CISC 3115 TY3

C20b: Class Design Guideline

Hui Chen

Department of Computer & Information Science

CUNY Brooklyn College

Outline

- Discussed
  - Recap
    - Inheritance and polymorphism
    - Abstract method and class
  - Interface
    - Motivation
    - Define interface
    - Extend interface
    - Implement interface
    - Use interface as data type
  - Interfaces in selected Java API
- Class design guidelines
- Reiterate your project considering the guideline

Guideline

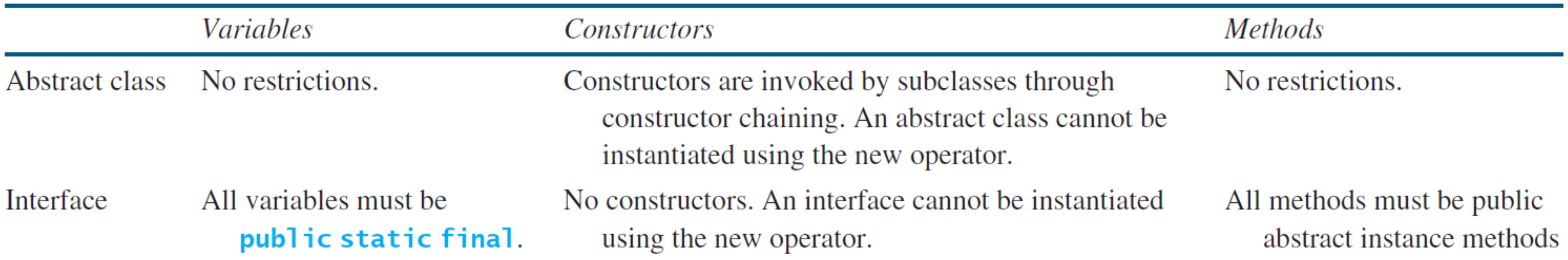
- Consider
  - Cohesion
  - Consistency
  - Encapsulation
  - Clarity
  - Completeness
  - Instance vs. static members
  - Inheritance vs. aggregation
  - Interface vs. abstract class

Interface vs. Abstract Class

- (Since JDK 8)
- 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.

Interface vs. Abstract Class: Data Fields and Methods

- Data fields, constructors, and methods



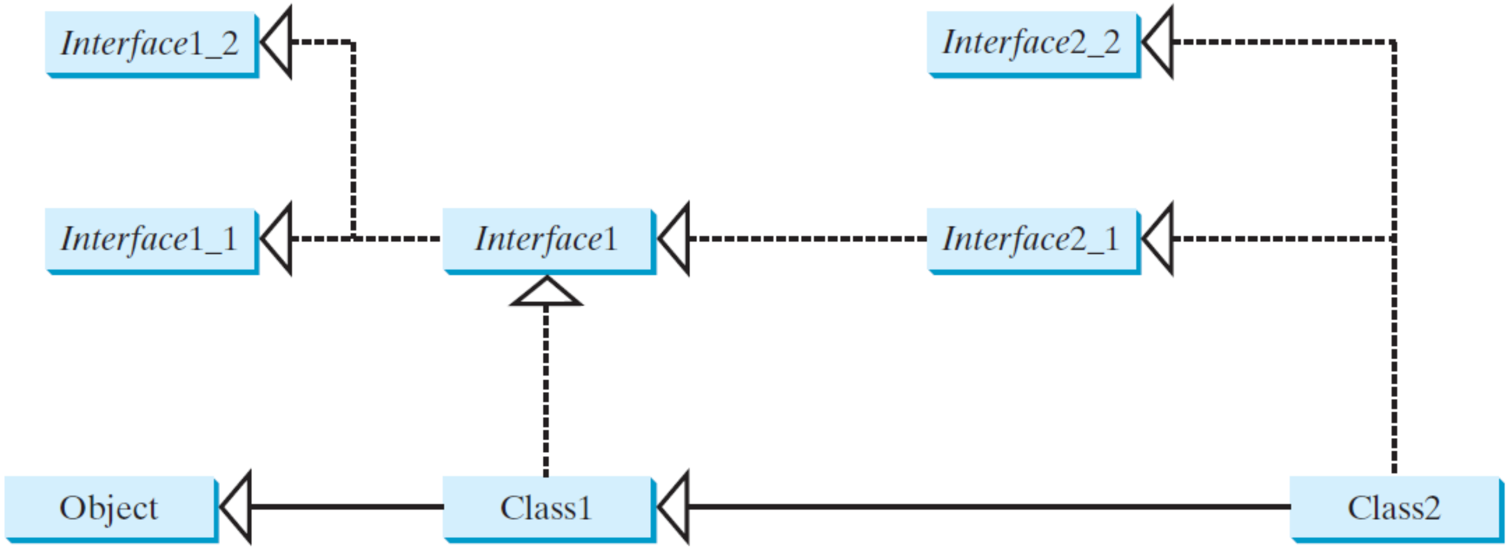
	Variables	*Constructors*	*Methods*
Abstract class	No restrictions.	Constructors are invoked by subclasses through constructor chaining. An abstract class cannot be instantiated using the new operator.	No restrictions.
Interface	All variables must be `public static final`.	No constructors. An interface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

Interface vs. Abstract Class: Class Hierarchy

- 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 implements 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.

Interface vs. Abstract Class: Class Hierarchy

- UML class diagram



Caution: Conflict Interfaces

- In rare occasions, a class may implement two interfaces with conflict information
  - Examples
    - 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 Java compiler.

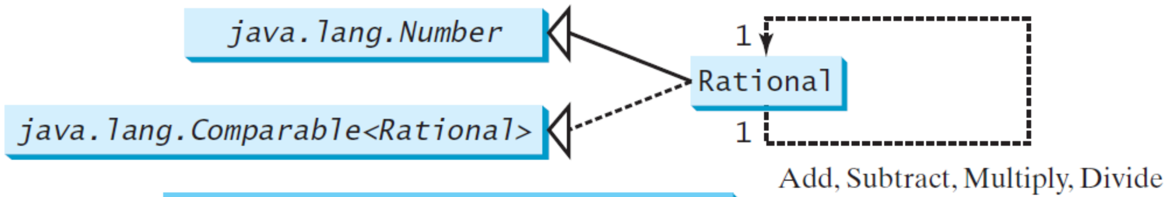
Design Guideline: Interface or Abstract Class?

- 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

Example: The Rational Class

- Just one more example …

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.

Questions?

- Interface vs. abstract class

Coherence

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

Separating Responsibilities

- A single entity with too many responsibilities can be broken into several classes to separate responsibilities.
  - Example: The classes String, StringBuilder, and StringBuffer all deal with strings, 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.

Reuse

- 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

No-arg Constructor, equals, and toString Methods

- Whenever possible,
- provide a public no-arg constructor,
- override the java.lang.Object::equals method (conforming the contract), and
- overrides the java.lang.Object::toString method

Convention and Styles

- 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 the other 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.
- And more in next slide …

Using Visibility Modifiers

- A class should use the private modifier to hide its data from direct access by clients.
- You can use getter methods and setter 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.
  - Example: the gcd method in the Rational class is private because it is intended 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

Questions?

- Consider
  - Cohesion
  - Consistency
  - Encapsulation
  - Clarity
  - Completeness
  - Instance vs. static members
  - Interface vs. abstract class
  - Inheritance vs. aggregation (behavior and states)