CIS2571 – Intro to Java

Chapter15 → Abstract Classes and Interfaces

Information consolidated from a variety of textbook and online resources

Topic Objectives

- Abstract Classes
- Interfaces
  - Comparable
    - Sorting an Array of Objects
  - Cloneable
    - Shallow versus Deep Copy
- Interfaces vs. Abstract Classes
- Interface or Class?
- Case Study: Custom Rational Class

Abstract Classes

- Using inheritance, **subclasses** are more specific and concrete than **superclasses**
  - **Superclass** defines common behavior for related subclasses
- **Superclass** that is so abstract where **no instances can be** created with **new** operator is known as an abstract class
  - Denoted by abstract modifier in class header
  - Not required to have abstract methods
  - Constructor for abstract class is **protected**; used only by **subclasses**

```
// abstract class
public abstract class GeometricObject {
  private java.util.date dateCreated;

  // protected constructor used by subclasses
  protected GeometricObject() {
    dateCreated = new java.util.Date();
  }
}
```

Abstract Classes

- Methods that are **defined without implementation**, but are dependent upon subclasses for their specific implementation, are known as abstract methods
  - Denoted by abstract modifier in method header
  - Are **non-static** methods (instance required)
  - Class becomes abstract class
- UML notation uses *italics* for abstract classes and methods

```
// abstract class denoted by abstract methods
public abstract class GeometricObject {

  // abstract methods implemented in subclass
  public abstract double getArea();
  public abstract double getPerimeter();

}
```

Abstract Classes

- **Subclass** can be abstract even if **superclass** is concrete
  - Object class is concrete
- Concrete class extended from abstract class must implement **all** abstract methods, **even if they are not used**
- **Subclass** can override concrete **superclass** method to make it abstract
  - *Unusual*, but useful when method implementation in **subclass** becomes invalid
  - **Subclass** becomes abstract

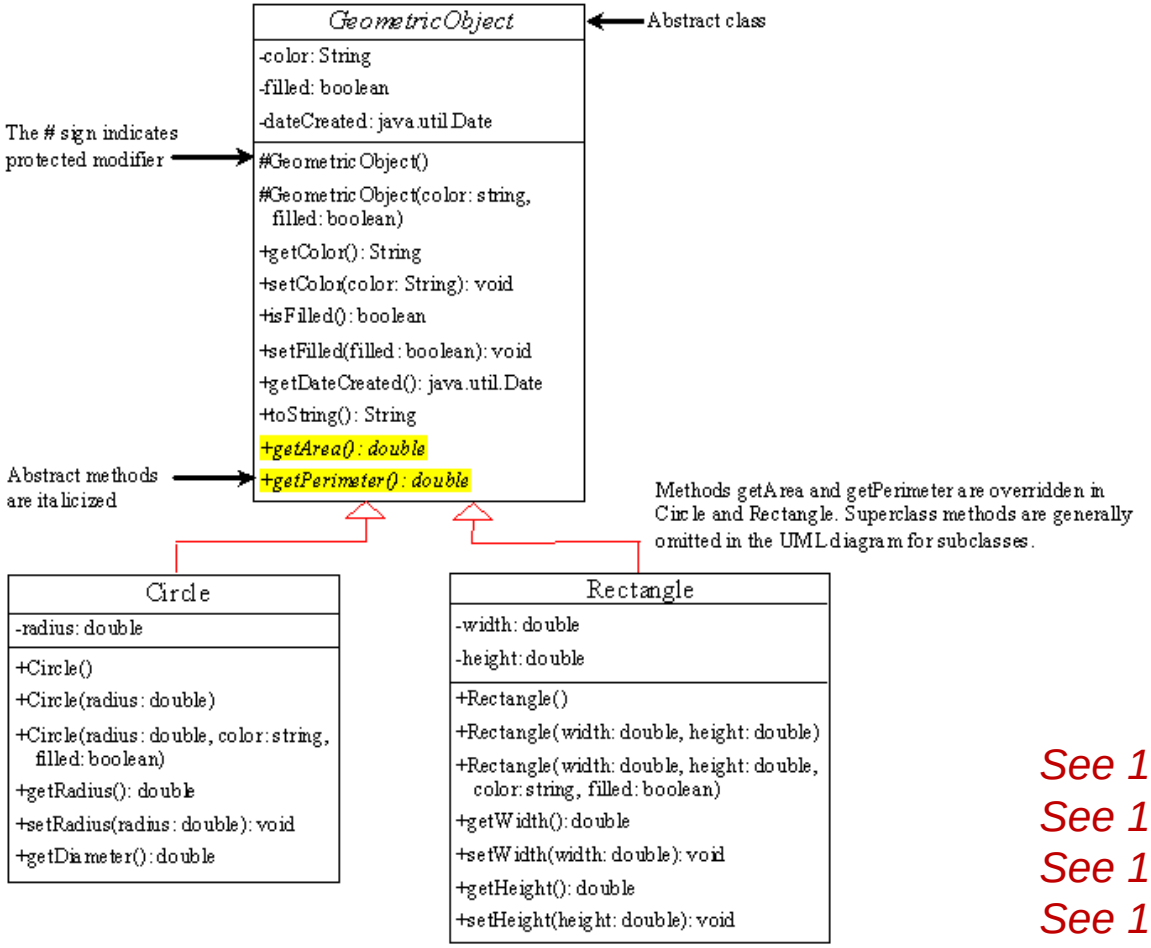
Abstract Classes

- Although abstract classes **cannot** be used to create objects, they **can be** used to create object **reference variables**
  - Can then create instance of concrete **subclass** type and assign to reference variable of abstract **superclass** type
    - generic programming

```
// create array of abstract superclass reference
variables
GeometricObject[] objects = new GeometricObjects[10];

// create and assign instances of concrete subclasses
objects[0] = new Circle();
objects[1] = new Rectangle();
```

→Examples

Abstract Classes Example #1

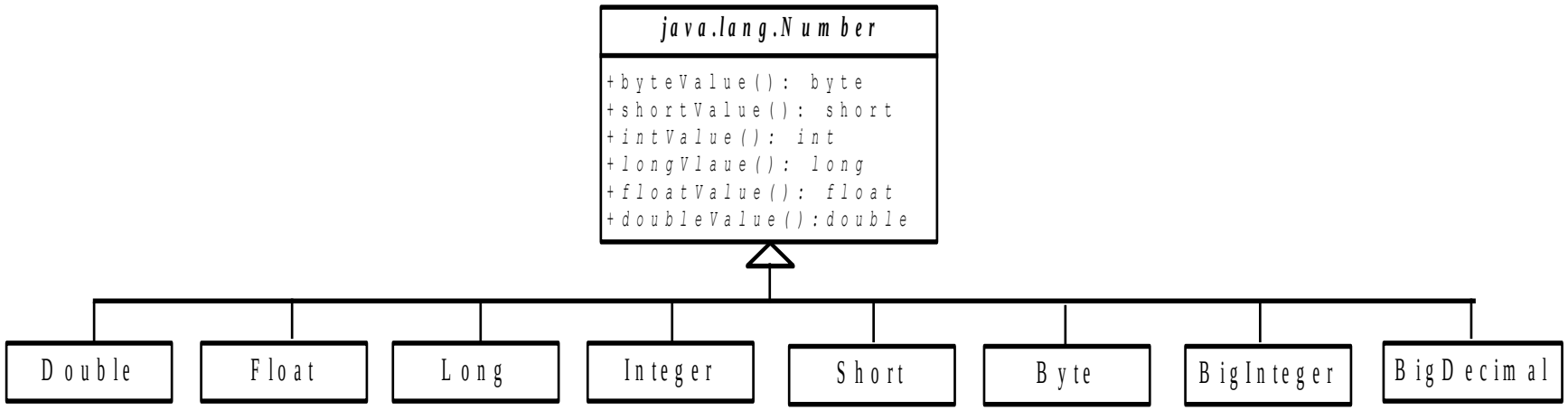


See 15.1 GeometricObject.java
See 15.2 Circle.java
See 15.3 Rectangle.java
See 15.4 TestGeometricObject.java

*case study for student review

Abstract Classes Example #2

- java.lang.Number is an abstract superclass for numeric wrapper classes
  - data conversion methods are abstract methods implemented in concrete subclass wrapper methods

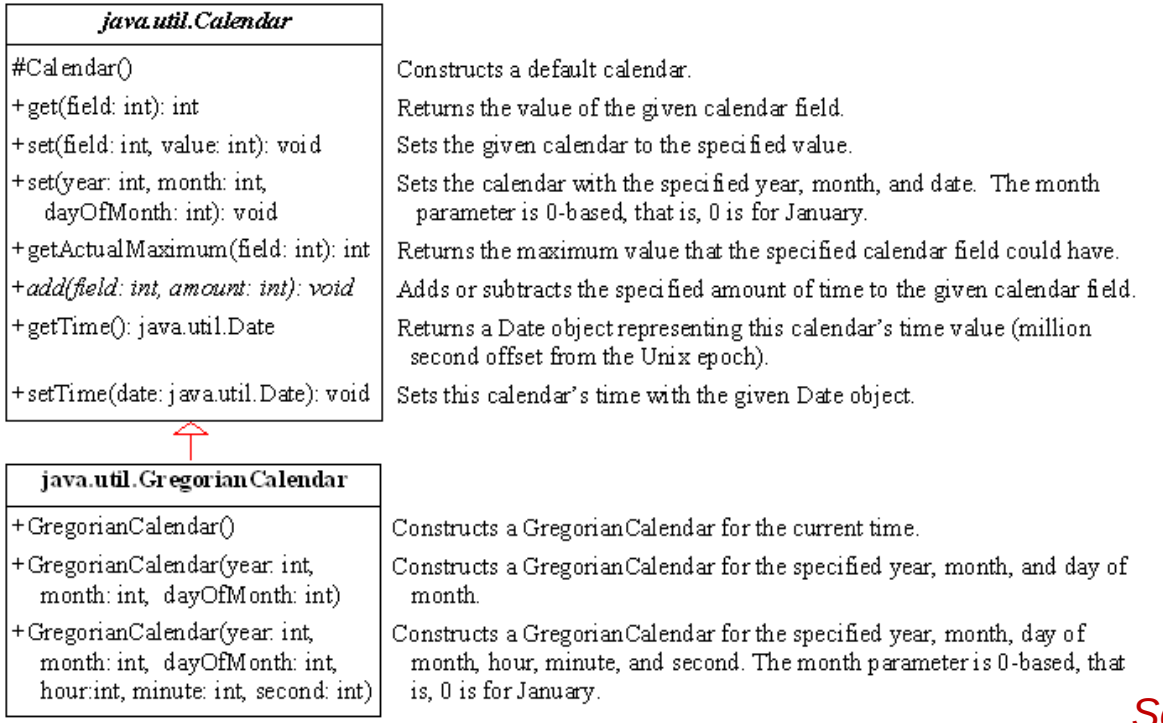


See 15.5 LargestNumbers.java

*case study for student review

Abstract Classes Example #3

- java.util.Calendar is an abstract base class for extracting detailed calendar information such as year, month, date, hour, minute, and second



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 day of month.
+GregorianCalendar(year: int, month: int, dayOfMonth: int, hour:int, minute: int, second: int)	Constructs a GregorianCalendar for the specified year, month, day of month, hour, minute, and second. The month parameter is 0-based, that is, 0 is for January.

See 15.6 TestCalendar.java

Interfaces

- Class-like construct that contains **only** constants and abstract methods
  - Compiled into separate bytecode file like a class
  - Purpose is to specify **common behavior** for objects
  - Provides another form of generic programming
  - Modifiers can be omitted
    - All data fields are **`public final static`**
    - All methods are **`public abstract`**

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

  public abstract void p();
}
```

Equivalent

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

  void p();
}
```

Interfaces

- Similar to abstract class
  - Cannot create instance of interface type using new operator
  - Can use as data type for reference variable
  - Can use in casting
  - UML notation uses *italics* for interface name and methods
- Java format

```
modifier interface InterfaceName {
  /** Constant declarations */
  /** Method signatures */
}
modifier class ClassName implements InterfaceName {
  /** Class declarations */
}
```

- When class implements an interface, it implements **all** methods defined in the interface with the exact signature and return type

→Examples

Interface Example #1

- Comparable Interface used to define a natural order for objects
  - **compareTo** method determines order of **this** object with specified object
    - returns negative integer, zero, or positive integer
    - generic type T replaced by concrete type when implementing interface
    - should be implemented in format consistent with **equals**

```
// Interface for comparing objects, defined in java.lang
package java.lang;
public interface Comparable<T> {
  public int compareTo(T o);
}
```

```
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
  }
}
```

Interface Example #1

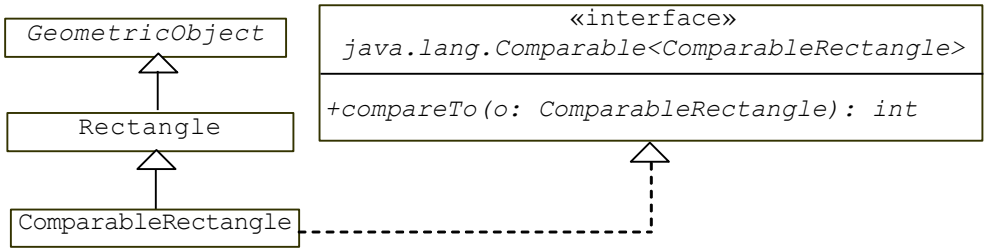
- Sort method used to sort an array of **any** object as long as the class implements the Comparable interface
  - generic programming
  - see java.util.Arrays.sort() static method

```
import java.math.*;

public class SortComparableObjects {
   public static void main(String[] args) {
      String[] cities = {"Savannah", "Boston", "Atlanta", "Tampa"};
      java.util.Arrays.sort(cities);
      for (String city: cities)
         System.out.print(city + " ");
      System.out.println();

      BigInteger[] hugeNumbers = {new BigInteger("2323231092923992"),
         new BigInteger("432232323239292"),
         new BigInteger("54623239292")};
      java.util.Arrays.sort(hugeNumbers);
      for (BigInteger number: hugeNumbers)
         System.out.print(number + " ");
   }
}
```

See 15.8 SortComparableObjects.java

Interface Example #1

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



See 15.9 ComparableRectangle.java
See 15.10 SortRectangles.java

Interface Example #2

- Cloneable Interface used to create a copy of an object
  - marker interface is an empty interface
  - does not contain constants or methods
  - used to denote that a class possesses certain properties

```
// Interface for copying objects, defined in
// java.lang
package java.lang;
public interface Cloneable {
}
```

Interface Example #2

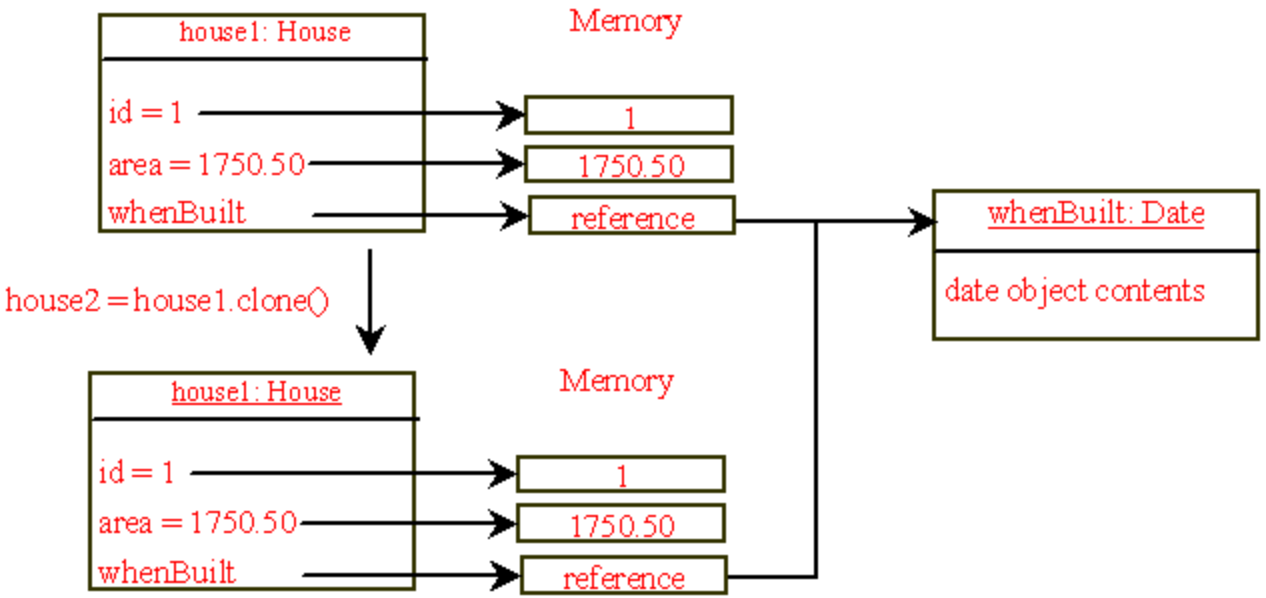
- The Cloneable Interface used to create a copy of an object
  - Custom class implementing **Cloneable** interface **must** override the clone() method in the Object class:

```
protected Object clone() throws
  CloneNotSupportedException;
```

  - protected method requires class implementing this interface override **clone()** method
    - default performs shallow copy; primitive fields and object references are copied
    - for a deep copy, can override **clone()** method with custom cloning operations after invoking **super.clone()**
    - use object **equals()** method to confirm object equality

Interface Example #2 *(shallow copy)*

```
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
```



See 15.11 House.java
See TestHouse.java

Interface Example #2

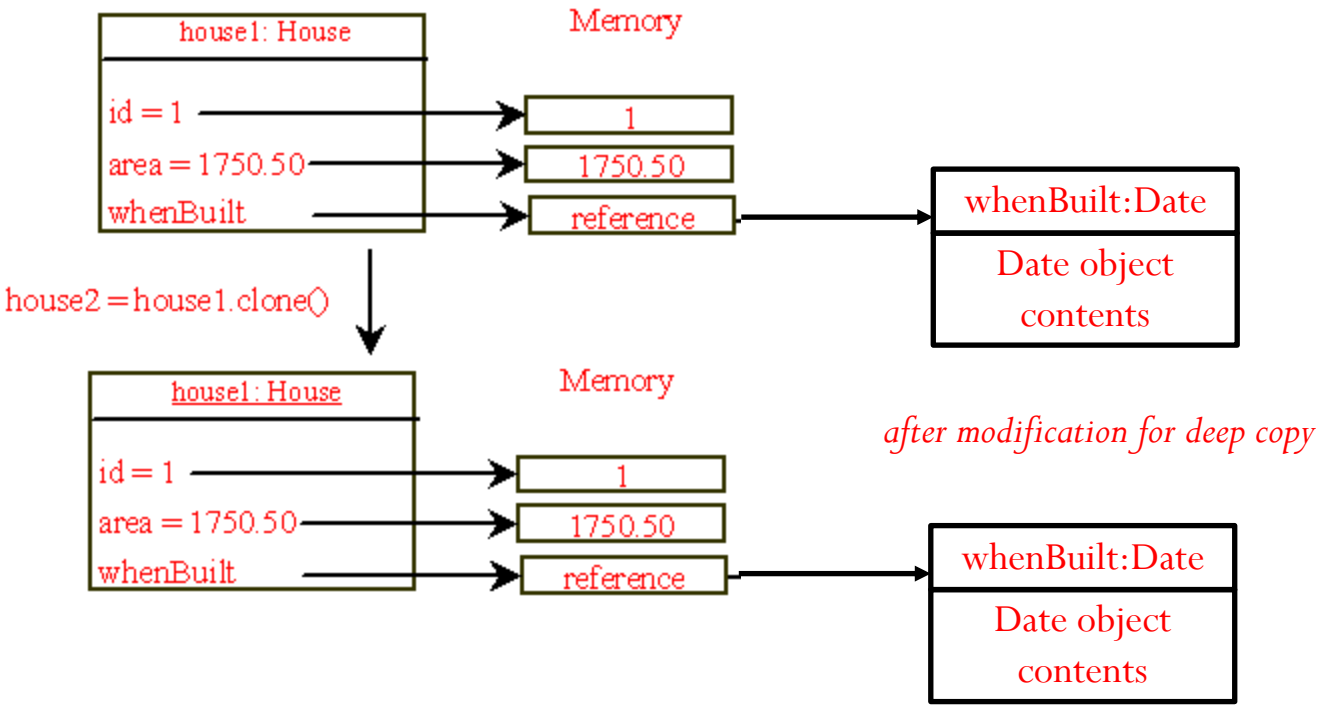
```
@Override /** Override the protected clone method defined in
   the Object class, and strengthen its accessibility */
 public Object clone() throws CloneNotSupportedException {
   return super.clone();
 }
```

shallow versus deep copy

```
@Override /** Override the protected clone method defined in
   the Object class, and strengthen its accessibility */
 public Object clone() throws CloneNotSupportedException {
   // perform a shallow copy
   House houseClone = (House)super.clone();
   // deep copy on whenBuilt
   houseClone.whenBuilt = (java.util.Date)(whenBuilt.clone());
   return houseClone;
 }
```

Interface Example #2 *(deep copy)*

```
House house1 = new House(1, 1750.50);
House house2 = (House)house1.clone();
```



Interface Example #2

deep copy

```
@Override /** Override the protected clone method defined in
   the Object class, and strengthen its accessibility */
 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

- Interface can be used in ways **similar** to an abstract class, but defining an interface is **different** than defining an abstract class

	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 (no implementation)

Interfaces vs. Abstract Classes

- Interfaces and classes define a **type**
- **All** classes share single parent class: Object
- Interfaces **do not** share single root class
- Can only specify **single** class inheritance
  - Classes can **extend** its **superclass** and **implement** multiple interfaces
- Can specify **multiple** interface implementations
  - Be careful not to implement interfaces with conflicting information (i.e. constants with different values, methods with same signature but different return type)
- Interfaces can **extend** other interfaces but **not** classes

Interfaces vs. Abstract Classes

- Interface that inherits other interfaces through **extends** keyword is known as subinterface

```
public interface NewInterface extends Interface1, …,
   InterfaceN {
   // constants and abstract methods
}
```

- Variable of interface type can reference any instance of the class that implements the interface
  - If a class **implements** an interface, interface plays same role as a **superclass** → can use interface as a data type and cast a variable of an interface type to its **subclass**, and vice versa
    - generic programming
- Interfaces *more flexible* than abstract classes because they can define **common** supertype for **unrelated** classes

Interface or Class?

- Strong **is-a** relationship that clearly describes parent-child relationship should be modeled using classes
  - staff member is a person → relationship should be modeled using class inheritance
- Weak **is-a** relationship, also known as **is-kind-of** relationship, indicates that object possesses a certain property and can be modeled using interfaces
  - all strings are comparable → String class implements the Comparable interface
- Use interfaces to circumvent single inheritance restriction if multiple inheritance is desired
  - design class one as a **superclass**, and others as **interfaces**

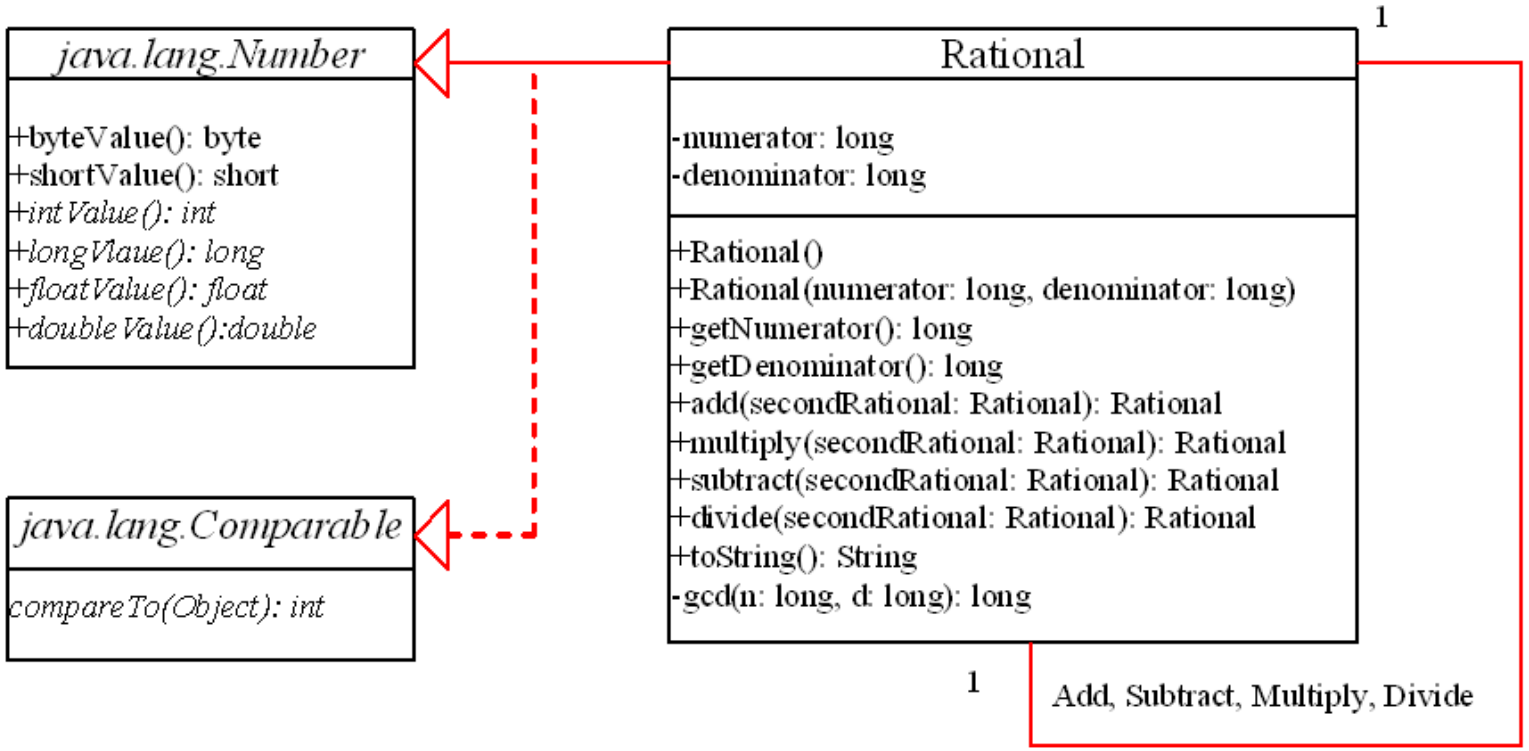
*case study for student review

Case Study: Custom Rational Class

- Rational number has numerator / denominator
  - Rational number **cannot have** denominator of 0
  - Rational number **can have** numerator of 0
- Rational Numbers used in exact computations involving fractions since floating point number cannot precisely represent a rational number
  - 2/3 versus 0.6666
- Rational class extends abstract Number class and implements Comparable interface
  - to reduce a rational number to its lowest terms , find greatest common divisor (GCD) of numerator and denominator

*case study for student review

Case Study: Custom Rational Class



See 15.13 Rational.java
See 15.12 TestRationalClass.java