UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

Dasar Dasar Pemrograman 2

Acuan: Introduction to Java Programming and Data Structure, Bab 13

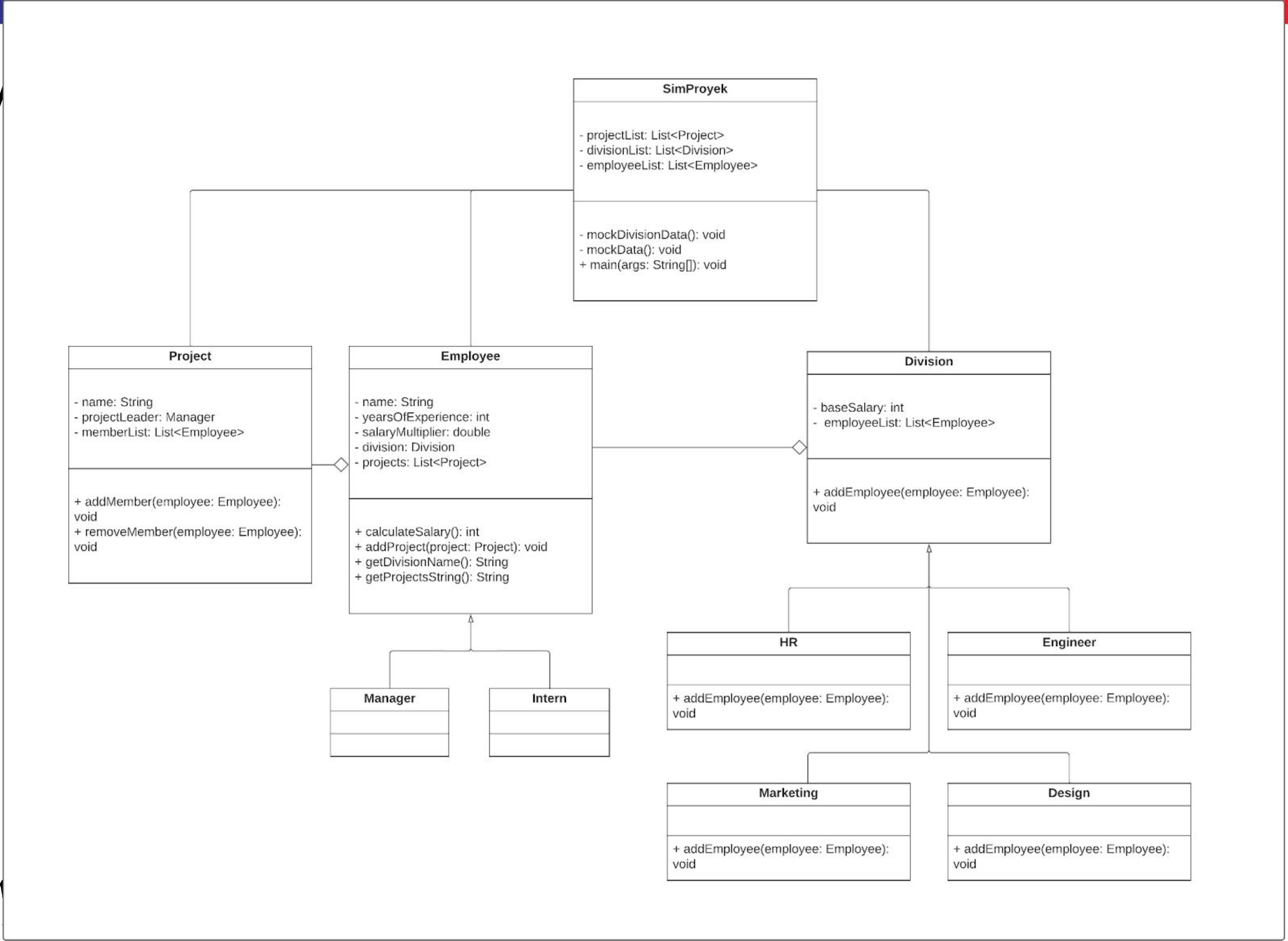
Sumber Slide: Liang

Dimodifikasi untuk Fasilkom UI oleh Ade Azurat

Untuk kalangan terbatas saja. Tidak untuk dipublikasikan terbuka.

Topik: Abstract Classes and Interfaces

Motiv


SimProyek
- projectList: List<Project>
- divisionList: List<Division>
- employeeList: List<Employee>
- mockDivisionData(): void
- mockData(): void
+ main(args: String[]): void
Project
- name: String
- projectLeader: Manager
- memberList: List<Employee>
+ addMember(employee: Employee): void
+ removeMember(employee: Employee): void
Employee
- name: String
- yearsOfExperience: int
- salaryMultiplier: double
- division: Division
- projects: List<Project>
+ calculateSalary(): int
+ addProject(project: Project): void
+ getDivisionName(): String
+ getProjectsString(): String
Division
- baseSalary: int
- employeeList: List<Employee>
+ addEmployee(employee: Employee): void
Manager
Intern
HR
+ addEmployee(employee: Employee): void
Engineer
+ addEmployee(employee: Employee): void
Marketing
+ addEmployee(employee: Employee): void
Design
+ addEmployee(employee: Employee): void

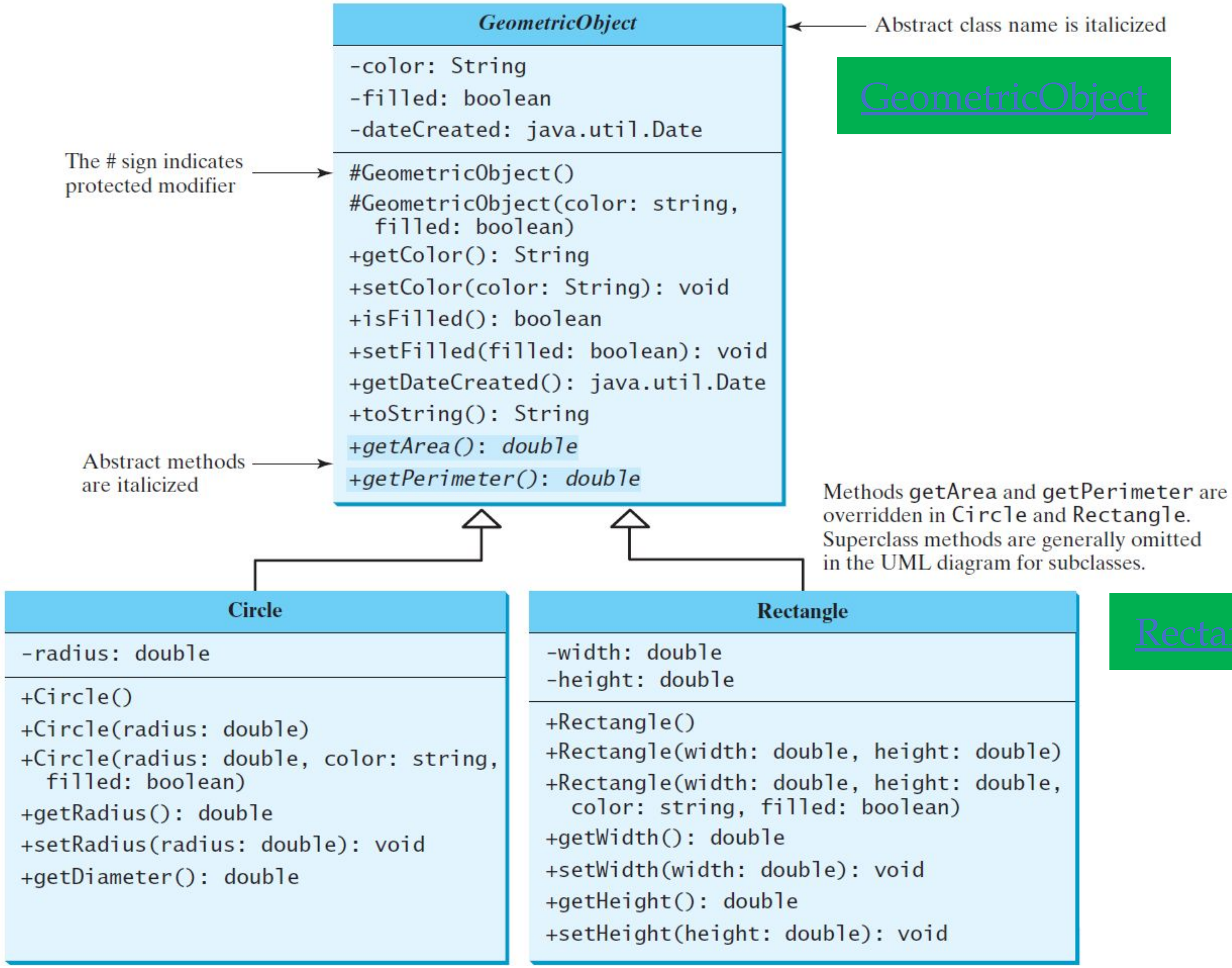
UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FACULTY
COM
SCIE

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).
- To design the **Rational** class for processing rational numbers (§13.9).
- To design classes that follow the class-design guidelines (§13.10).

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

TestGeometricObject

Circle

Rectangle

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

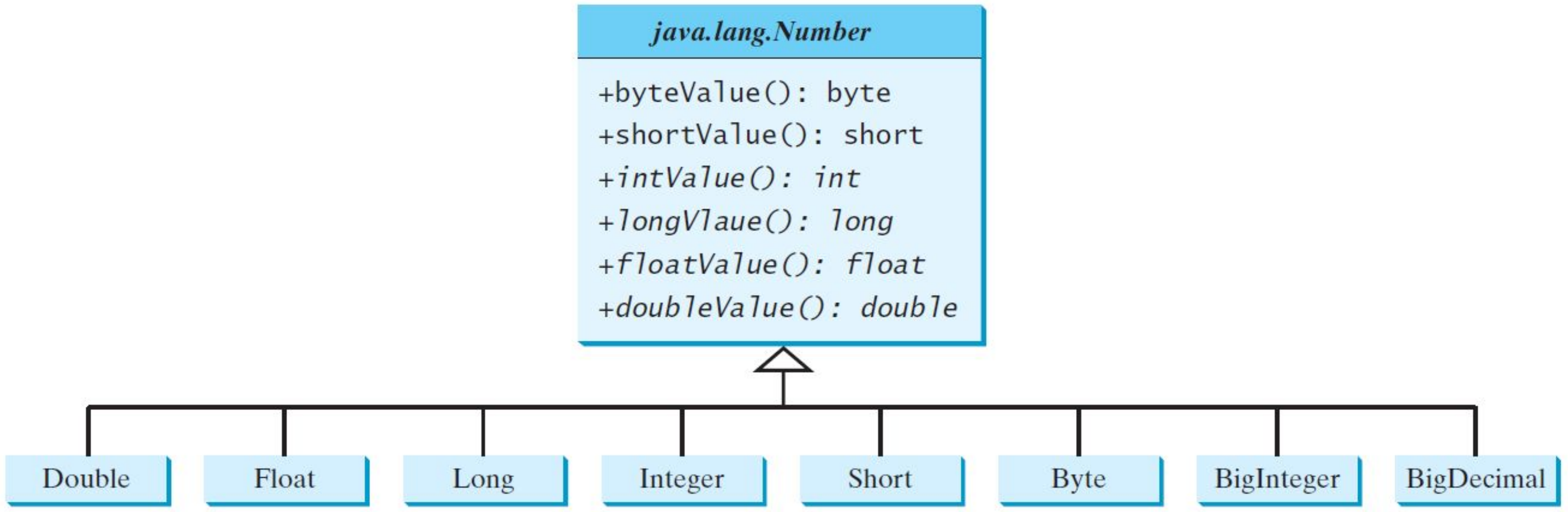
- ✔ A subclass can override a method from its superclass to define it abstract.
- ✔ This is rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass.
- ✔ In this case, the subclass must be defined abstract.

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];
```

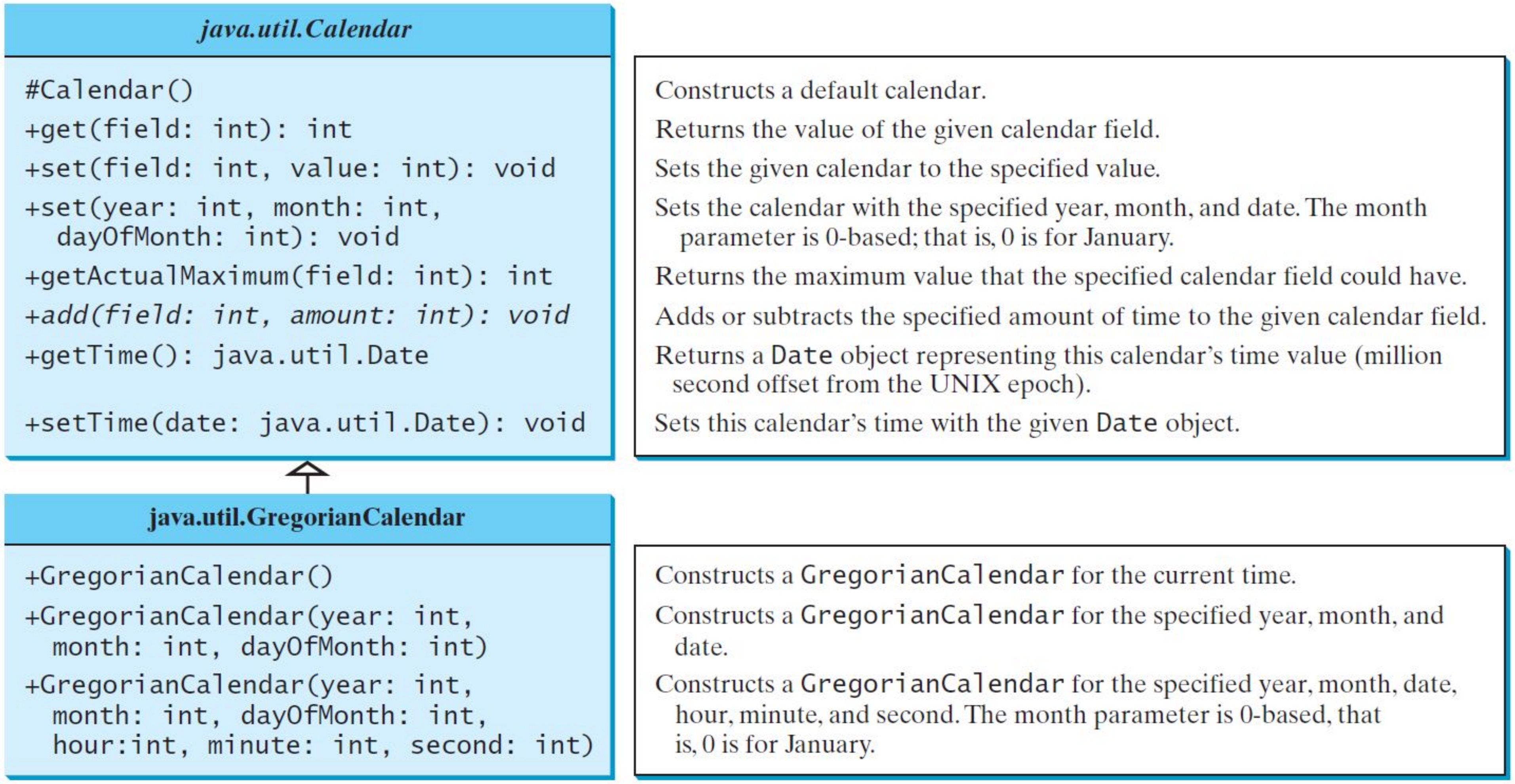
Case Study: the Abstract Number Class



LargestNumbers

The Abstract Calendar Class and Its GregorianCalendar subclass

https://docs.oracle.com/en/java/javase/11/docs/api/java.base/java/util/Calendar.html



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, *Hijria* 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).

TestCalendar

Ayo berhenti dulu!

- ❑ Apa itu *abstract classes*?
- ❑ Kapan sebuah *class* dapat didefinisikan sebagai *abstract class*?
- ❑ Bagaimana hubungan antara *abstract methods* dengan superclass dan subclass?

Interface

What is an interface?
Why is an interface useful?

- ✔ 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:

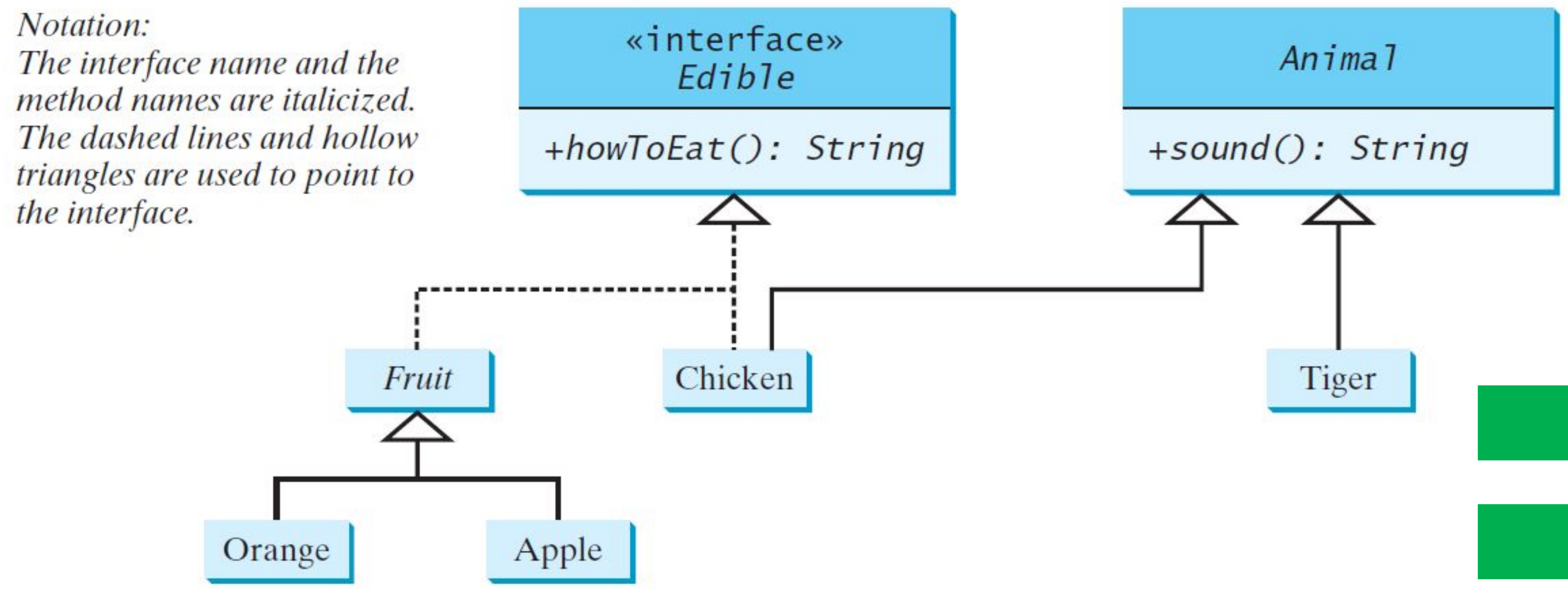
```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

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 (See `TestEdible.java`).


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

Edible

TestEdible

Omitting Modifiers in Interfaces

- ✔ All data **fields are *public final static***
- ✔ 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 Comparable Interface

```
// 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

```
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));
```

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
```

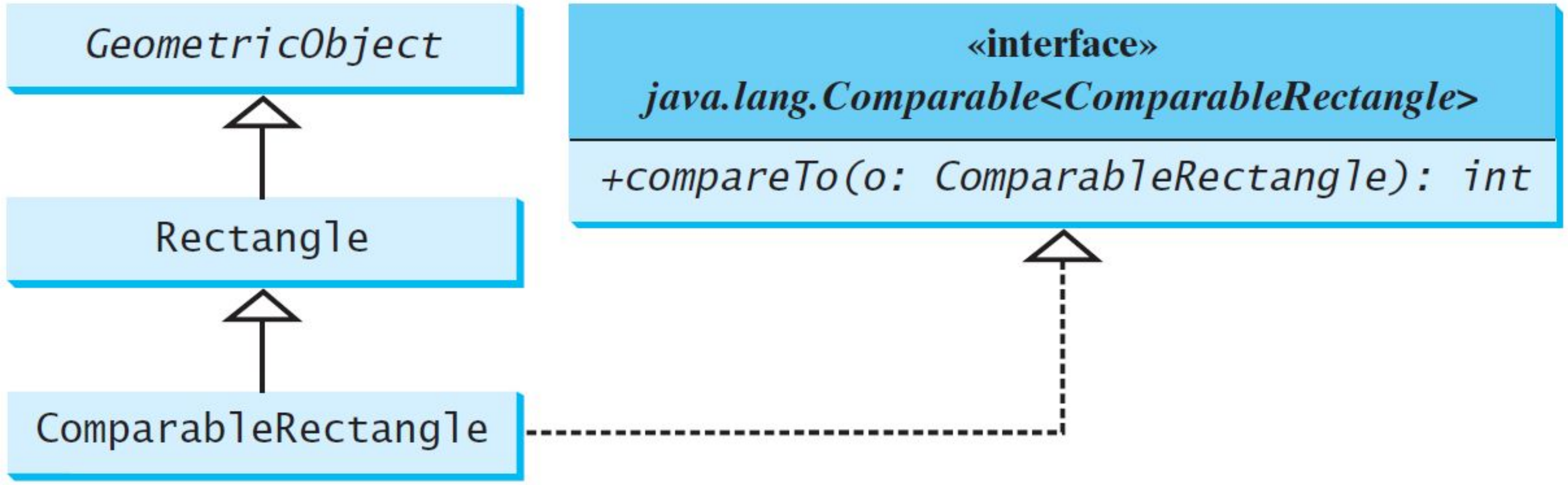
```
s instanceof String
s instanceof Object
s instanceof Comparable
```

```
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

Defining Classes to Implement Comparable


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

ComparableRectangle

SortRectangles

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

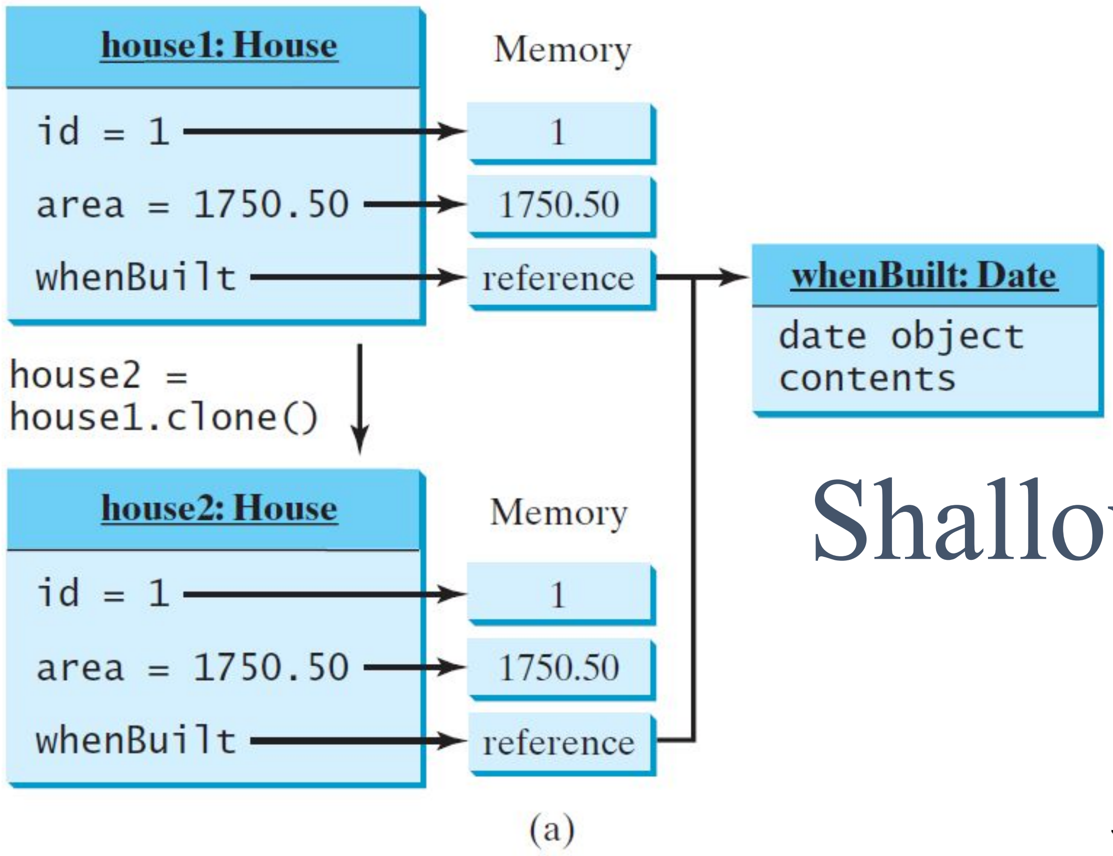
To define a custom class that implements the Cloneable interface, the class must override the clone() method in the Object class.

The following code defines a class named House that implements Cloneable and Comparable.

House

Shallow vs. Deep Copy: Default clone() from Object Class

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

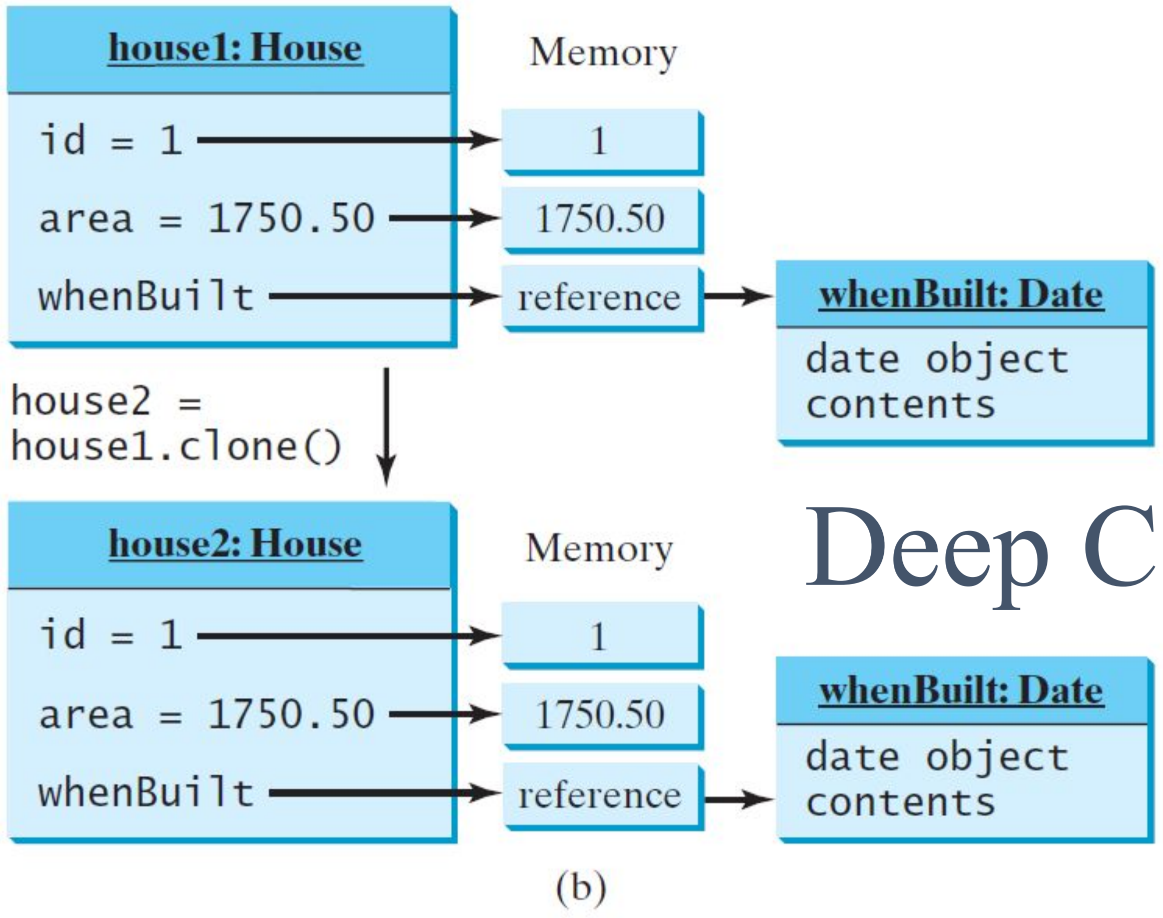


Shallow Copy

Shallow vs. **Deep** Copy: Override clone()

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

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



```
@Override
public Object clone() {
        House houseClone = new House(id, area);
        Date d = houseClone.getWhenBuilt();
        d.setTime(whenBuilt.getTime());
        return houseClone;
}
```

Deep Copy

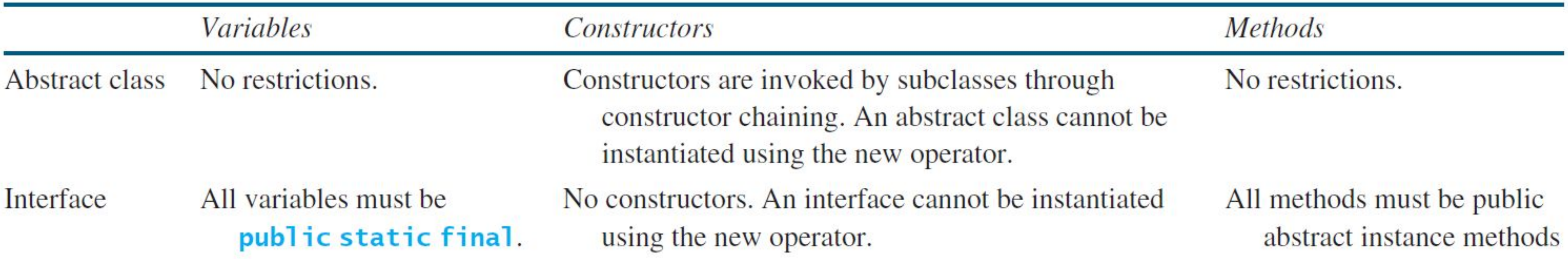
Ayo berhenti dulu!

- ❑ Apa itu interface?
- ❑ Untuk apa menggunakan interface?
- ❑ Bagaimana cara mendefinisikan interface?
- ❑ Bagaimana cara menggunakan interface?

Interfaces vs. Abstract Classes

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

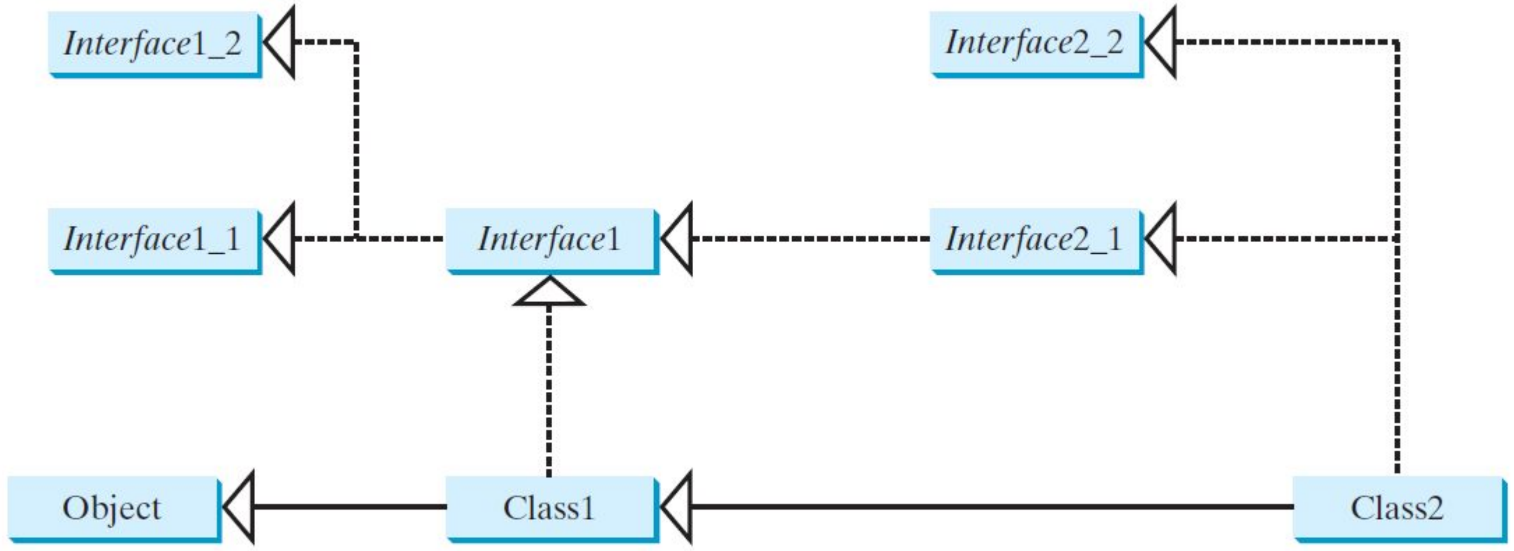
Each method in an interface has only a signature without implementation; an abstract class can have concrete methods.



	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

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

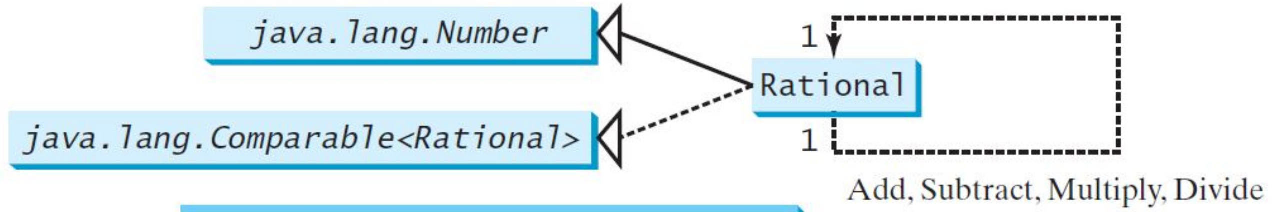
- In rare occasions, a class may implement two interfaces with conflict information
- (e.g., two same constants with different values or two methods with same signature but different return type).
- This type of errors will be detected by the compiler.

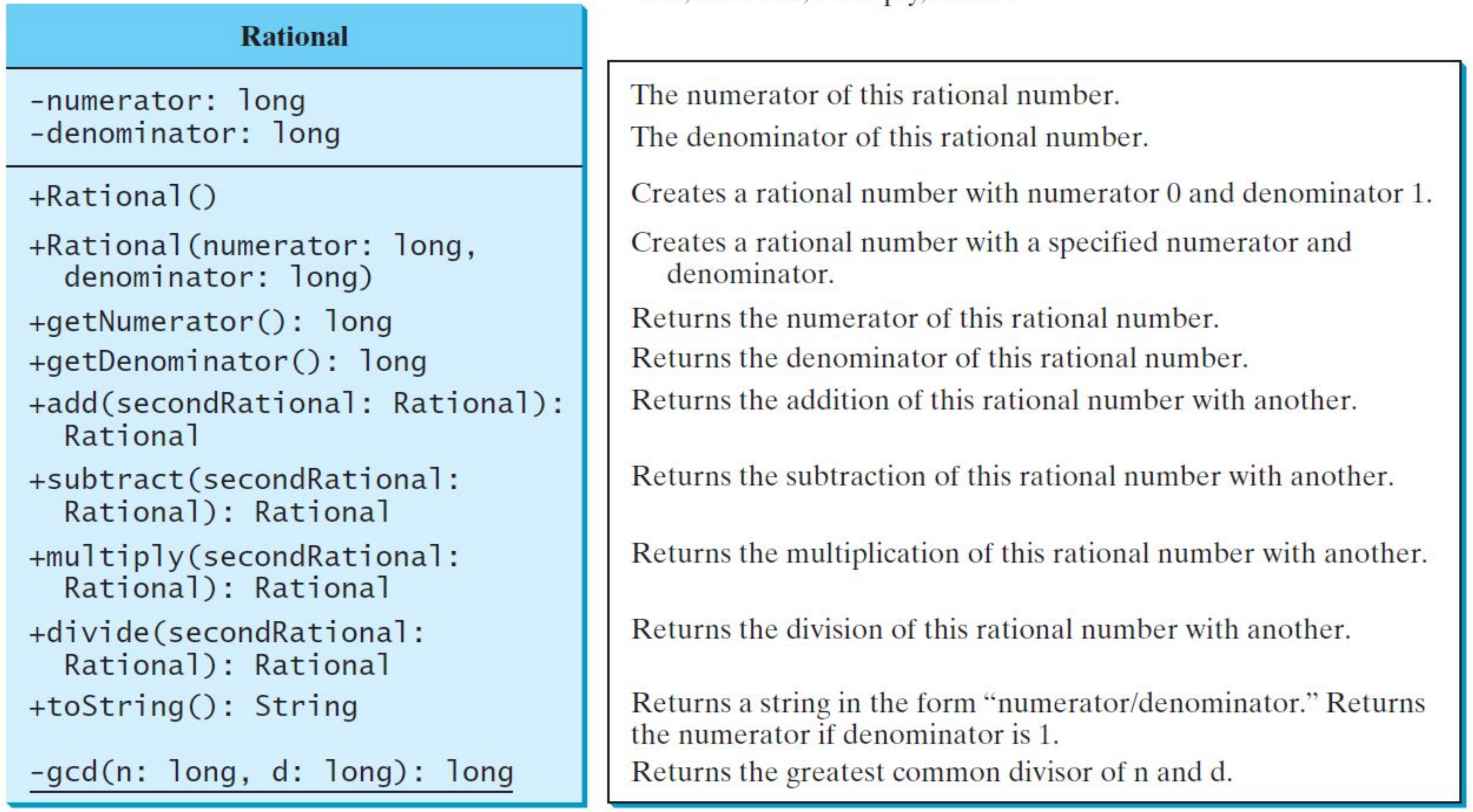
Whether to use an interface or a class?

Abstract classes and interfaces can both be used to model common features. ***How do you decide whether to use an interface or a class***?

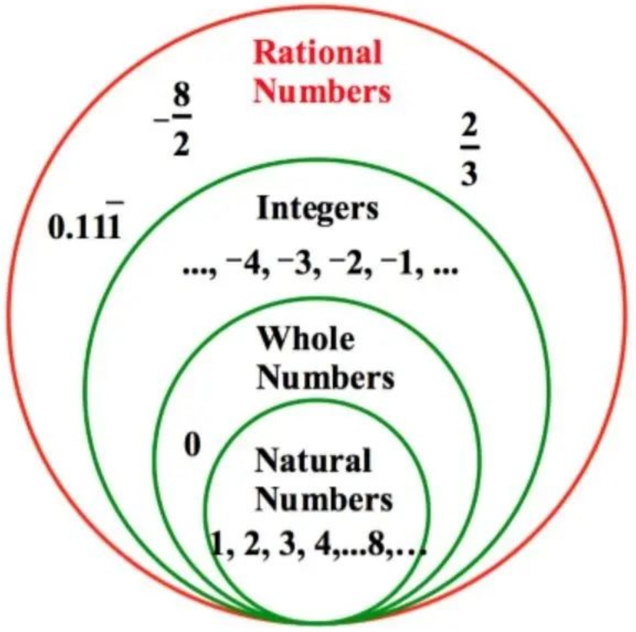
- ❑ In general, a ***strong is-a*** relationship that clearly describes a parent-child relationship should be modeled using classes. For example, a staff member is a person.
- ❑ A ***weak is-a*** *relationship*, also known as ***an is-kind-of*** relationship, indicates that an object possesses a certain property. (*A bird can fly; a person can move and can breathe*)
- ❑ 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.

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

Liang, Introduction to Java Programming, Tenth Edition, Global Edition. © Pearson Education Limited 2015

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS
ILMU
KOMPUTER

Ayo berhenti dulu!

- ❑ Apa saja perbedaan abstract class dan interface?
- ❑ Kapan sebaiknya menggunakan abstract classes?
- ❑ Kapan sebaiknya menggunakan interface?

Designing Classes

Object Oriented Principles – Design Guidelines

Designing a Class

(Strong) Coherence and (Loose) Coupling

- 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 to support ***loosely coupled***.

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.

Liang, Introduction to Java Programming, Tenth Edition, Global Edition. © Pearson Education Limited 2015

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.
- For example:
  The *gcd* method in the Rational class is private, because it is only for internal use within the class.

Rational

FACULTY OF COMPUTER SCIENCE

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

Using the *static modifier* Or Apply *Singleton Pattern*

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

Or better:

Apply Singleton Pattern!
(will be studied later in Software engineering courses)

Next !

We will study a famous Design Principle in OOP : SOLID!

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia
FAKULTAS ILMU KOMPUTER

Ayo berhenti dulu!

- ❑ Kapan menggabungkan *class*, kapan memisahkan?
- ❑ Apa hubungannya *class* dengan *reuse*?
- ❑ Haruskah membuat *constructor* tanpa parameter?
- ❑ Perlukan meng-*override*, toString?
- ❑ Bagaimana mengatur *visibility* dari *modifier*?

Group Discussion – (Kuis)

- Make a group of 3 students, discuss the following issues:

A Startup company would like to create an IT product of delivering fresh vegetables. Since it is fresh vegetable, it should be delivered right a way and in a short distance. It provides direct access from farmer to the users. Other details are not available yet. Use your creativity!

Your task:

❑ Define what are the classes, which one is abstract class, which one is the interface.

❑ Apply the previously mentioned design principles.

Submission:

❑ Draw the class diagram (min 3 class, min 1 abstract class, min 1 interface)

❑ Write short description how your class diagram design follow the design principles.

❑ (Next possibly on exam) Implement it in Java.

Bottom-Up Approach for TP

1) Atur packaging (Bila perlu), check apakah sudah bisa compile atau belum, lakukan *commit-push* bisa sudah berhasil, berikan pesan yang sesuai.
2) Buat *class skeleton* berdasarkan *class diagram,* buat *method stub* nya dengan dummy implementation (*return default value*, atau *do nothing*) , lakukan *commit-push* dengan pesan yang sesuai.
3) Per *class*:
   i. Buat unit test untuk sebuah method sesuai ekspektasi soal, *commit-push*!
   ii. Implementasi method tersebut, check unit test-nya sampai benar, lakukan *commit-push* dengan pesan yang sesuai.
   iii. Ulangi untuk seluruh method dalam class
   iv. Ulangi untuk *class* yang lain
4) Buat/Lengkapi *class* utama atau *simulator* menu-nya. Lakukan *commit-push*, dengan pesan “Yey,.. Akhir nya selesai ☺️ ”.

UNIVERSITAS INDONESIA
Veritas, Probitas, Iustitia

FAKULTAS ILMU KOMPUTER

Selamat Berlatih!

Perhatikan lagi List Objective yang perlu dikuasai pekan ini.
Baca buku acuan dan berlatih!
Bila masih belum yakin tanyakan ke dosen, tutor atau Kak Burhan.

Semangat !