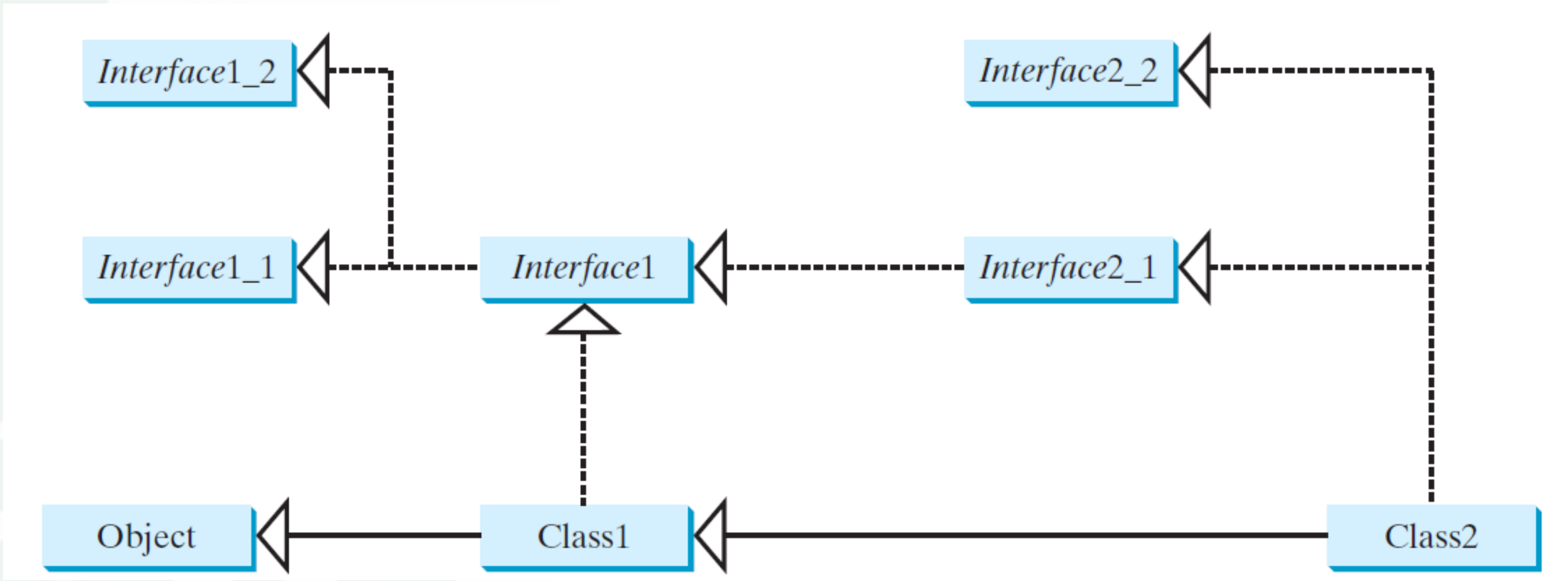
Each method in an interface has only a signature without implementation; an abstract class can have concrete methods.



	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

Interfaces vs. Abstract Classes, cont.

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.



Suppose that c is an instance of Class2. c is also an instance of Object, Class1, Interface1, Interface1_1, Interface1_2, Interface2_1, and Interface2_2.

Caution: conflict interfaces

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.

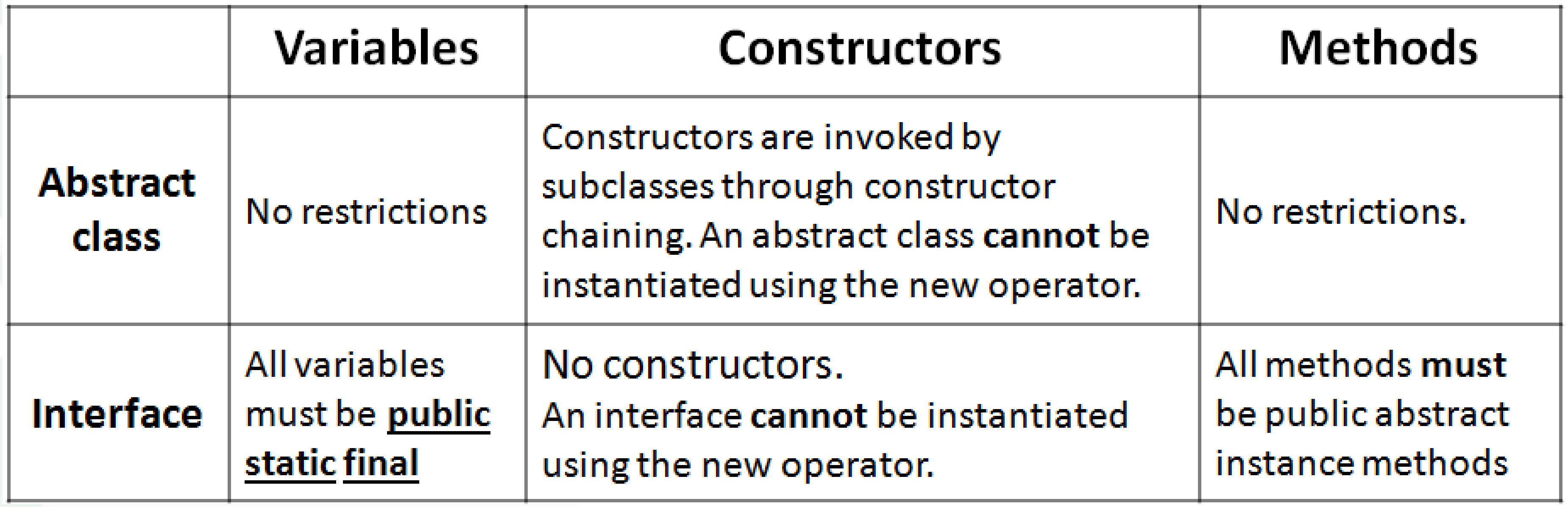
Whether to use an interface or a class?

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. 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.

Interfaces vs. Abstract Classes

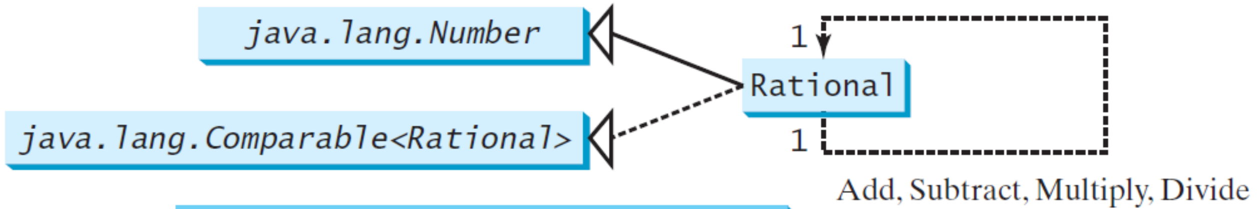
❖ In an **interface**, the data must be **constants**; an **abstract** class can have all types of data.

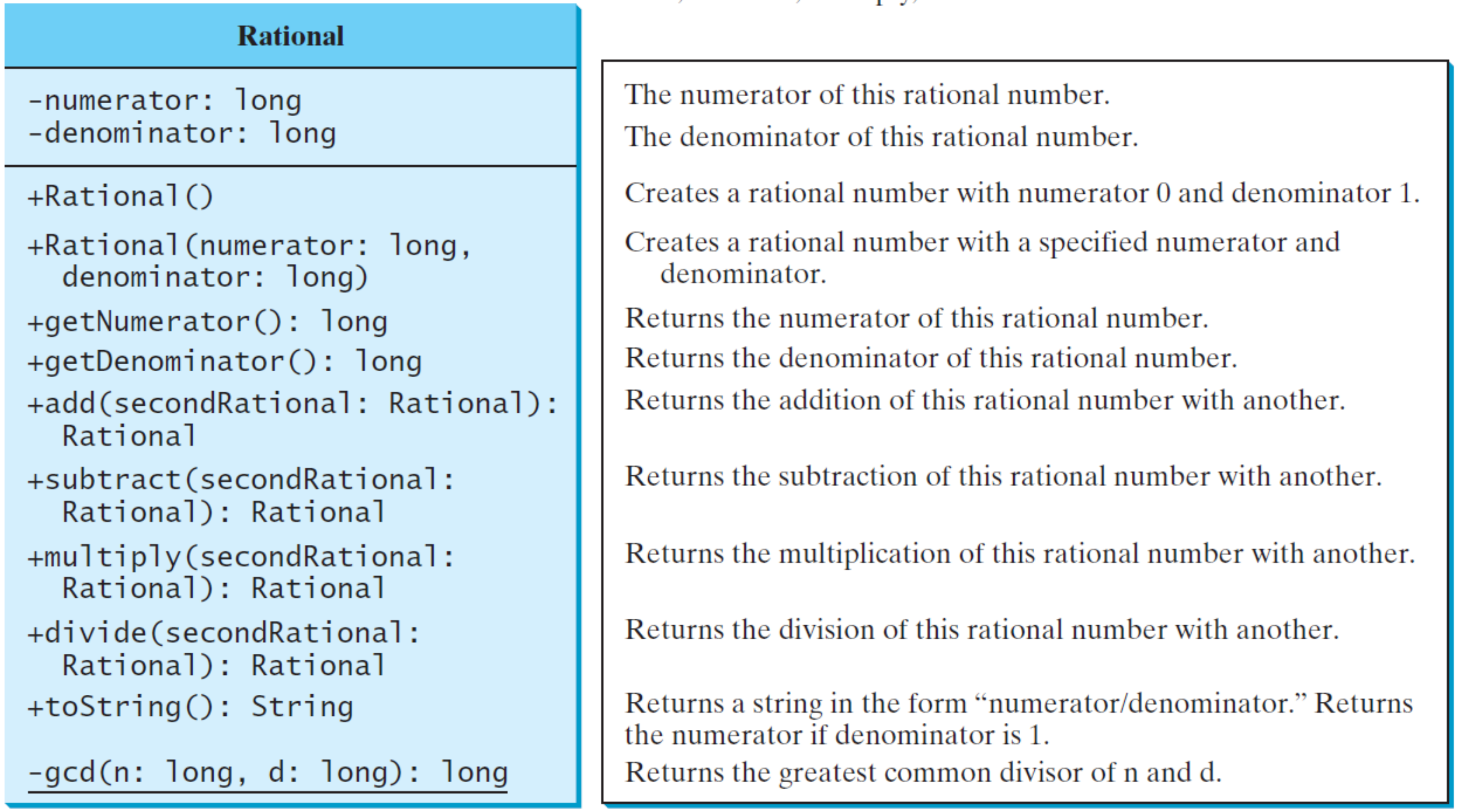
❖ Each method in an **interface** has only a signature without implementation; an **abstract** class can have concrete methods.



	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

The `Rational` Class


java.lang.Number
java.lang.Comparable<Rational>
Rational
1
1
Add, Subtract, Multiply, Divide



Rational	
-numerator: long	The numerator of this rational number.
-denominator: long	The denominator of this rational number.
+Rational()	Creates a rational number with numerator 0 and denominator 1.
+Rational(numerator: long, denominator: long)	Creates a rational number with a specified numerator and denominator.
+getNumerator(): long	Returns the numerator of this rational number.
+getDenominator(): long	Returns the denominator of this rational number.
+add(secondRational: Rational): Rational	Returns the addition of this rational number with another.
+subtract(secondRational: Rational): Rational	Returns the subtraction of this rational number with another.
+multiply(secondRational: Rational): Rational	Returns the multiplication of this rational number with another.
+divide(secondRational: Rational): Rational	Returns the division of this rational number with another.
+toString(): String	Returns a string in the form "numerator/denominator." Returns the numerator if denominator is 1.
-gcd(n: long, d: long): long	Returns the greatest common divisor of n and d.

Rational | TestRationalClass | Run