Abstract Classes and Interfaces

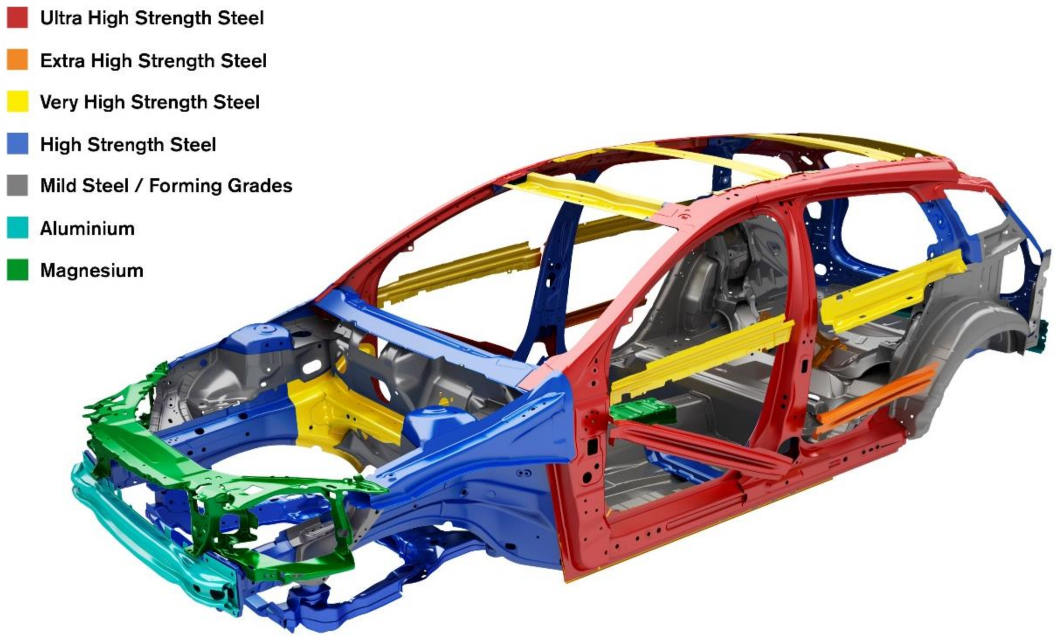
- Contracts between a library implementation and its clients

Shing-Chi Cheung

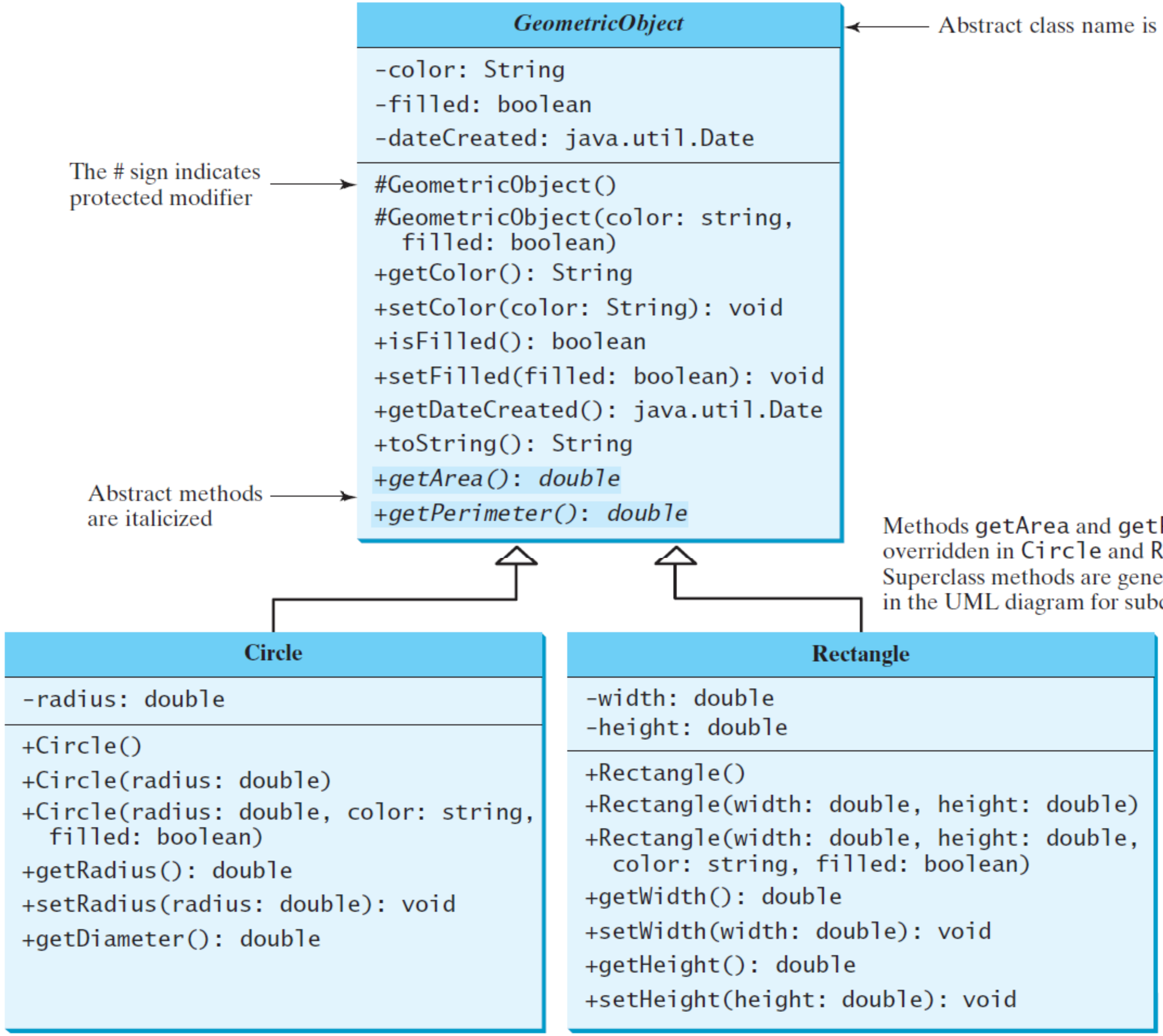
Computer Science and Engineering

HKUST

Part I – Abstract Classes and Methods


🟥 Ultra High Strength Steel
🟧 Extra High Strength Steel
🟨 Very High Strength Steel
🟦 High Strength Steel
⬛ Mild Steel / Forming Grades
🟦 Aluminium
🟩 Magnesium

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.java

TestGeometricObject.java

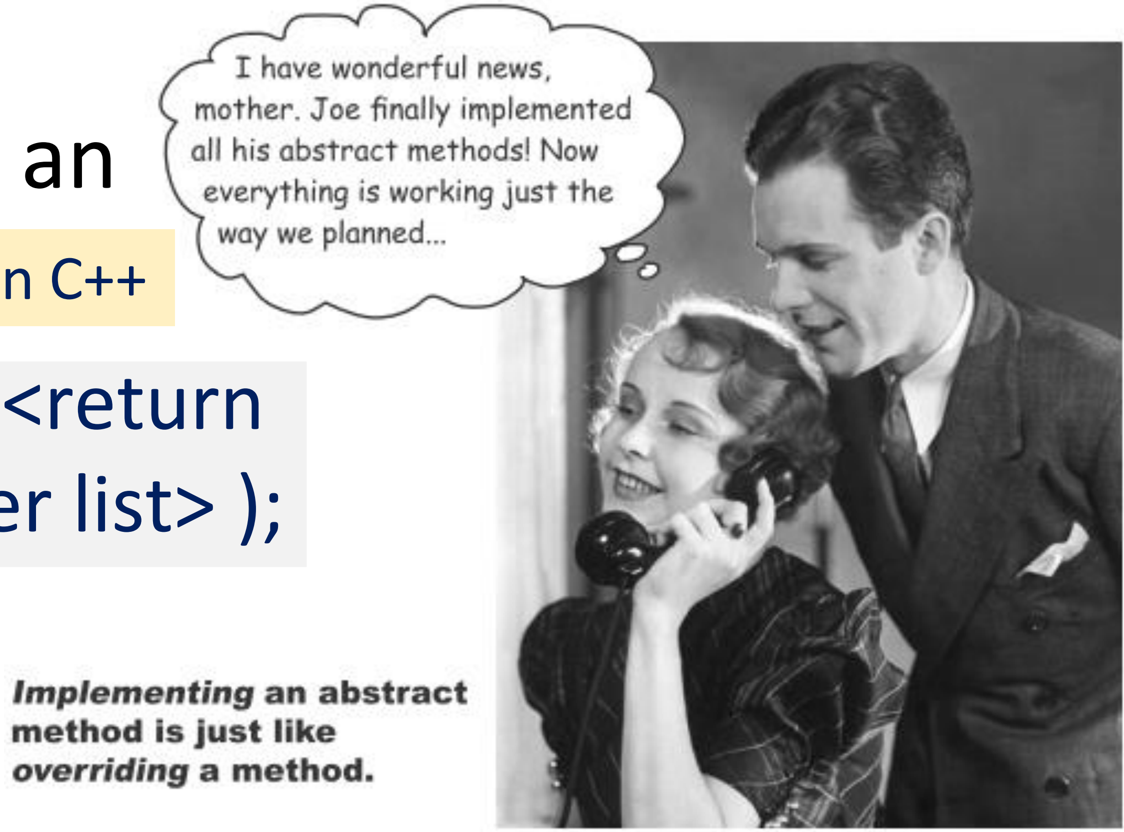
Abstract Methods

- An abstract method always has an empty body

c.f. pure virtual function in C++

Syntax: <access modifier> abstract <return type> <method name> (<parameter list>);

- An abstract method must be overridden before use
- A non-abstract method is considered to be concrete


I have wonderful news, mother. Joe finally implemented all his abstract methods! Now everything is working just the way we planned...

***Implementing** an abstract method is just like **overriding** a method.*

Why Need Abstract Methods?

- Suppose we want to sum up the areas of geometric objects in a list

```
public static double getTotalArea(@NotNull GeometricObject [] geoArray) {
  var area = 0.0;
  for (var o: geoArray) area += o.getArea();
  return area;
}
```

- Different objects (e.g., circles and rectangles) have their own area formula
- In the GeometricObject class, we declare getArea to be an abstract method

```
public abstract double getArea(); // abstract method
```

- Enforce each GeometricObject’s subclass to implement its own getArea()

TestGeometricObject.java

Abstract Classes

- A class is abstract if it is declared to be abstract

Syntax: <access modifier> abstract class <class name> { … }

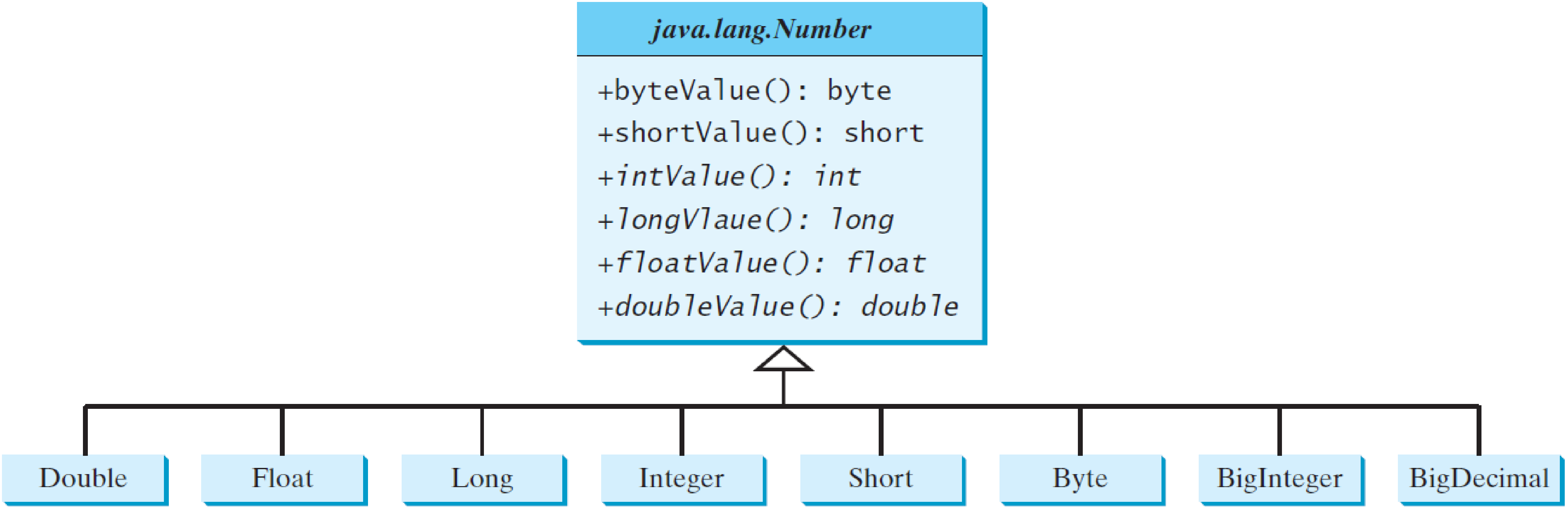
- An abstract class can contain abstract and concrete methods
- A concrete (i.e., non-abstract) class cannot contain any abstract methods

Rules of Using Abstract Classes

Can an abstract class A:

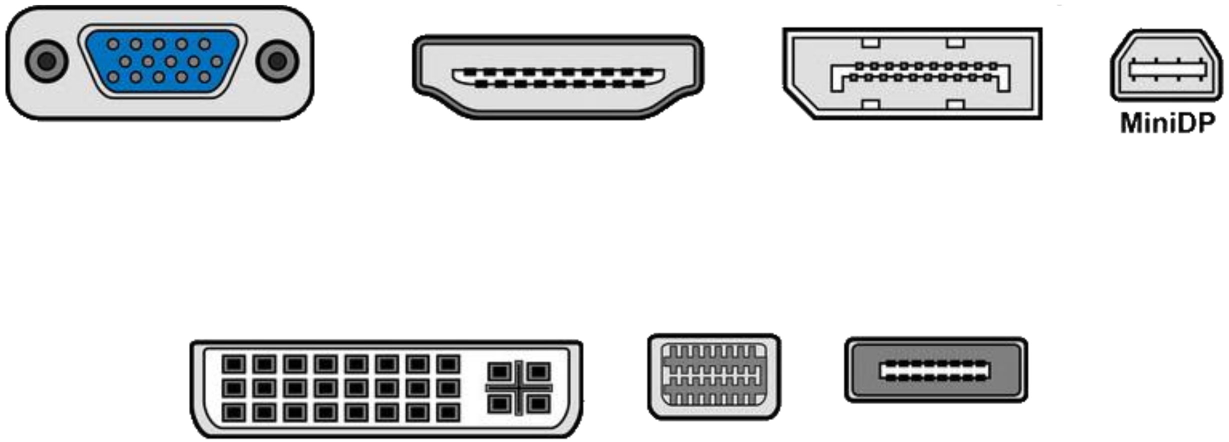
- contain private abstract methods
- be used to instantiate objects using the new operator
- be used to create an instance of itself
- have constructors
- have no abstract methods
- have no methods
- have a concrete superclass
- have an abstract method overriding a concrete method
- be used as a type to declare variables

Number - A Useful Built-in Abstract Class



The built-in Number class allows us to handle different types of numeric values uniformly

Part II – Interfaces



What is an interface? Why is it useful?

- Interface is an important language construct introduced by Java to manage code complexity
  - Adopted by essentially all modern OO programming languages
- An interface is a class-like construct that contains constants and abstract methods *(prior to Java 8)*
  - Java 8 adds support of default methods
  - Java 9 adds support of private interface methods

What is an interface? Why is it useful?

- In many ways, an interface is similar to an abstract class, but the intent of an interface is to specify common features or capabilities for objects. Examples of built-in interfaces are Comparable, Serializable and Cloneable
- Methods defined in an interface must be abstract unless they are explicitly declared to be default or private
- Methods defined in an interface can only be overridden by public methods

Default Methods since Java 8

- Java 8 introduces a new feature called default methods, which can add method bodies into interfaces
- Why default methods?
  - We can add new capabilities (as public non-abstract methods) to an interface without breaking the implementation of the classes that have implemented the interface
  - An example is the addition of data streaming capability by introducing the forEach() default method
- We will visit default methods later

Private Interface Methods since Java 9

- Java 9 introduces a new feature called private interface methods, which can add private non-abstract methods into interfaces
- Why private interface methods?
  - They are helper methods local to an interface, aiming to ease the implementation of the multiple default methods in the interface
- We will visit private interface methods in the lab
- **In the remaining slides, we will study the interface concepts that are valid up to Java 7 (i.e., do not consider default and private interface methods).**

Interface Declaration (as Implementation Specification)

To distinguish an interface from a class, Java uses the following syntax to declare an interface:

Syntax:

```
public interface <interface name> {
  <constant declarations>
  <abstract method signatures>
}
```

Example:

```
public interface Taxable {
  public final static double TAX_RELIEF = 40000;
  public abstract double getEarnings();
}
```

Interface is a Special Abstract Class

- An interface is treated like a special Java abstract class that has only abstract methods and constants.
- We can use an interface in a way similar to an abstract class. For example:
  - We cannot use an interface to create an instance of itself.
  - We can use an interface as a data type for a variable, as the result of casting, and so on.

This statement is applicable only up to Java 7. Still, it is an important language design decision that has been adopted by many modern programming languages.

Rules of Using Interfaces

Can an interface

- contain instance variables?
- contain mutable static variables?
- contain non-public immutable static variables?
- have constructors?
- have static initializer blocks?
- contain abstract static methods?
- contain non-abstract public instance methods?

Interface Example

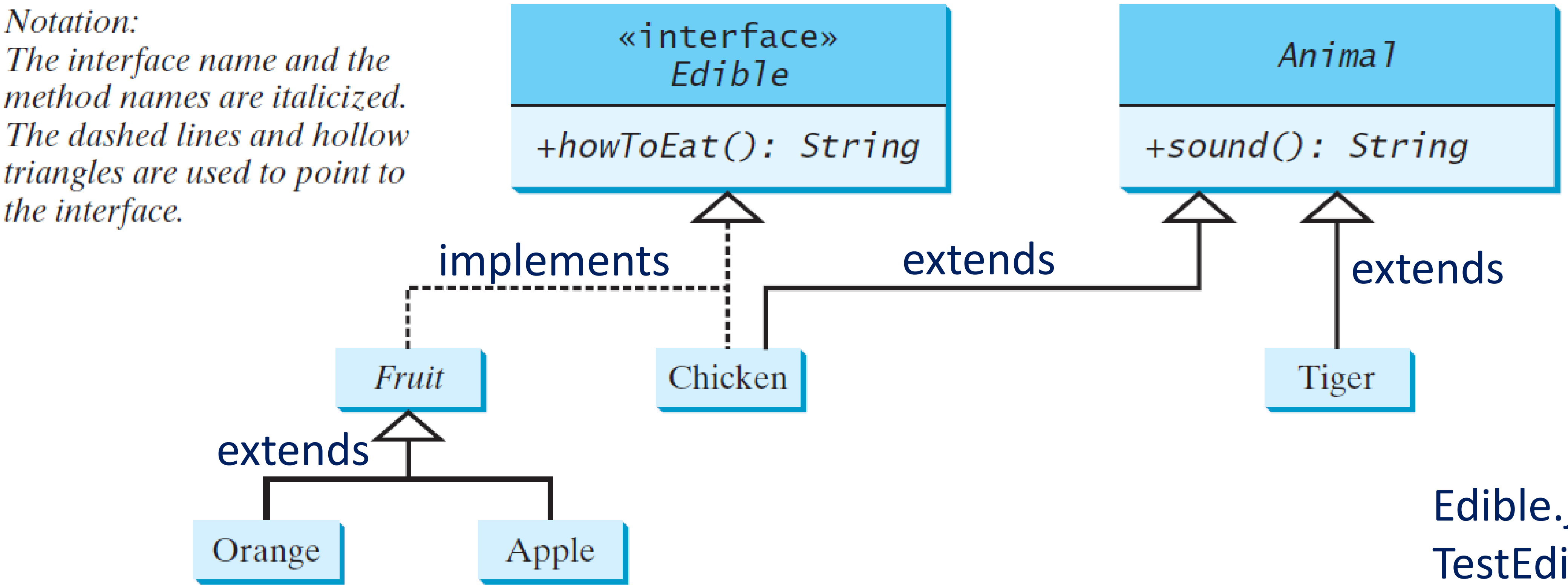
- Edible is an interface to specify if an object is edible
- A class can extend one superclass and at the same time implement multiple interfaces

Examples:

- Chicken is a class that extends an abstract class Animal and implements Edible
- Apple is a class that extends an abstract class Fruit, which implements Edible

Implementing interfaces

A class implements an interface by implementing the interface's abstract instance methods



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

Edible.java
TestEdible.java

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 to

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

  void p();
}
```

A constant defined in an interface can be accessed using syntax <interface name>.<CONSTANT NAME> (e.g., T1.K).

IfcRules.java

Implementing Multiple Interfaces

- Java allows a class to implement multiple interfaces

Syntax: <access modifier> class <class name> extends <superclass name>
implements <interface 1>, …, <interface n> { … }

- Java allows an interface to extend multiple interfaces

Syntax: <access modifier> interface <interface name>
extends <interface 1>, …, <interface n> { … }

- This let us enjoy most of the benefits of multiple class inheritance in C++ without worrying the ambiguity problem thus induced. Why?

TestInterface.java

Built-in Interface: Comparable<E>

- The Comparable is a built-in interface in the java.lang package.

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

E is a generic type variable. Assign E to the type of objects to be compared; otherwise any type of objects can be compared.

- All built-in wrapper classes (e.g., Boolean) override the toString(), equals(Object o), and hashCode() methods defined in Object.
- All built-in numeric wrapper classes (e.g., Integer) and the Character class implement the Comparable interface, and implement the compareTo method.

Integer and BigInteger Classes

- Many Java library classes implement Comparable in order to define a natural order for their instances. Examples of typical implementation:

```
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
  }
}
```

Usage: `System.out.println(Integer.valueOf(3).compareTo(Integer.valueOf(5)));`

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 Usage:

```
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>.

Arrays.sort() is guaranteed to be stable.

SortComparableObjects.java

Generic Method

```
public static Comparable max1
  (Comparable o1, Comparable o2) {
  if (o1.compareTo(o2) > 0)
    return o1;
  else
    return o2;
}                              (a)
```

```
public static Object max2
  (Object o1, Object o2) {
  if (((Comparable)o1).compareTo(o2) > 0)
    return o1;
  else
    return o2;
}                                  (b)
```

```
var s1 = "abcdef";
var s2 = "abcdee";
var s3 = (String) Max.max(s1, s2);
```

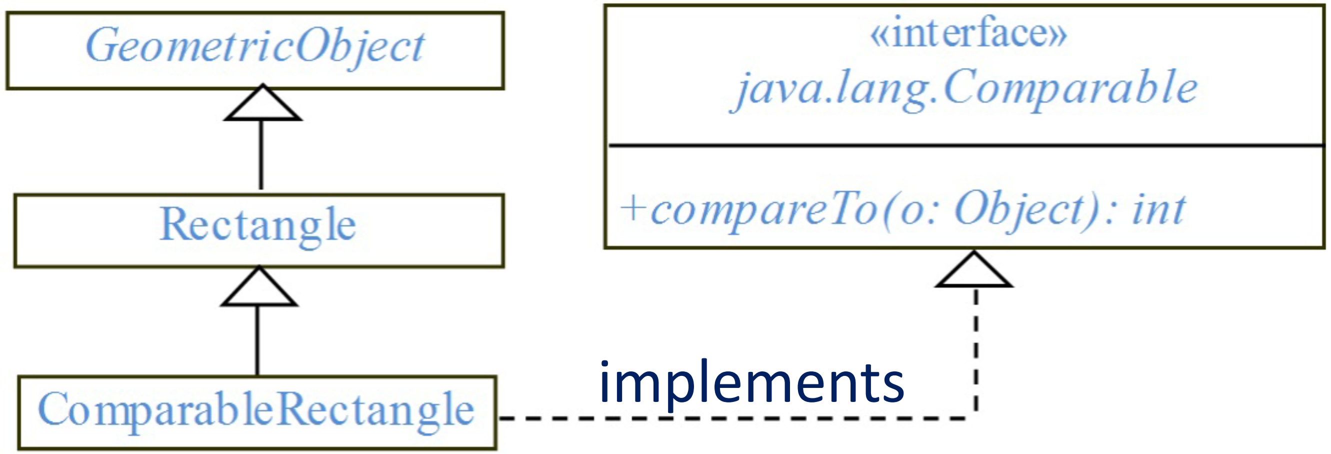
```
var d1 = new Date(2013, 1, 1);
var d2 = new Date(2012, 1, 1);
var d3 = (Date) Max.max(d1, d2);
```

The return value from the max method is of the Comparable type. So, you need to cast it to String or Date explicitly.

GenericMaxTest.java

Defining Classes to Implement Comparable

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



- The Rectangle class does not implement Comparable
- To leverage the generic max method to compare two rectangles, we define a new rectangle class, ComparableRectangle, that implements Comparable

```
ComparableRectangle rectangle1 = new ComparableRectangle(4, 5);
ComparableRectangle rectangle2 = new ComparableRectangle(3, 6);
System.out.println(Max.max(rectangle1, rectangle2));
```

ComparableRectangle.java
ComparableRectangleWithGenerics.java

The Cloneable Interface

empty body

public interface Cloneable { }

- Cloneable is defined in the java.lang package as a marker interface whose body is empty.
- A marker interface has an empty body. It denotes a specific capability
- An instance of the class that implements Cloneable can be cloned

  Example: class CloneableCircle extends Circle implements Cloneable { … }
- clone() is a protected instance method defined in the Object class for cloning.
- *Note: Java uses the clone() method instead of copy constructors*

```
java.lang.Object
      protected Object clone()
      throws CloneNotSupportedException
```

The Cloneable Interface

- Many built-in Java classes (e.g., Date and Calendar) have implemented Cloneable. Thus, instances of these classes can be cloned

For example, the following code:

```
var calendar = new GregorianCalendar(2003, 2, 1);
var calendarCopy = (Calendar) calendar.clone();
System.out.println("calendar == calendarCopy is " + (calendar == calendarCopy));
System.out.println("calendar.equals(calendarCopy) is " + calendar.equals(calendarCopy));
```

will output:

```
calendar == calendarCopy is false
calendar.equals(calendarCopy) is true
```

CalendarClone.java

An Example of Defining a Cloneable Class

In the following, we define a cloneable House

```java
public class House implements
    Cloneable, Comparable<House> {
  private int id;
  private double area;
  private java.util.Date whenBuilt;

  public House(int id, double area) {
    this.id = id;
    this.area = area;
    whenBuilt = new java.util.Date();
  }
```

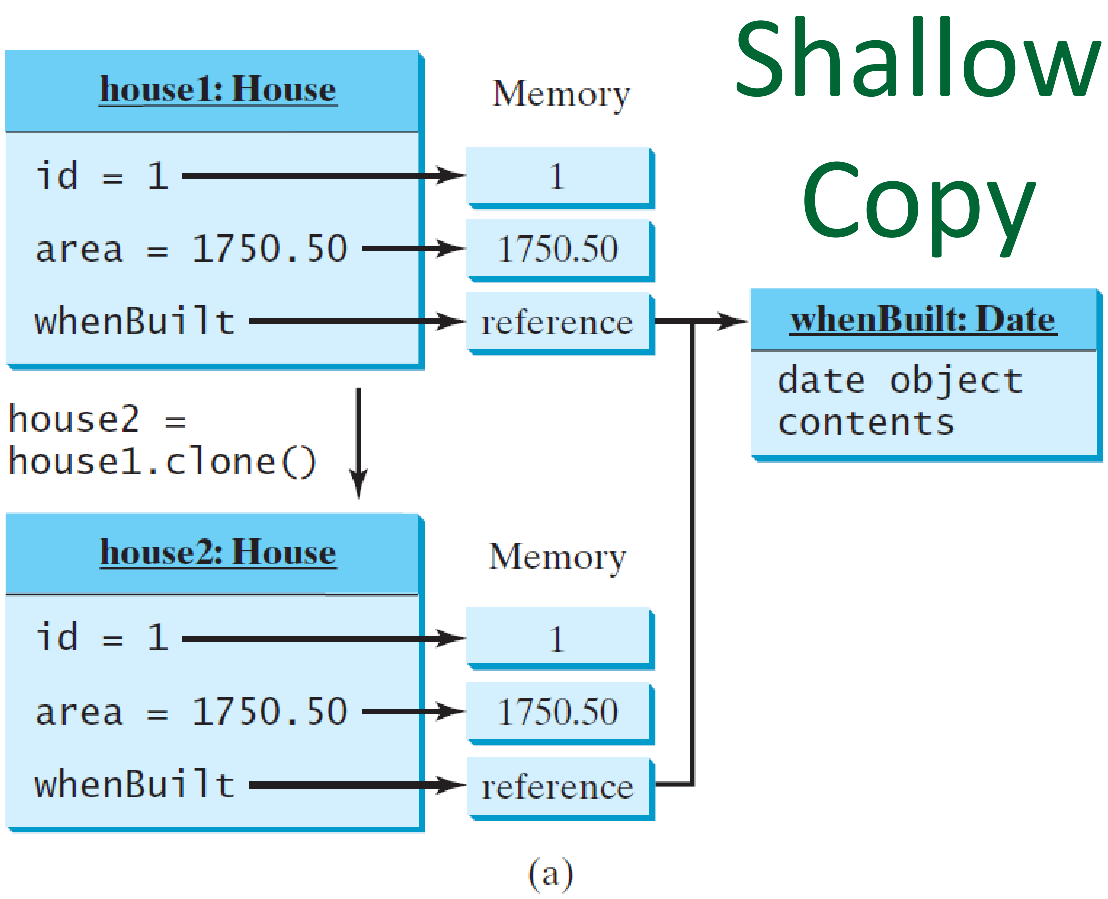
```java
@Override // Object.clone()
public House clone() {
  try {
    return (House) super.clone();
  } catch (CloneNotSupportedException ex) {
    return null;
  }
}
@Override
public int compareTo(@NotNull House h) {
  return Double.valueOf(area).compareTo(h.area);
}
```

House.java

Shallow vs. Deep Copy

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

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



Shallow Copy

house1: House

id = 1

area = 1750.50

whenBuilt

Memory

1

1750.50

reference

whenBuilt: Date

date object contents

house2 = house1.clone()

house2: House

id = 1

area = 1750.50

whenBuilt

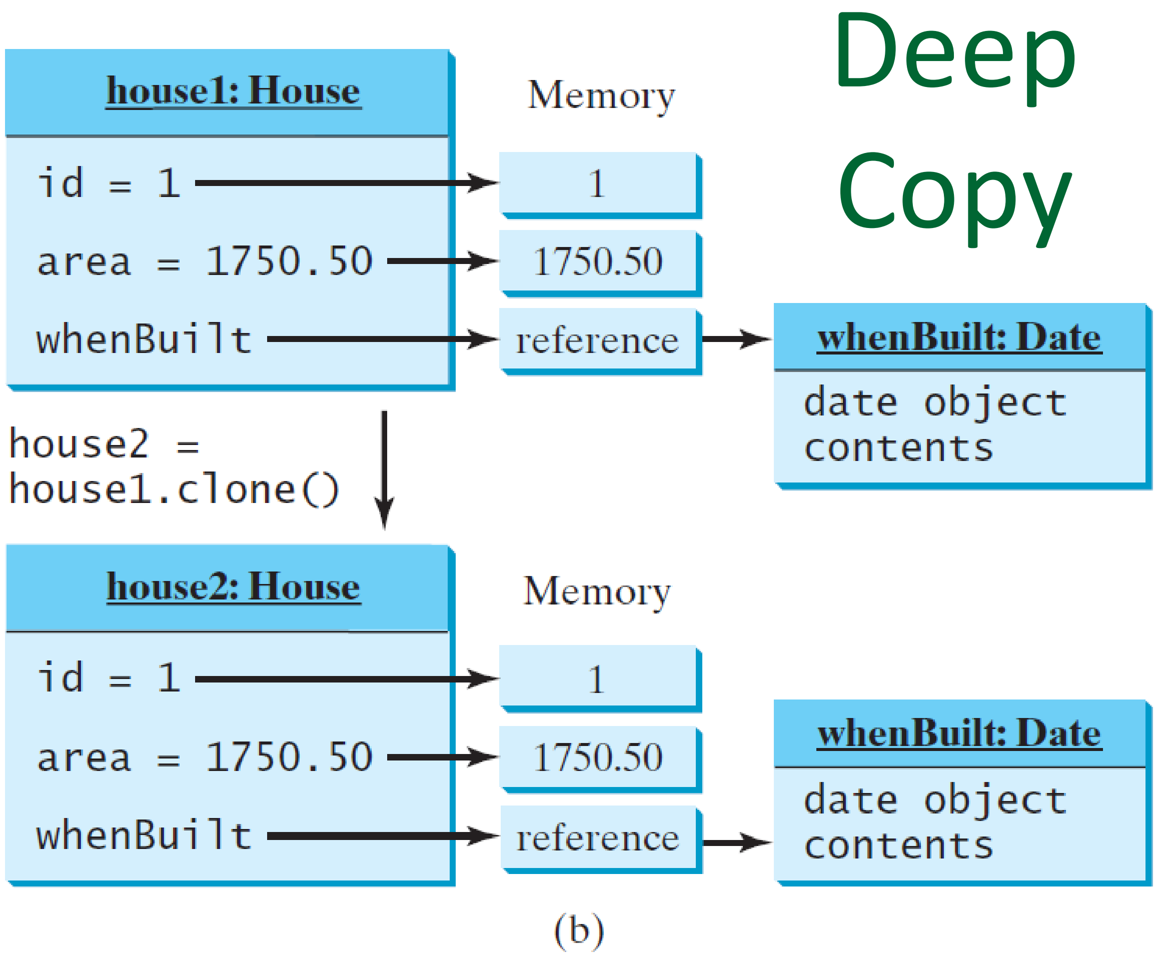
Memory

1

1750.50

reference

(a)



Deep Copy

house1: House

id = 1

area = 1750.50

whenBuilt

Memory

1

1750.50

reference

whenBuilt: Date

date object contents

house2 = house1.clone()

house2: House

id = 1

area = 1750.50

whenBuilt

Memory

1

1750.50

reference

whenBuilt: Date

date object contents

(b)

How to implement deep copy in the clone() of House?

```
@Override
public House clone() {
  try {
    var h = (House) super.clone();
    h.whenBuilt = (java.util.Date) this.whenBuilt.clone();
    return h;
  } catch (CloneNotSupportedException ex) {
    return null;
  }
```

House.java

Access Rules on Protected Fields/Methods

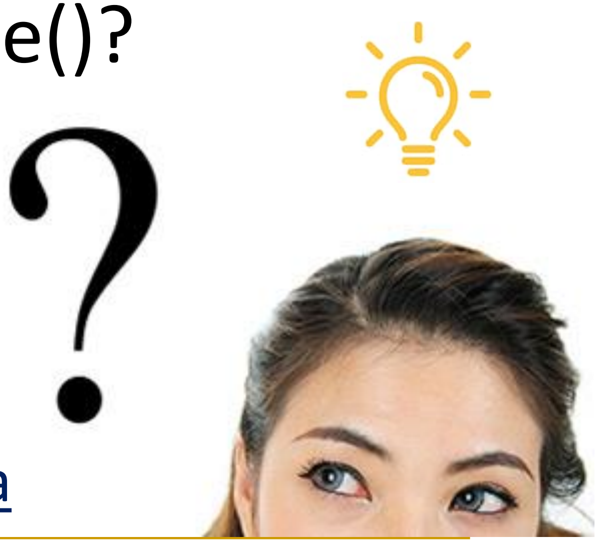
```
java.lang.Object
      protected Object clone()
      throws CloneNotSupportedException
```

	Same package as the protected fields / methods	Different package from the protected fields / methods
Inherited protected fields / methods	✓	✓
Non-inherited protected fields / methods	✓	✘

CloneTest.java

Implementing Cloneable Interface

- A cloneable class must implement the Cloneable interface and must override the protected clone() method in the Object class.
  - What if the class does not implement Cloneable?
  - What if the class does not override Object’s clone()?
  - What are their purposes?
  - What does Object’s clone() do?



CloneTest.java

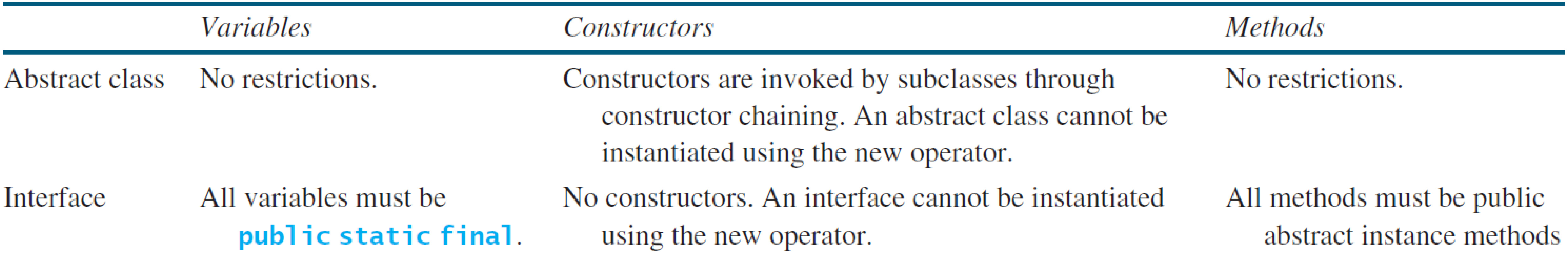
Summary - Rules of Using Interfaces

- All **interface variables** are public, static and final
- All **interface methods** are public and abstract. They cannot be declared static or final. Why?
- A **class** can extend one superclass and implement multiple interfaces
- An **interface** can extend multiple interfaces but it cannot extend any classes
- An **interface** can be used as a type

Interfaces vs. Abstract Classes

Data fields of an interface must be constants; an abstract class can have all types of data fields.

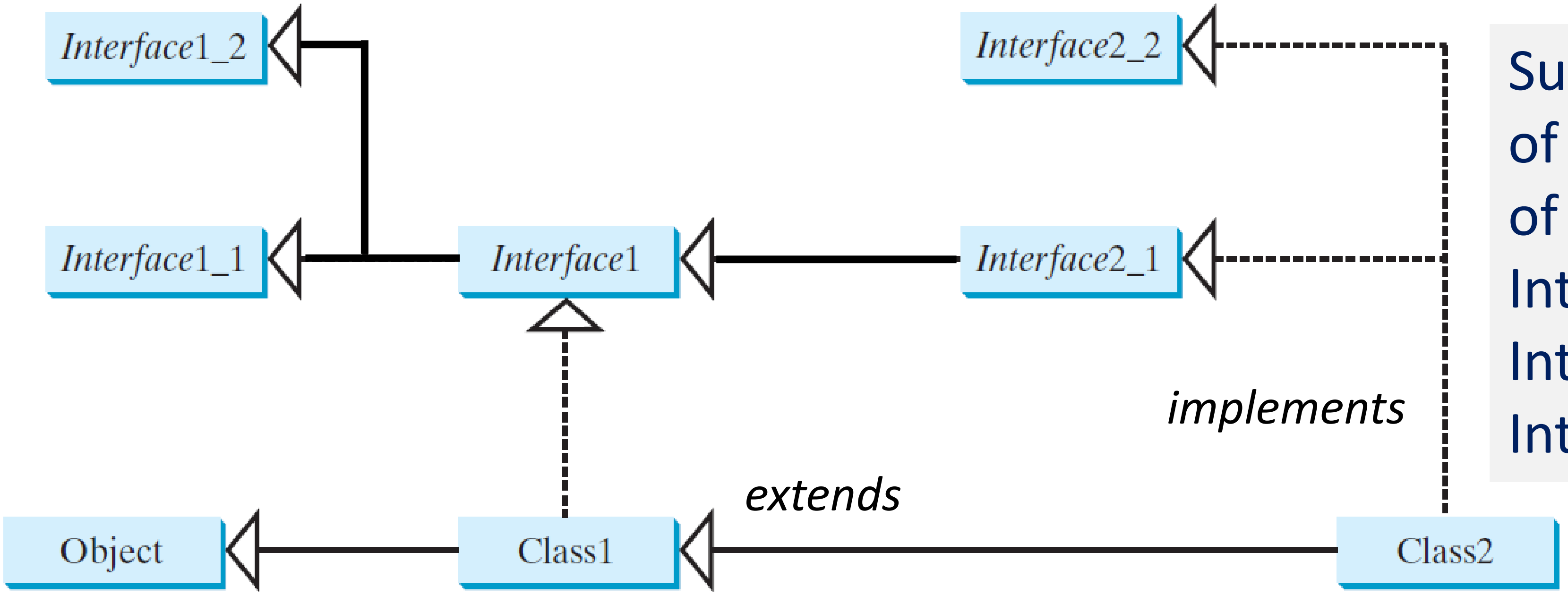
Methods of an interface must be abstract; an abstract class can have non-abstract methods.



	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 (unless they are explicitly declared to be private or default)

Interfaces vs. Abstract Classes, cont.

- Unlike classes, interfaces do not share a single supertype
- An interface is used like an abstract superclass
  - We can use an interface as a type
  - We can cast a variable of an interface type to its implementation class, and vice versa
  - A variable of an interface type can reference any instance of its implementation class



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.

Summary: Interfaces vs Abstract Classes?

- Both abstract classes and interfaces can model common features. How do we decide whether to use an interface or an abstract class?
- 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 specific feature/capability. A weak is-a relationship can be modeled using interfaces. For example, all strings are comparable, so the String class implements the Comparable interface
- We can use interfaces to support multiple supertypes, circumventing the single class inheritance restriction
  - ❑ A class can implement multiple interfaces
  - ❑ An interface can extend multiple other interfaces

More Practice?

- Google code jam

code jam

System.out.println("hello, world!");

https://code.google.com/codejam/contests.html