Abstract Classes and Interfaces Generics

Reading: Liang, Chapter 13 and Chapter 17

Horstmann, Chapter 6.1, Chapter 8

Bonus: Number objects, Integer pool, and the String pool

Midterm

- Don't worry, it's not hard

- 10 questions about Java
  - Explain your reasoning. This is takehome, so there aren't any "show what this method outputs" questions, because you could just run that code yourself.

- Two programming questions

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.

The `Integer` and `Double` Classes

java.lang.Integer
-value: int
+MAX_VALUE: int
+MIN_VALUE: int
+Integer(value: int)
+Integer(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longVlaue(): long
+floatValue(): float
+doubleValue():double
+compareTo(o: Integer): int
+toString(): String
+valueOf(s: String): Integer
+valueOf(s: String, radix: int): Integer
+parseInt(s: String): int
+parseInt(s: String, radix: int): int

java.lang.Double
-value: double
+MAX_VALUE: double
+MIN_VALUE: double
+Double(value: double)
+Double(s: String)
+byteValue(): byte
+shortValue(): short
+intValue(): int
+longVlaue(): long
+floatValue(): float
+doubleValue():double
+compareTo(o: Double): int
+toString(): String
+valueOf(s: String): Double
+valueOf(s: String, radix: int): Double
+parseDouble(s: String): double
+parseDouble(s: String, radix: int): double

The `Integer` Class and the `Double` Class

❑ Constructors

❑ Class Constants `MAX_VALUE, MIN_VALUE`

❑ Conversion Methods

Numeric Wrapper Class Constructors

You can construct a wrapper object either from a primitive data type value or from a string representing the numeric value. The constructors for Integer and Double are:

public Integer(int value)

public Integer(String s)

public Double(double value)

public Double(String s)

Numeric Wrapper Class Constants

Each numerical wrapper class has the constants MAX_VALUE and MIN_VALUE. MAX_VALUE represents the maximum value of the corresponding primitive data type. For Byte, Short, Integer, and Long, MIN_VALUE represents the minimum byte, short, int, and long values. For Float and Double, MIN_VALUE represents the minimum *positive* float and double values. The following statements display the maximum integer (2,147,483,647), the minimum positive float (1.4E-45), and the maximum double floating-point number (1.79769313486231570e+308d).

Conversion Methods

Each numeric wrapper class implements the abstract methods <u>doubleValue</u>, <u>floatValue</u>, <u>intValue</u>, <u>longValue</u>, and <u>shortValue</u>, which are defined in the <u>Number</u> class. These methods "convert" objects into primitive type values.

The Static <u>valueOf</u> Methods

The numeric wrapper classes have a useful class method, valueOf(String s). This method creates a new object initialized to the value represented by the specified string. For example:
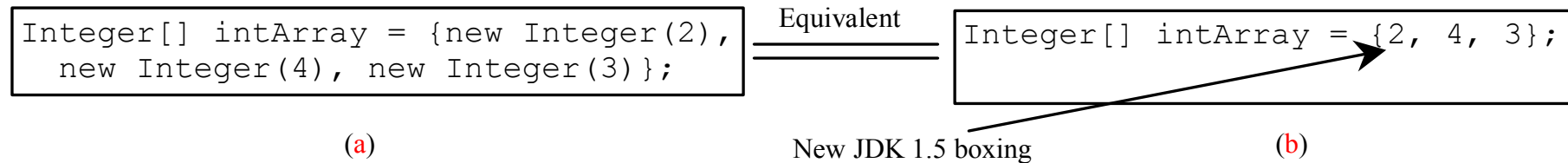
Double doubleObject = Double.valueOf("12.4");

Integer integerObject = Integer.valueOf("12");

The Methods for Parsing Strings into Numbers

You have used the parseInt method in the Integer class to parse a numeric string into an int value and the parseDouble method in the Double class to parse a numeric string into a double value. Each numeric wrapper class has two overloaded parsing methods to parse a numeric string into an appropriate numeric value.

Automatic Conversion Between Primitive Types and Wrapper Class Types

JDK 1.5 allows primitive type and wrapper classes to be converted automatically. For example, the following statement in (a) can be simplified as in (b):

```
Integer[] intArray = {new Integer(2),
    new Integer(4), new Integer(3)};
```

Equivalent

```
Integer[] intArray = {2, 4, 3};
```

New JDK 1.5 boxing

(a)

(b)

Integer[] intArray = {1, 2, 3};
System.out.println(intArray[0] + intArray[1] + intArray[2]);

Unboxing

BigInteger and BigDecimal

If you need to compute with very large integers or high precision floating-point values, you can use the BigInteger and BigDecimal classes in the java.math package. Both are *immutable*. Both extend the Number class and implement the Comparable interface.

BigInteger and BigDecimal

BigInteger a = **new** BigInteger("9223372036854775807");

BigInteger b = **new** BigInteger("2");

BigInteger c = a.multiply(b); // 9223372036854775807 * 2

System.out.println(c);

LargeFactorial

BigDecimal a = **new** BigDecimal(1.0);

BigDecimal b = **new** BigDecimal(3);

BigDecimal c = a.divide(b, 20, BigDecimal.ROUND_UP);

System.out.println(c);

The String Pool and the Integer Pool

- Remember if you create two Strings with the name value, they won't be the same string:

```
String a = new String("Hello!");
String b = new String("Hello!");
System.out.println(a == b); // prints "false"
```

- But if you declare strings as literals, the compiler implicitly instantiates that String, and every *other* time it encounters that String literal, it retrieves it from the same place:

```
String c = Hello!";
String d = Hello!";
System.out.println(c == d); // prints "true" (!)
```

The String Pool

- What Java does is store all these String literals in a place called the "String pool", so it does not duplicate and recreate String literals when they're encountered. `c` and `d` here will point to the same String literal

```
String c = "Hello!";
String d = "Hello!";
```

```
String e = "Goodbye!";
String f = new String("Goodbye!");
String g = new String("Goodbye!");
```

c   d   e   f

"Hello!"   "Goodbye!"

Goodbye!

g

Goodbye!
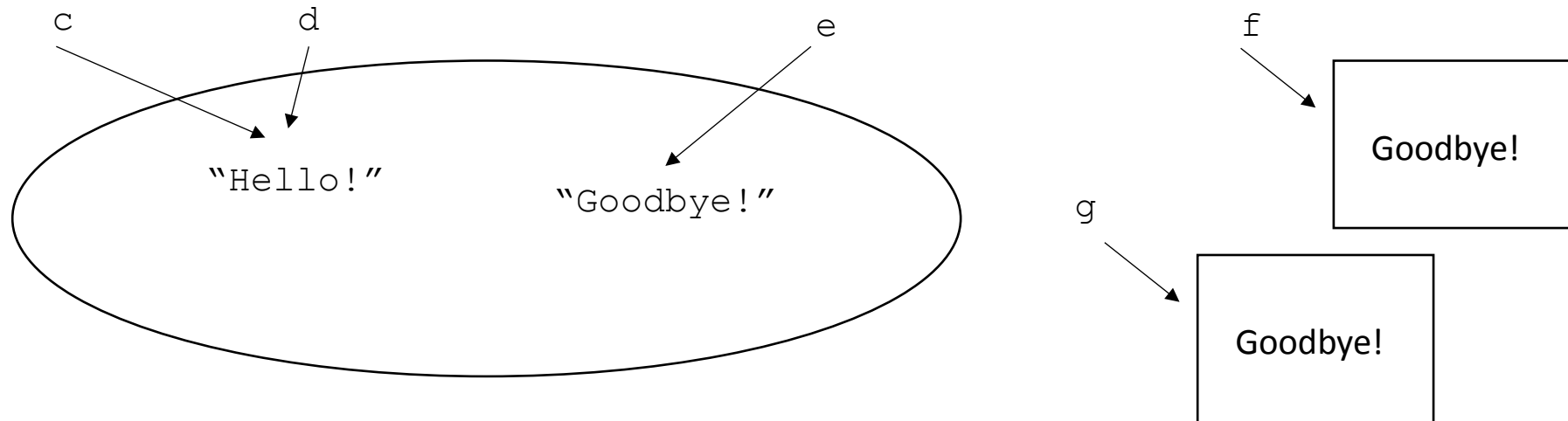
The Integer Pool

- Works like the String Pool, except that explicitly creating new Integer objects are all drawn from the Integer pool for all Integers from -128 to 127

```
Integer i = new Integer(12);
Integer j = new Integer(12);
System.out.println(i == j); // prints "true"

Integer m = new Integer(999);
Integer n = new Integer(999);
System.out.println(m == n); // prints "false"
```

Motivations

❑An *interface* is for defining common behavior for classes (including unrelated classes). Before discussing interfaces, we introduce a closely related subject: abstract classes.

Objectives

❑ To design and use abstract classes (§13.2).

❑ To generalize numeric wrapper classes, **BigInteger**, and **BigDecimal** using the abstract **Number** class (§13.3).

❑ To process a calendar using the **Calendar** and **GregorianCalendar** classes (§13.4).

❑ To specify common behavior for objects using interfaces (§13.5).

❑ To define interfaces and define classes that implement interfaces (§13.5).

❑ To define a natural order using the **Comparable** interface (§13.6).

❑ To make objects cloneable using the **Cloneable** interface (§13.7).

❑ To explore the similarities and differences among concrete classes, abstract classes, and interfaces (§13.8).

abstract method in abstract class

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.

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. 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.
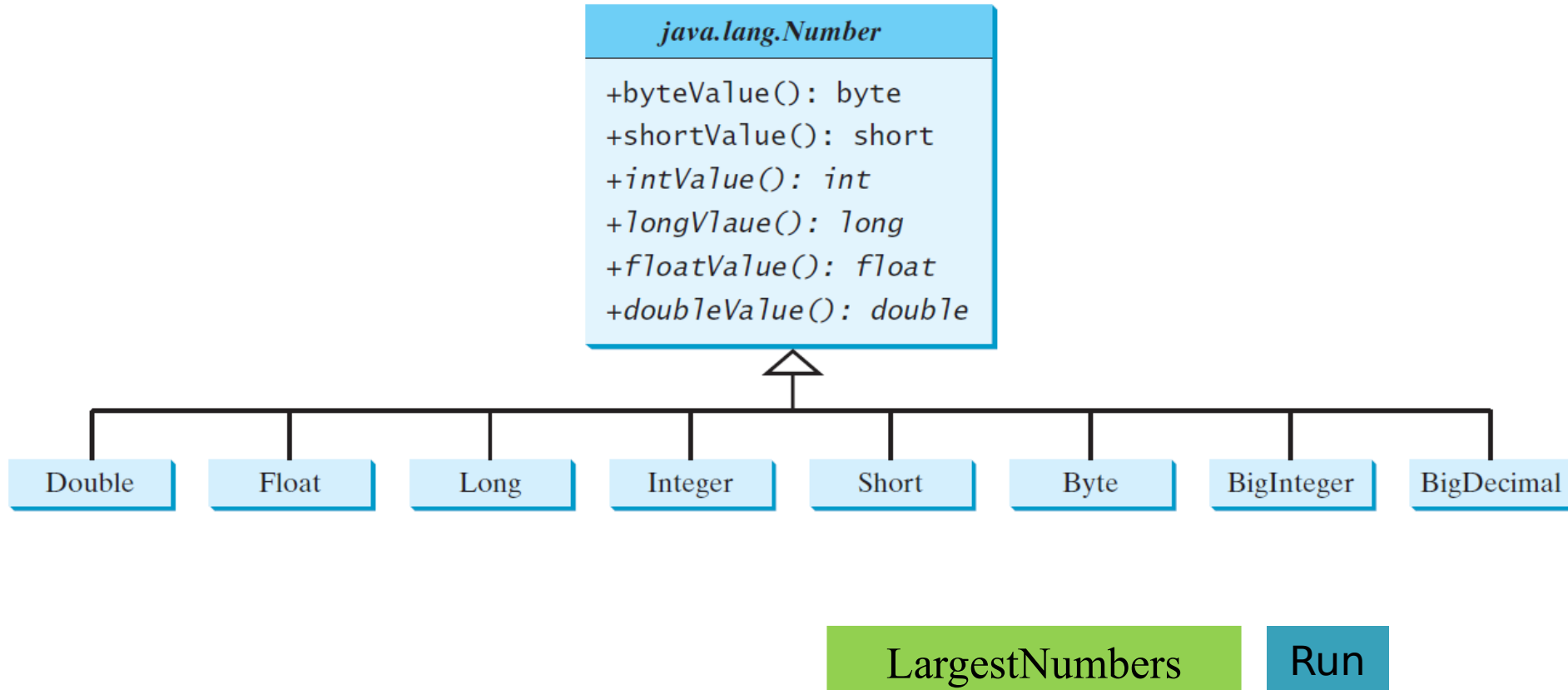
abstract class as type

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];

Case Study: the Abstract Number Class

```
         java.lang.Number

+byteValue(): byte
+shortValue(): short
+intValue(): int
+longVlaue(): long
+floatValue(): float
+doubleValue(): double
```

| Double | Float | Long | Integer | Short | Byte | BigInteger | BigDecimal |

LargestNumbers   Run

25

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.

The Abstract Calendar Class and Its 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 and 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).

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 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();
}
```
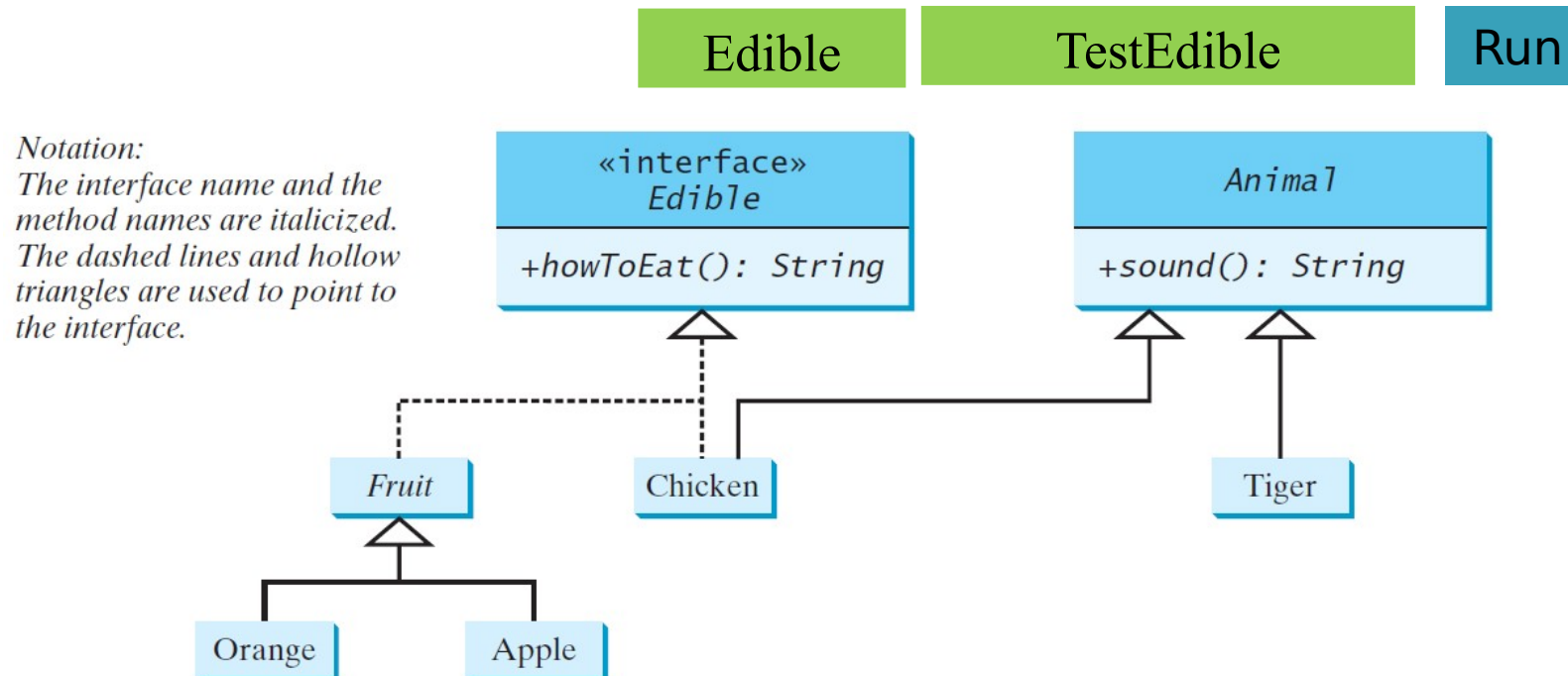
Interface is a Special Class

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 a 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.

«interface»
Edible

+howToEat(): String

Animal

+sound(): String

Fruit

Chicken

Tiger

Orange

Apple

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).

Example: The <u>Comparable</u> Interface

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

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

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.

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
}

}
```

Example

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

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 <u>Cloneable</u> interface is marked cloneable, and its objects can be cloned using the <u>clone()</u> method defined in the <u>Object</u> 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.
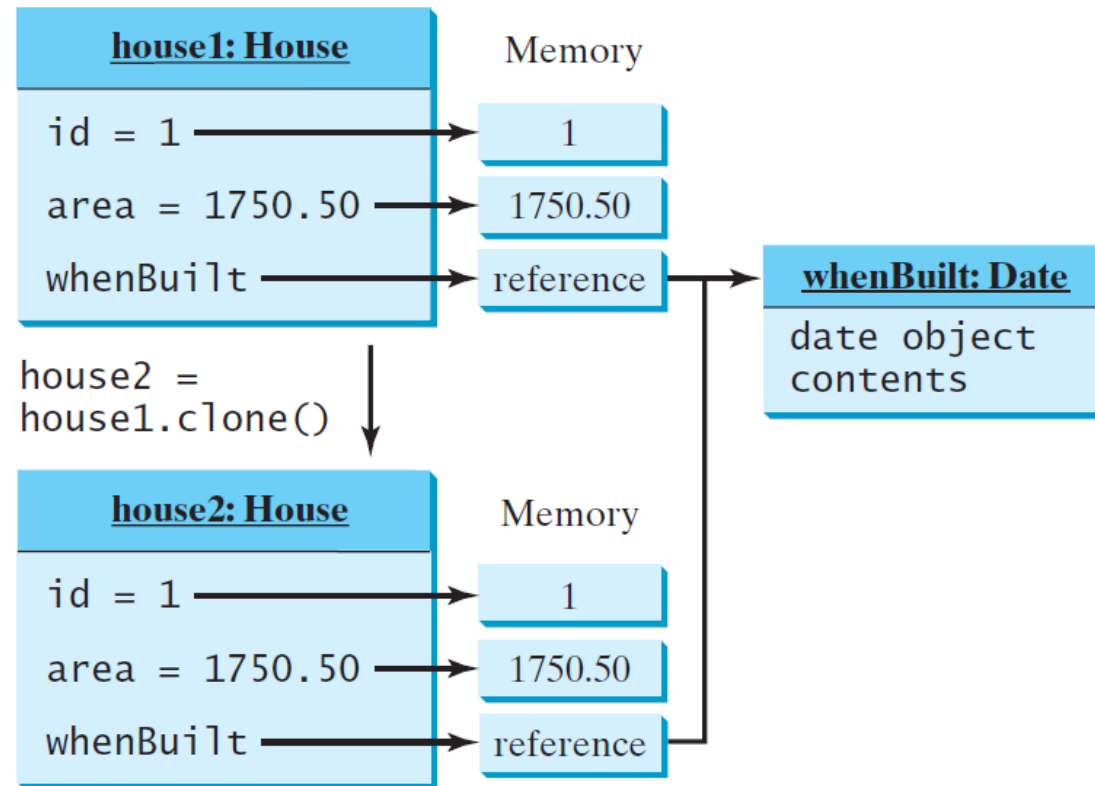
House

Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);

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

Shallow Copy

(a)
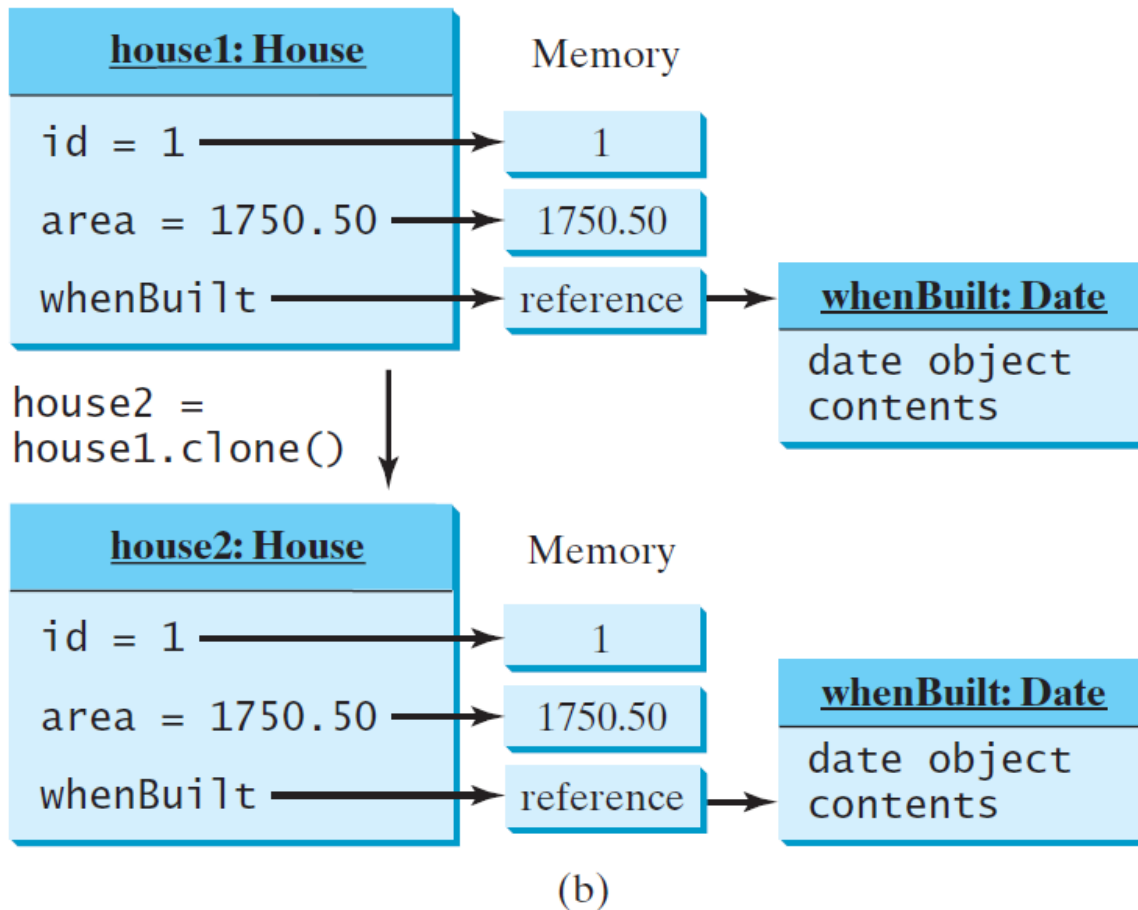
Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);

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

Deep
Copy

(b)

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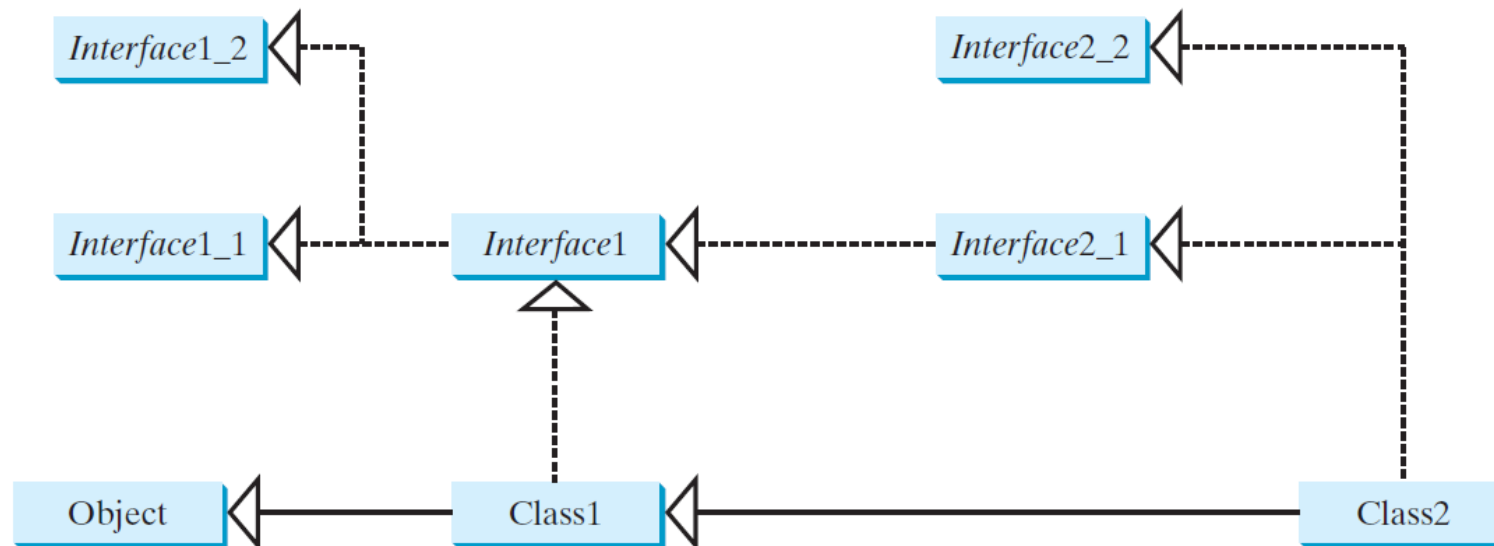
Suppose that c is an instance of Class2. c is also an instance of Object, Class1, Interface1, Interface1_1, Interface1_2, Interface2_1, and Interface2_2.

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.

Next: Generics

Why Do You Get a Warning?

```
public class ShowUncheckedWarning {
  public static void main(String[] args) {
    java.util.ArrayList list =
      new java.util.ArrayList();
    list.add("Java Programming");
  }
}
```

To understand the compile warning on this line, you need to learn JDK 1.6 generics.

Fix the Warning

```
public class ShowUncheckedWarning {
   public static void main(String[] args) {
      java.util.ArrayList<String> list =
         new java.util.ArrayList<String>();
      list.add("Java Programming");
   }
}
```

No compile warning on this line.

What is Generics?

Generics is the capability to parameterize types. With this capability, you can define a class or a method with generic types that can be substituted using concrete types by the compiler. For example, you may define a generic stack class that stores the elements of a generic type. From this generic class, you may create a stack object for holding strings and a stack object for holding numbers. Here, strings and numbers are concrete types that replace the generic type.

Why Generics?

The key benefit of generics is to enable errors to be detected at compile time rather than at runtime. A generic class or method permits you to specify allowable types of objects that the class or method may work with. If you attempt to use the class or method with an incompatible object, a compile error occurs.

Generic Type

```
package java.lang;

public interface Comparable {
    public int compareTo(Object o)

}
```

(a) Prior to JDK 1.5

```
package java.lang;

public interface Comparable<T> {
    public int compareTo(T o)

}
```

(b) JDK 1.5

Generic Instantiation

Runtime error

```
Comparable c = new Date();
System.out.println(c.compareTo("red"));
```

(a) Prior to JDK 1.5

```
Comparable<Date> c = new Date();
System.out.println(c.compareTo("red"));
```

(b) JDK 1.5

Improves reliability

Compile error

Generic ArrayList in JDK 1.5

java.util.ArrayList
+ArrayList()
+add(o: Object): void
+add(index: int, o: Object): void
+clear(): void
+contains(o: Object): boolean
+get(index:int): Object
+indexOf(o: Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): int
+remove(o: Object): boolean
+size(): int
+remove(index: int): boolean
+set(index: int, o: Object): Object

(a) ArrayList before JDK 1.5

java.util.ArrayList<E>
+ArrayList()
+add(o: E): void
+add(index: int, o: E): void
+clear(): void
+contains(o: Object): boolean
+get(index:int): E
+indexOf(o: Object): int
+isEmpty(): boolean
+lastIndexOf(o: Object): int
+remove(o: Object): boolean
+size(): int
+remove(index: int): boolean
+set(index: int, o: E): E

(b) ArrayList since JDK 1.5

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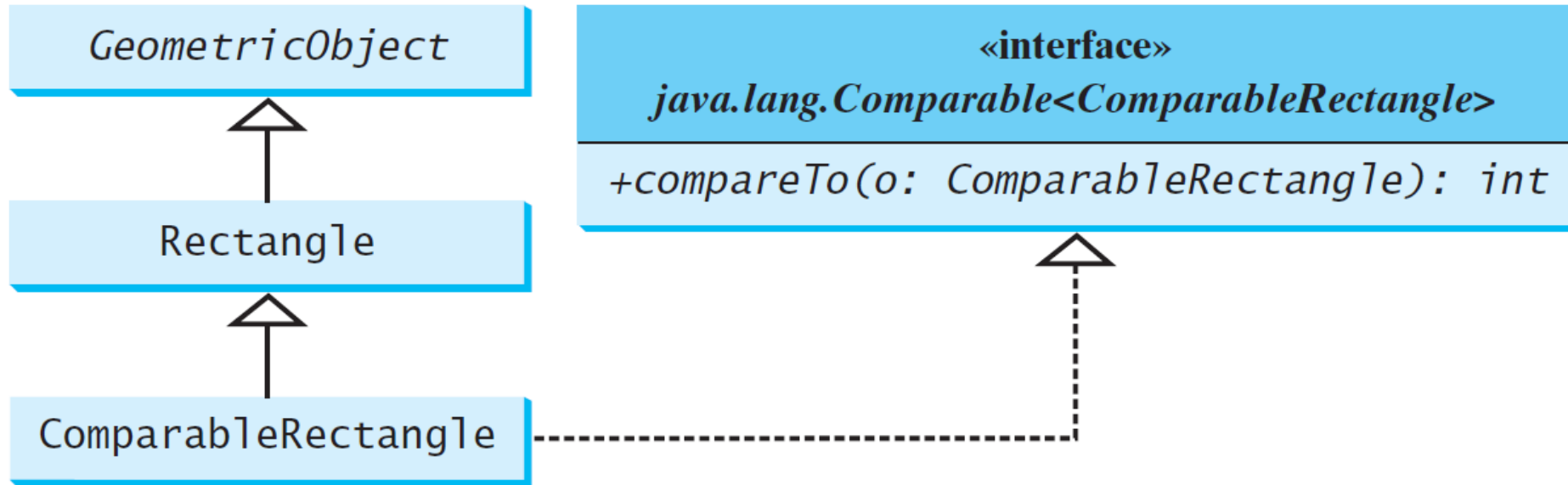
```
d instanceof java.util.Date
d instanceof Object
d instanceof Comparable
```

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

No Casting Needed

ArrayList<Double> list = new ArrayList<>();

list.add(5.5); // 5.5 is automatically converted to new Double(5.5)

list.add(3.0); // 3.0 is automatically converted to new Double(3.0)

Double doubleObject = list.get(0); // No casting is needed

double d = list.get(1); // Automatically converted to double

Declaring Generic Classes and Interfaces

GenericStack<E>	
-list: java.util.ArrayList<E>	An array list to store elements.
+GenericStack()	Creates an empty stack.
+getSize(): int	Returns the number of elements in this stack.
+peek(): E	Returns the top element in this stack.
+pop(): E	Returns and removes the top element in this stack.
+push(o: E): void	Adds a new element to the top of this stack.
+isEmpty(): boolean	Returns true if the stack is empty.

GenericStack

Generic Methods

```java
public static <E> void print(E[] list) {
    for (int i = 0; i < list.length; i++)
        System.out.print(list[i] + " ");
    System.out.println();
}




public static void print(Object[] list) {
    for (int i = 0; i < list.length; i++)
        System.out.print(list[i] + " ");
    System.out.println();
}
```

Bounded Generic Type

```
public static void main(String[] args ) {
    Rectangle rectangle = new Rectangle(2, 2);
    Circle circle = new Circle (2);
    System.out.println("Same area? " + equalArea(rectangle, circle));
}

public static <E extends GeometricObject> boolean
    equalArea(E object1, E object2) {
    return object1.getArea() == object2.getArea();
}
```

Raw Type and Backward Compatibility

// raw type

ArrayList list = new ArrayList();

This is *roughly* equivalent to
ArrayList<Object> list = new ArrayList<Object>();

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.

Raw Type is Unsafe

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

Runtime Error:

Max.max("Welcome", 23);

Make it Safe

```
// Max1.java: Find a maximum object
public class Max1 {
  /** Return the maximum between two objects */
  public static <E extends Comparable<E>> E max(E o1, E o2) {
    if (o1.compareTo(o2) > 0)
      return o1;
    else
      return o2;
  }
}

Max.max("Welcome", 23);
```

Wildcards

Why wildcards are necessary? See this example.

WildCardNeedDemo

?   unbounded wildcard
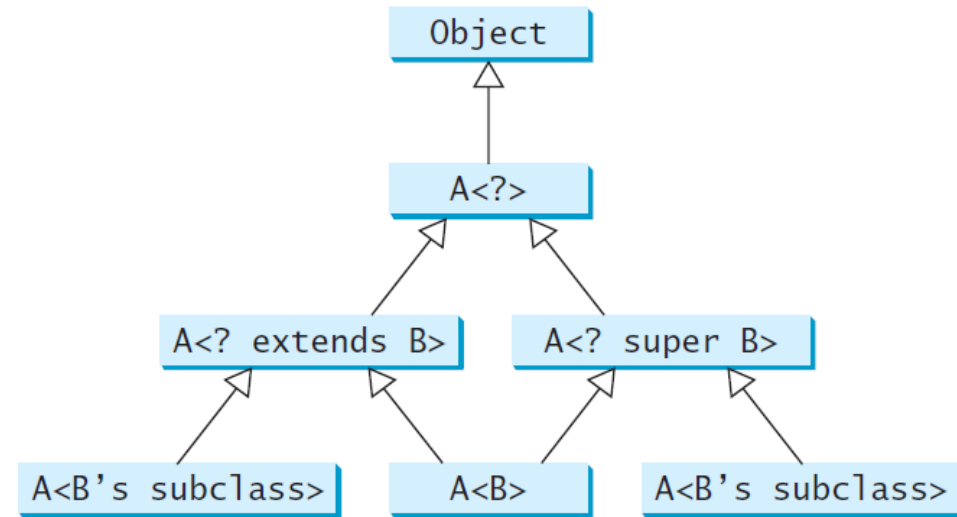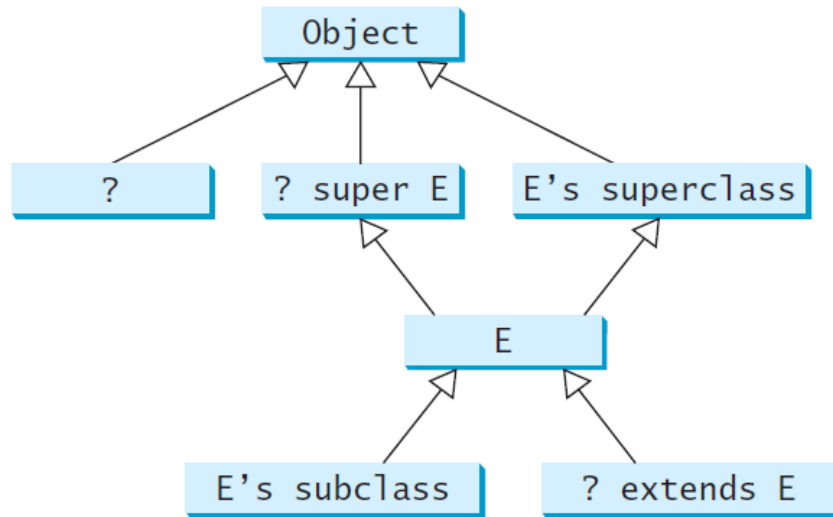
? extends T   bounded wildcard

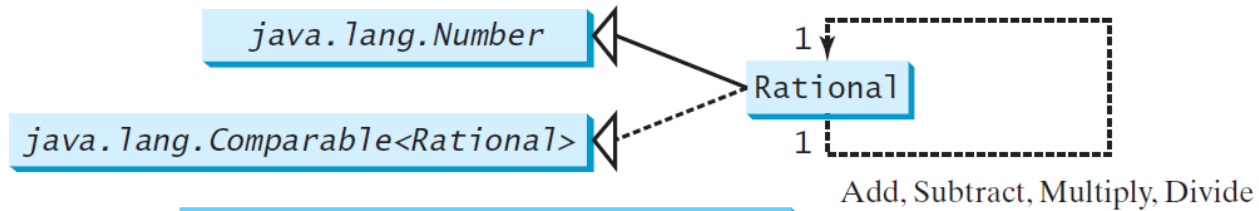? super T   lower bound wildcard

AnyWildCardDemo   SuperWildCardDemo

Generic Types and Wildcard Types

End

The `Rational` Class

```
java.lang.Number
java.lang.Comparable<Rational>
```

```
                    1
            Rational
                    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

Designing a Class

(Coherence) A class should describe a single entity, and all the class operations should logically fit together to support a coherent purpose. You can use a class for students, for example, but you should not combine students and staff in the same class, because students and staff have different entities.

Designing a Class, cont.

(Separating responsibilities) A single entity with too many responsibilities can be broken into several classes to separate responsibilities. The classes String, StringBuilder, and StringBuffer all deal with strings, for example, but have different responsibilities. The String class deals with immutable strings, the StringBuilder class is for creating mutable strings, and the StringBuffer class is similar to StringBuilder except that StringBuffer contains synchronized methods for updating strings.

Designing a Class, cont.

Classes are designed for reuse. Users can incorporate classes in many different combinations, orders, and environments. Therefore, you should design a class that imposes no restrictions on what or when the user can do with it, design the properties to ensure that the user can set properties in any order, with any combination of values, and design methods to function independently of their order of occurrence.

Designing a Class, cont.

Provide a public no-arg constructor and override the <u>equals</u> method and the <u>toString</u> method defined in the <u>Object</u> class whenever possible.

Designing a Class, cont.

Follow standard Java programming style and naming conventions. Choose informative names for classes, data fields, and methods. Always place the data declaration before the constructor, and place constructors before methods. Always provide a constructor and initialize variables to avoid programming errors.

Using Visibility Modifiers

Each class can present two contracts – one for the users of the class and one for the extenders of the class. Make the fields private and accessor methods public if they are intended for the users of the class. Make the fields or method protected if they are intended for extenders of the class. The contract for the extenders encompasses the contract for the users. The extended class may increase the visibility of an instance method from protected to public, or change its implementation, but you should never change the implementation in a way that violates that contract.

Using Visibility Modifiers, cont.

A class should use the private modifier to hide its data from direct access by clients. You can use get methods and set methods to provide users with access to the private data, but only to private data you want the user to see or to modify. A class should also hide methods not intended for client use. The gcd method in the Rational class is private, for example, because it is only for internal use within the class.

Using the static Modifier

A property that is shared by all the instances of the class should be declared as a static property.