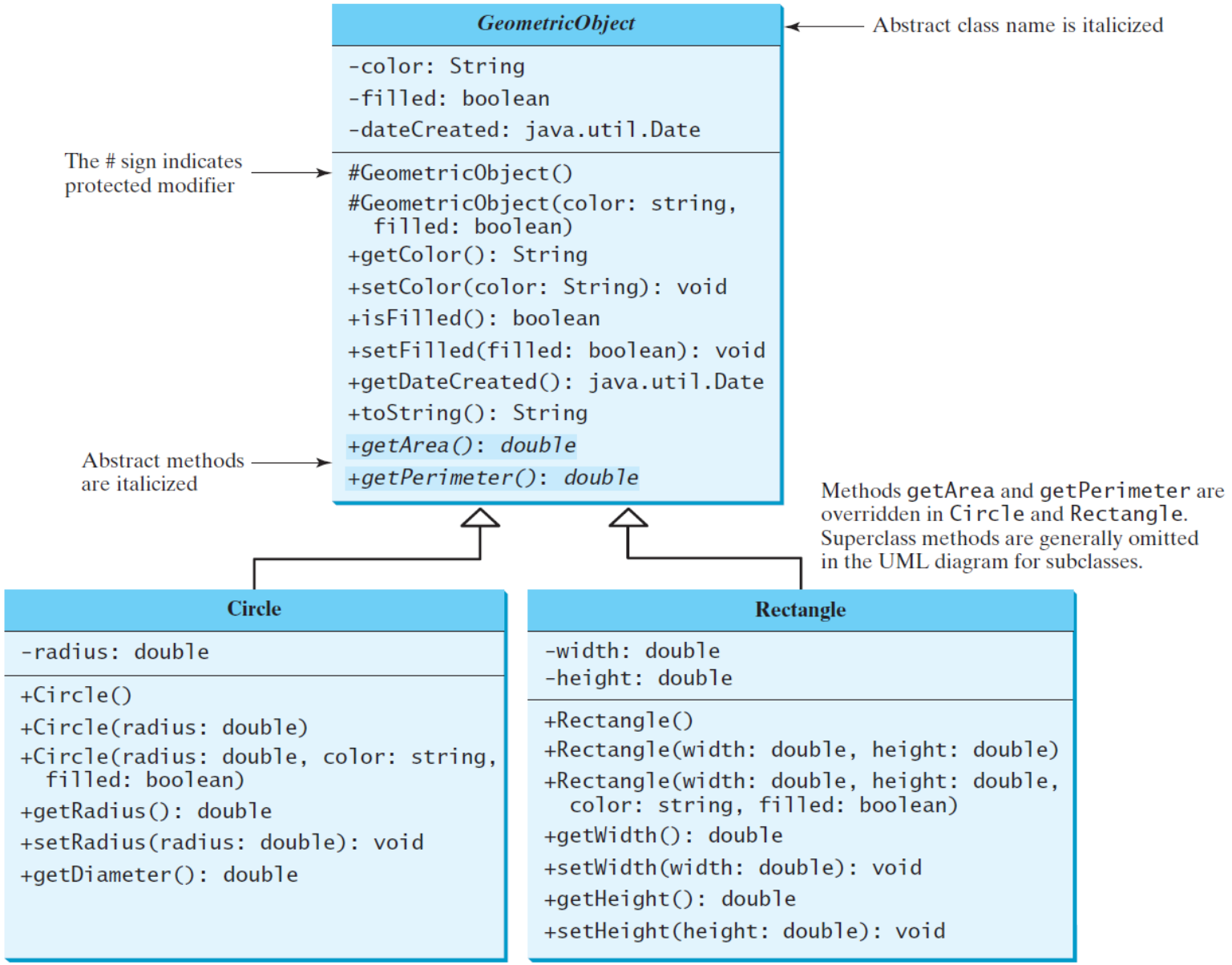
NYU

Lecture 10

Abstract Classes and Interfaces

Abstract Classes and Methods



Abstract Classes and Methods

```
public abstract class GeometricObject {
  private String color = "white";
  private boolean filled;
  private java.util.Date dateCreated;

  protected GeometricObject() {
    dateCreated = new java.util.Date();
  }

  ...

  /** Abstract method getArea */
  public abstract double getArea();

  /** Abstract method getPerimeter */
  public abstract double getPerimeter();
}
```

Abstract Classes and Methods

```
public class Circle extends GeometricObject {
  private double radius;
  public Circle() {
  }

  ...

  public double getArea() {
    return radius * radius * Math.PI;
  }

  public double getPerimeter() {
    return 2 * radius * Math.PI;
  }
}
```

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

Objects 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 and 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

- 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 becomes 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.

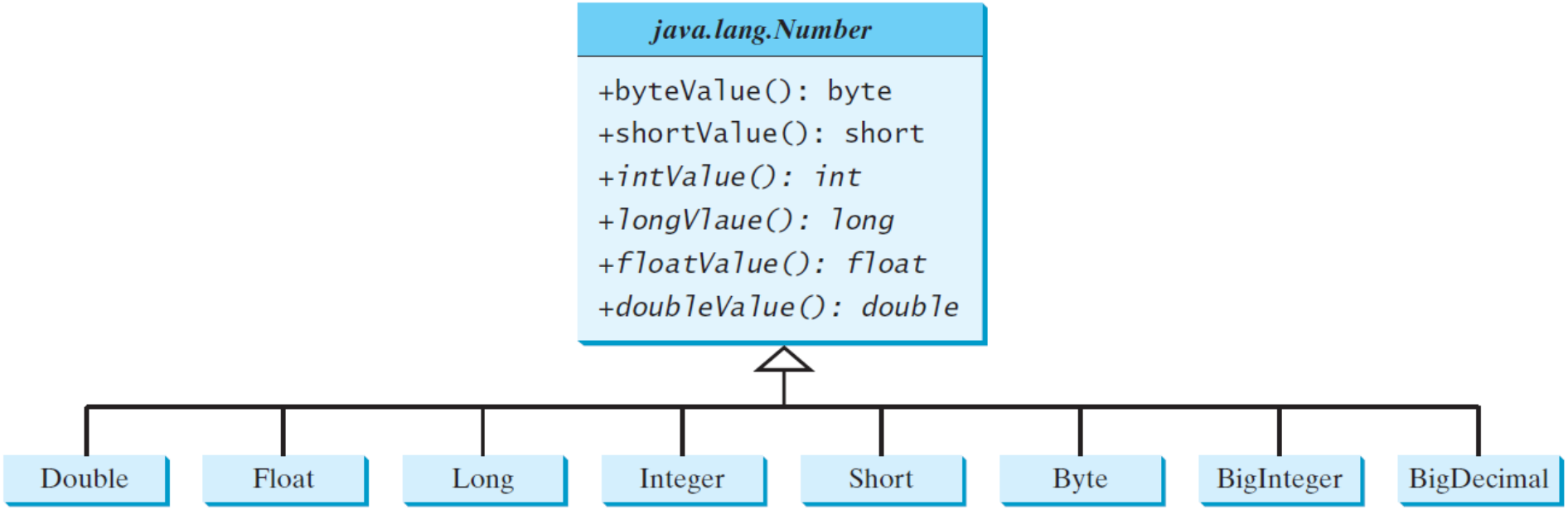
Abstract class as type

- You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type.

- Example

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

Abstract Number Class



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?

- 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();
}
```

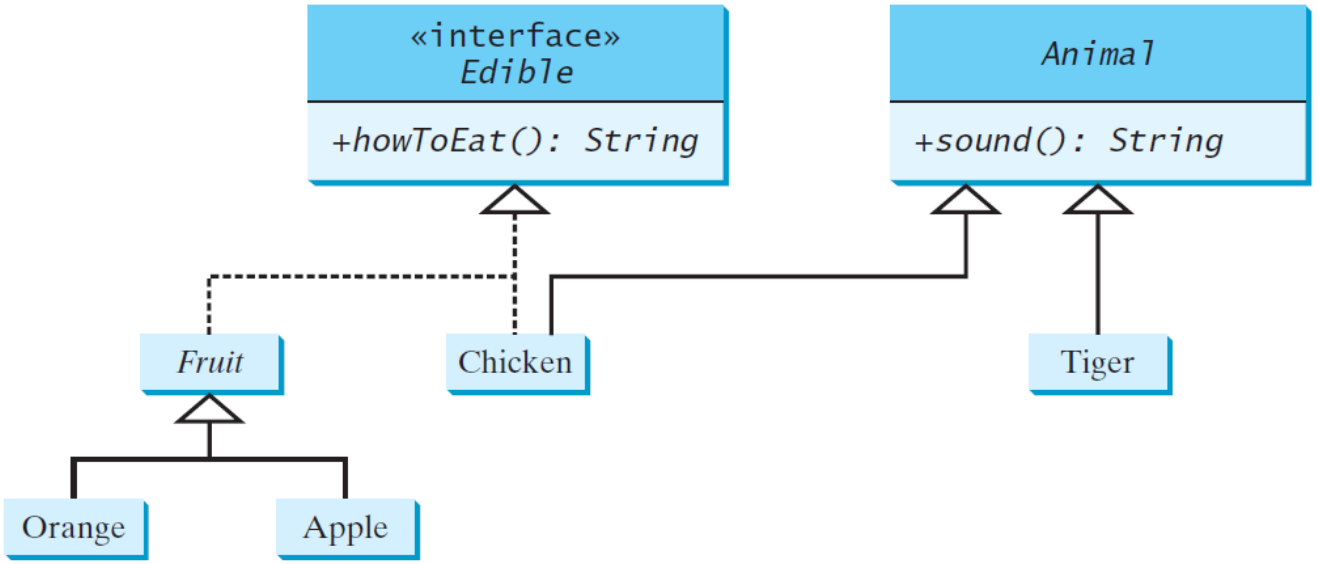
Interface is a Special Class

- 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 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.



Example

```
public abstract class Animal {
  private double weight;

  public double getWeight() {
    return weight;
  }

  public void setWeight(double weight) {
    this.weight = weight;
  }

  /** Return animal sound */
  public abstract String sound();
}
```

Example

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

  @Override
  public String sound() {
    return "Chicken: cock-a-doodle-doo";
  }
}
```

Example

```
class Tiger extends Animal {
  @Override
  public String sound() {
    return "Tiger: RROOAARR";
  }
}
```

Example

```
abstract class Fruit implements Edible {
  // Data fields, constructors, and methods
omitted here
}

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

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
- Example:

```
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

Example: The Comparable Interface

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

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

toString, equals, and hashCode

- 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

```
public class Integer extends Number
                    implements Comparable<Integer> {
  ...
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}

public class BigInteger extends Number
                 implements Comparable<BigInteger> {
  ...
  public int compareTo(BigInteger o) {
    // Implementation omitted
  }
}
```

String and Date Classes

```
public class String extends Object
                    implements Comparable<String> {
   ...
   public int compareTo(String o) {
     // Implementation omitted
   }
}

public class Date extends Object
                         implements Comparable<Date> {
    ...
    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));
  ```
  ```

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
```

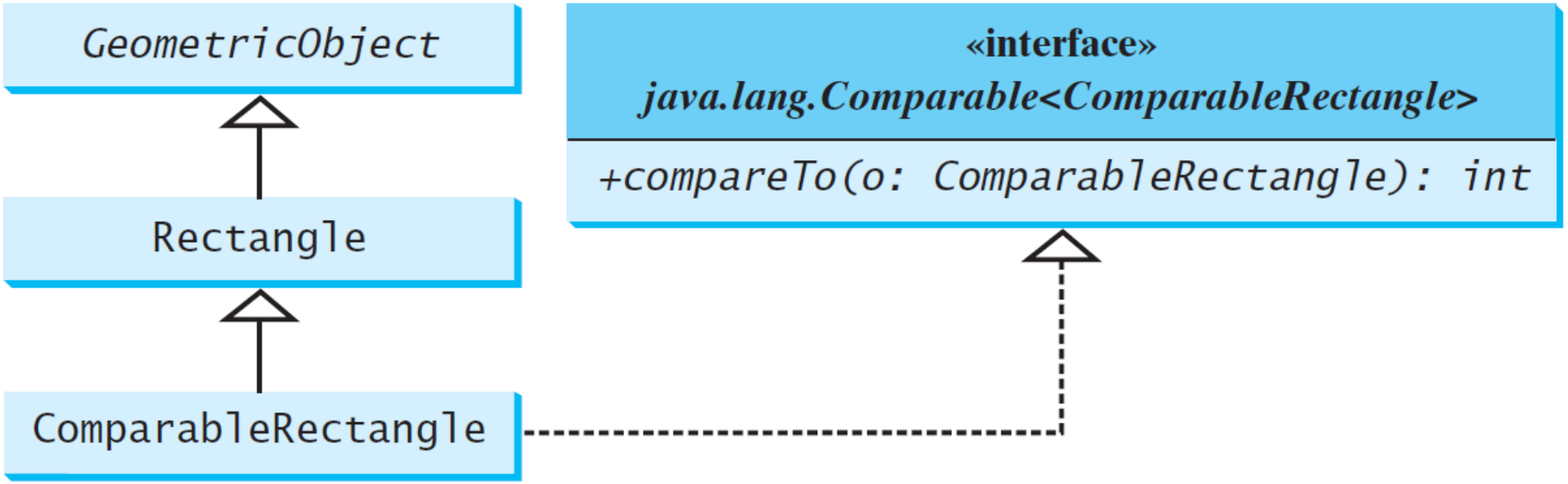
Generic sort Method

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

```
String[] cities =
      {"Savannah", "Boston", "Atlanta”}
java.util.Arrays.sort(cities);

BigInteger[] hugeNumbers ={ ... };
java.util.Arrays.sort(hugeNumbers);
```

Implement Comparable



ComparableRectangle

```
public class ComparableRectangle
   extends Rectangle
   implements Comparable<ComparableRectangle> {
   ...

public int compareTo(ComparableRectangle o) {
     if (getArea() > o.getArea())
       return 1;
     else if (getArea() < o.getArea())
       return -1;
     else
       return 0;
   }
}
```

Sort rectangles

```
public class SortRectangles {
  public static void main(String[] args) {
    ComparableRectangle[] rectangles = {
      new ComparableRectangle(3.4, 5.4),
      new ComparableRectangle(13.24, 55.4),
      new ComparableRectangle(7.4, 35.4),
      new ComparableRectangle(1.4, 25.4)};

    java.util.Arrays.sort(rectangles);
    for (Rectangle rectangle: rectangles) {
      System.out.print(rectangle + " ");
      System.out.println();
    }
  }
}
```

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.

- Example:

```
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

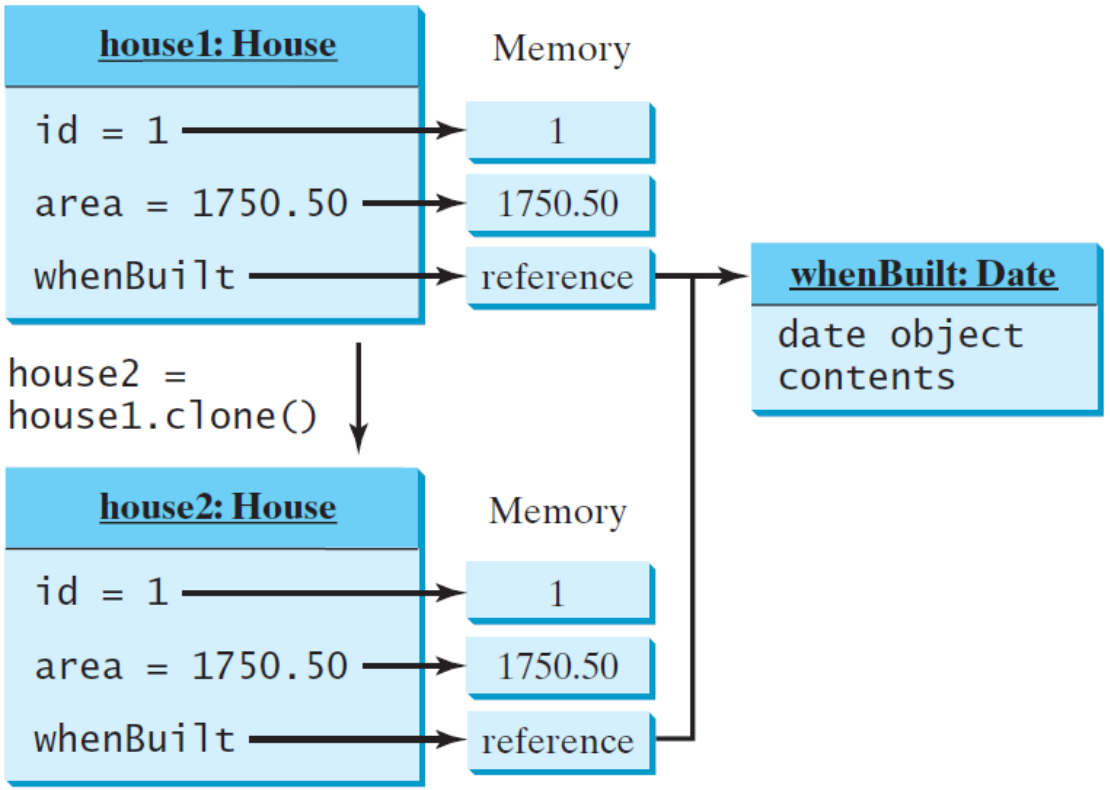
Example

```
public class House implements Cloneable,
Comparable<House> {
    ...
    public Object clone() {
     try {
       return super.clone();
     }
     catch (CloneNotSupportedException ex) {
       return null;
     }
   }
}
```

Shallow vs. Deep Copy

```
House house1 = new House(1, 1750.50);

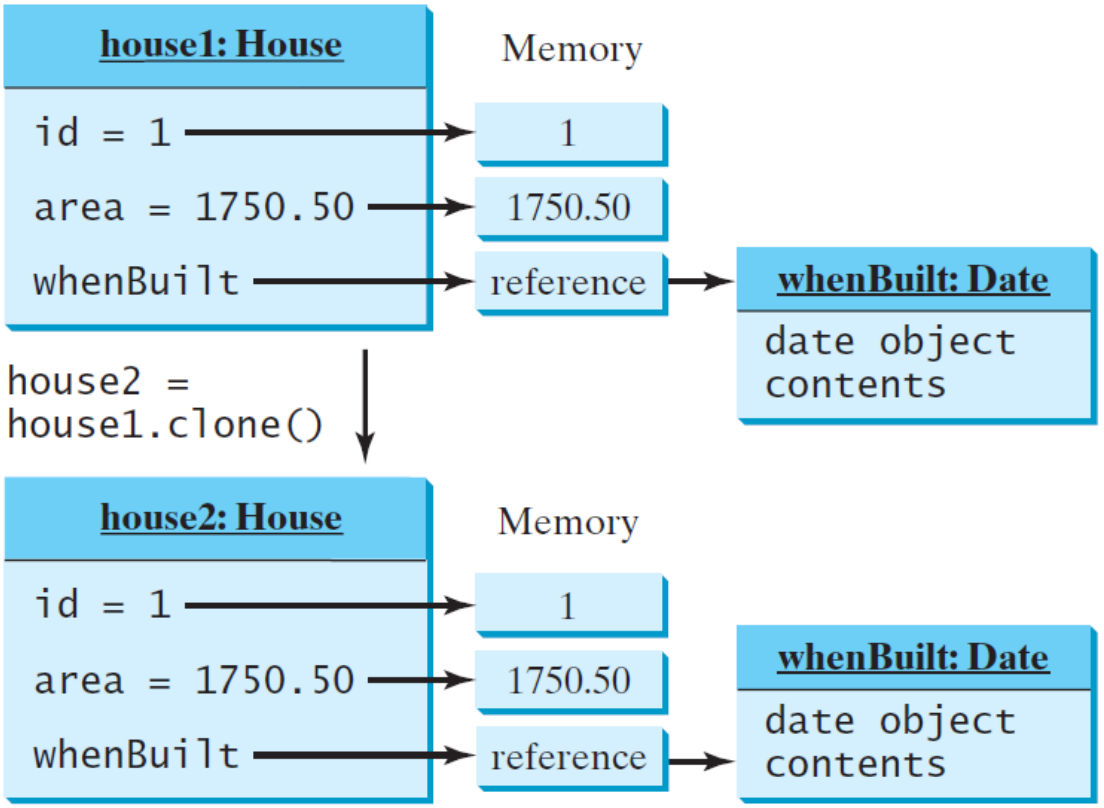
House house2 = (House)house1.clone();
```



Shallow vs. **Deep Copy**

```
House house1 = new House(1, 1750.50);

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



Interfaces vs. Abstract Classes

- In an interface, the data must be constant
- 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.

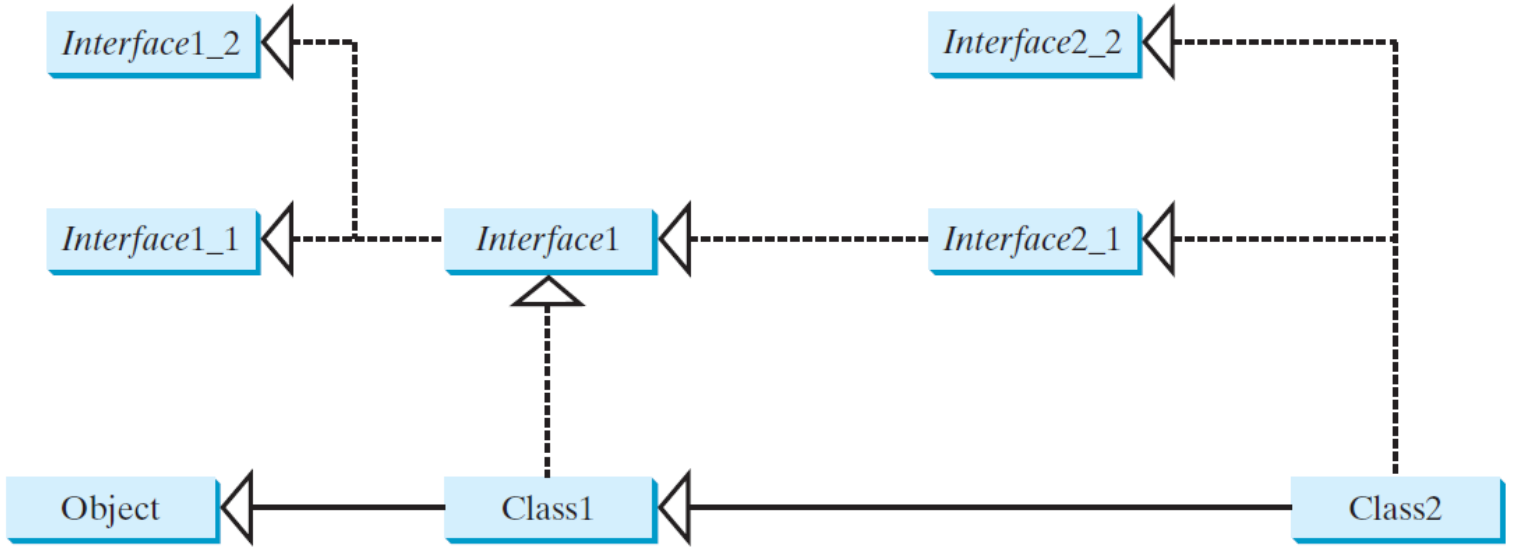
	Variables	*Constructors*	*Methods*
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be `public static final`.	No constructors. An interface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

Interfaces vs. Abstract Classes

- 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.

Interfaces vs. Abstract Classes

- 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
  - 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.

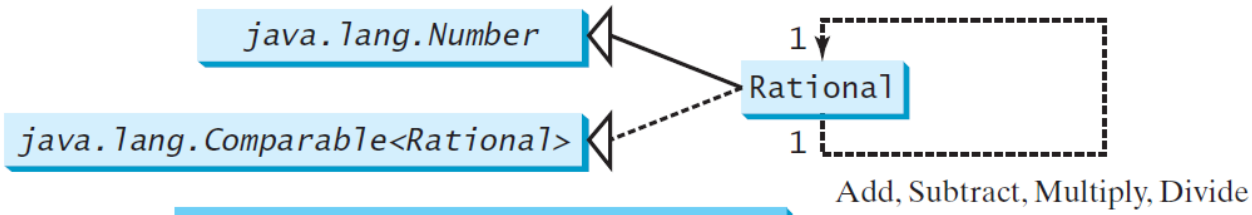
Interface or class?

- 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.
- Example:
  - a staff member is a person.

Interface or class?

- 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.
- 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.

The Rational Class



Rational
-numerator: long -denominator: long
+Rational() +Rational(numerator: long, denominator: long) +getNumerator(): long +getDenominator(): long +add(secondRational: Rational): Rational +subtract(secondRational: Rational): Rational +multiply(secondRational: Rational): Rational +divide(secondRational: Rational): Rational +toString(): String -gcd(n: long, d: long): long

The numerator of this rational number.
The denominator of this rational number.
Creates a rational number with numerator 0 and denominator 1.
Creates a rational number with a specified numerator and denominator.
Returns the numerator of this rational number.
Returns the denominator of this rational number.
Returns the addition of this rational number with another.
Returns the subtraction of this rational number with another.
Returns the multiplication of this rational number with another.
Returns the division of this rational number with another.
Returns a string in the form "numerator/denominator." Returns the numerator if denominator is 1.
Returns the greatest common divisor of n and d.

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

- **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
  - The StringBuffer class is similar to StringBuilder except that StringBuffer contains synchronized methods for updating strings.

Designing a Class

- 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

- Provide
  - a public no-arg constructor
- Override
  - the `equals` method
  - the `toString` method
- Use **common sense**

Designing a Class

- 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

- 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.