Abstract Class & Interface

Dasar – Dasar Pemrograman 2

Slide Acknowledgment: Tim Pengajar DDP 2

Pudy Prima

References

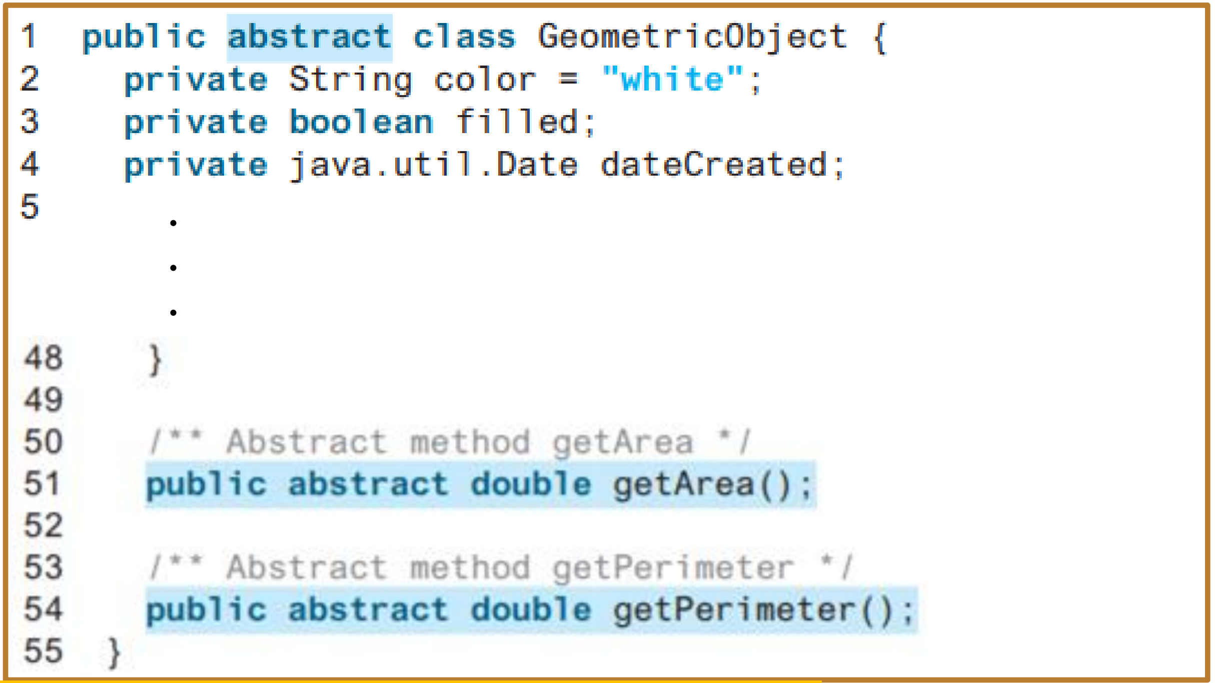
- Liang, Introduction to Java Programming, 11th Edition, Ch. 13
- Code examples are taken/modified from:
  - https://www3.ntu.edu.sg/home/ehchua/programming/java/J3b_OOPInheritancePolymorphism.html
  - https://guide.freecodecamp.org/java/interfaces

Abstract Classes

Motivation

- A superclass defines common behavior for related subclasses. An interface can be used to define common behavior for classes (including unrelated classes).
- In the inheritance hierarchy, classes become more specific and concrete *with each new subclass*. Class design should ensure a superclass contains common features of its subclasses. Sometimes, a superclass is so abstract it **cannot be used to create any specific instances**. Such a class is referred to as an *abstract class*.

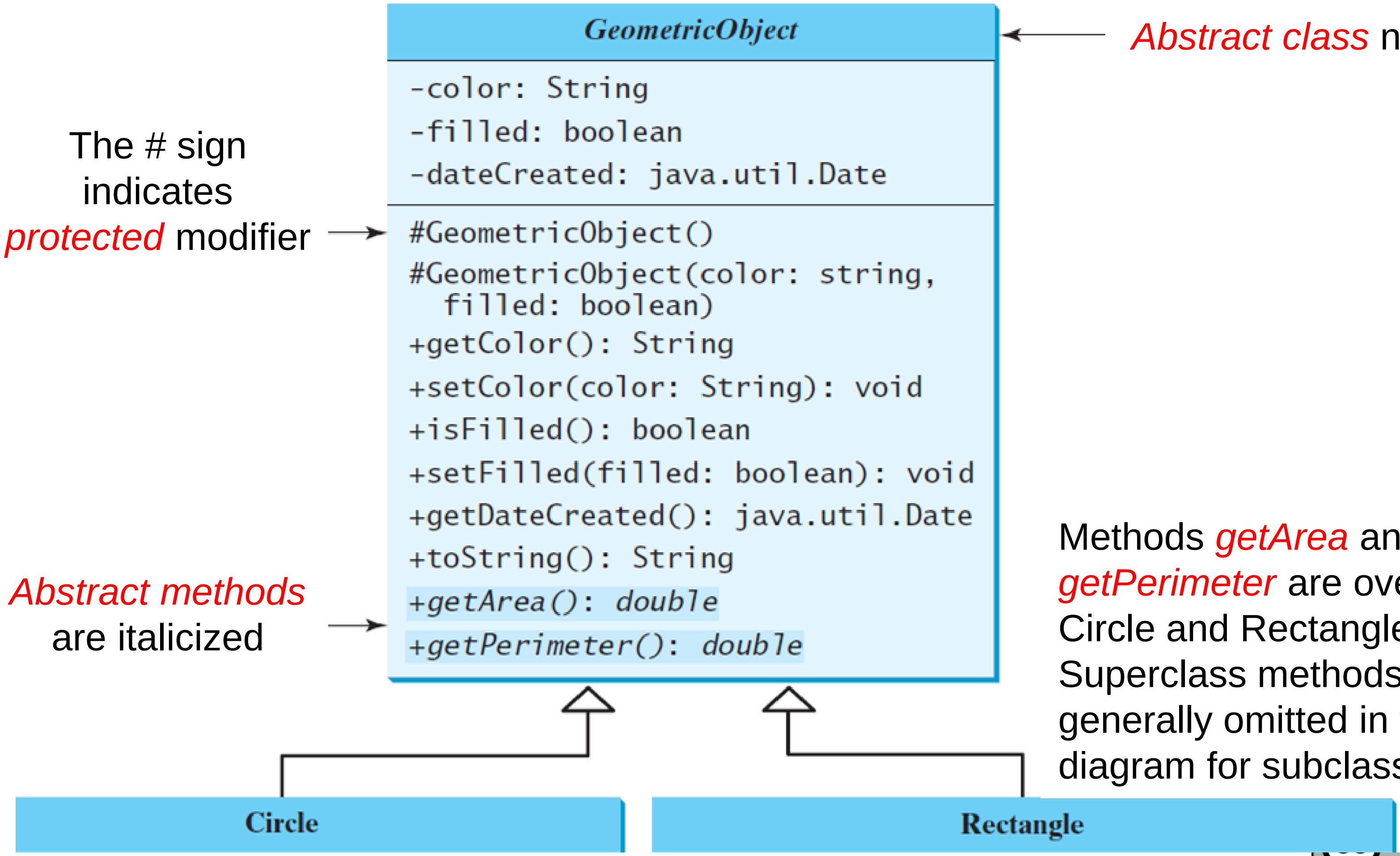
Abstract class → class yang tidak bisa diinstansiasi
Abstract class biasanya dibangun sebagai base class dari class lain.



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

      .
      .
      .
    }

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

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

```java
public class Circle extends GeometricObject{
  // ...
}
```

```java
public class Rectangle extends GeometricObject{
  // ...
}
```


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
Circle
Rectangle

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.

Abstract Method

Abstract method implementation depends on the specific type of object.

```
public abstract class Shape {
  // ...
  public abstract double getArea();
}
```

See..?! No implementation

Abstract method → method yang belum memiliki implementasi (tidak ada method body)
Class yang mengandung abstract method harus dideklarasikan sebagai abstract class.
Implementasi abstract method dapat dibuat di class turunannya (override).

Abstract class: rule #1

- An abstract method cannot be contained in a nonabstract (or concrete) class.
- An abstract class may have abstract methods (or non-abstract methods). It is possible to define an abstract class that contains no abstract methods.
- A class extending an abstract class must implement all the abstract methods, except the class is also abstract.

Abstract class: rule #2

- 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.

```
abstract class Shape {
  private String color;

  Shape (String color) {
    this.color = color;
  }
}
```

✖

```
class TestShape {
  Shape s = new Shape("green");
}
```

Abstract class: rule #3

- 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.

Abstract class: rule #4

- 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.

```
class Circle {
  private double radius;
  Circle(double radius) {
    this.radius = radius;
  }
  double getArea(){
    return Math.PI * radius * radius;
  }
}
```

```
abstract class Round extends Circle{
  double r1;
  double r2;
  Round(double r1, double r2){
    super(r1);
    this.r1 = r1;
    this.r2 = r2;
  }
  abstract double getArea();
}
```

Abstract class: rule #5

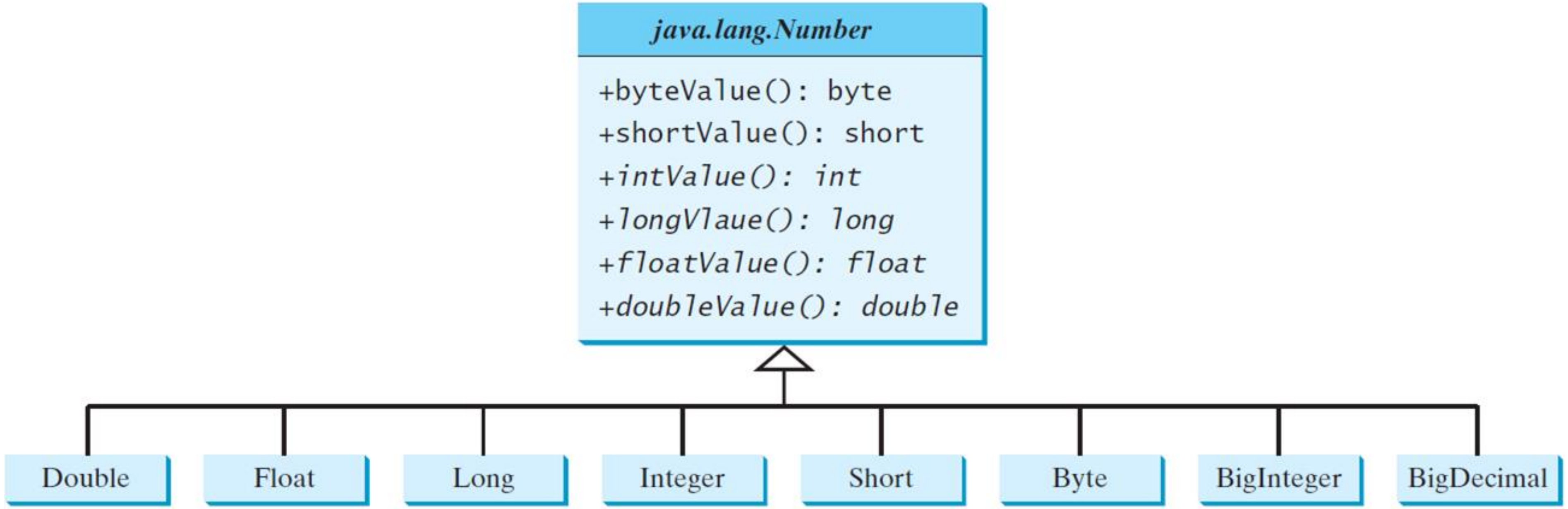
- You cannot create an instance from an abstract class using the new operator, but an abstract class can be used as a data type.

```
abstract class Shape {
  String color;
  Shape (String color) {
    this.color = color;
  }
}
```

```
class Circle extends Shape {
  private double radius;
  Circle(String color, double radius){
    super(color);
    this.radius = radius;
  }
  public static void main(String[] ar){
    Circle c = new Circle(3);
    System.out.println(c.color);
  }
}
```

```
class TestShape {
  Shape[] s = new Shape[3];
  s[0] = new Circle("red",2.3);
}
```

Abstract class: Number



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.
- Interfaces are like features, for example, you can add features to your smartphone, like GPS-featured and radio-featured.
- For example, you can specify that the objects are comparable, edible, cloneable using appropriate interfaces; Rentable could be an interface for Car, and Book; and Edible could be an interface for Mushroom, and Chicken

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.
- **Unlike an abstract class**, interface cannot have constructors, and you (literally) “`implements`” it, not “`extends`” it.

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:

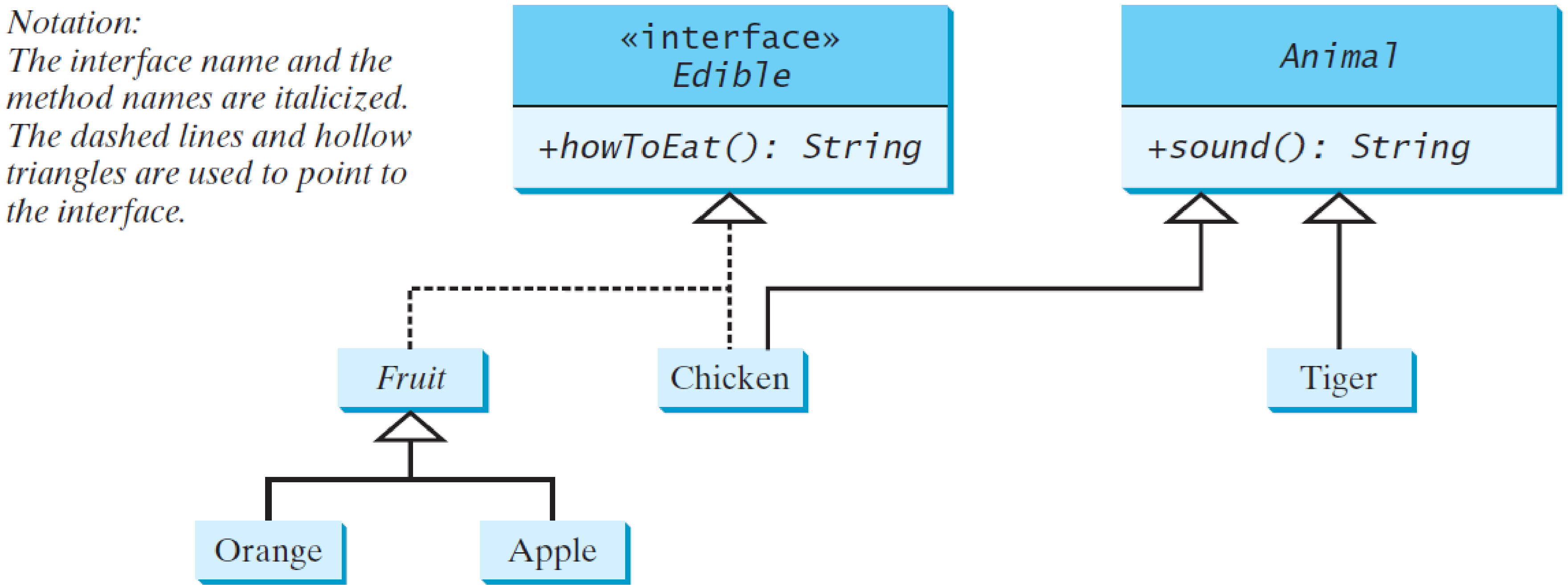
```
public interface Edible {
  /** Describe how to eat */
  public abstract String howToEat();
}
```

Example

For example, the classes Chicken and Fruit implement the Edible interface (See TestEdible).

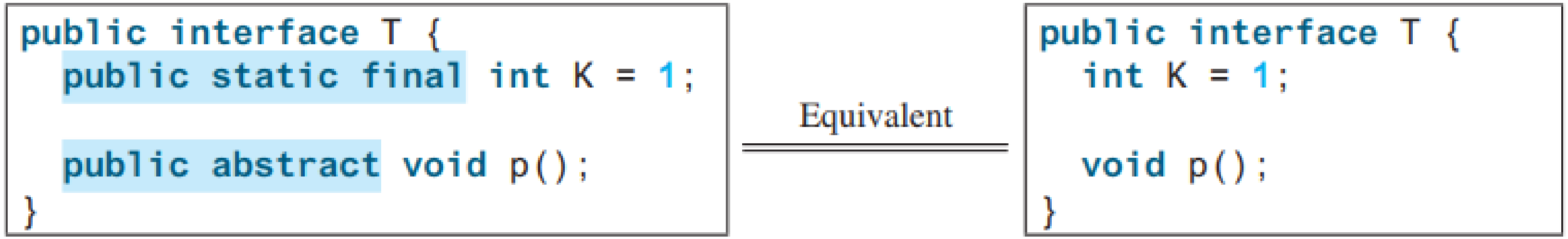
Edible.java (http://www.cs.armstrong.edu/liang/intro10e/html/Edible.html)

TestEdible.java (http://www.cs.armstrong.edu/liang/intro10e/html/TestEdible.html)


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

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 T {
  public static final int K = 1;

  public abstract void p();
}
```

Equivalent

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

  void p();
}
```

Although the public modifier may be omitted for a method defined in the interface, the method must be defined public when it is implemented in a subclass.

Interfaces for Cellphone

```java
// GPSEnabled.java
public interface GPSEnabled {
    public void printLocation();
}
```

```java
// RadioEnabled.java
public interface RadioEnabled {
    public void startRadio();
    public void stopRadio();
}
```

```java
public class Cellphone implements GPSEnabled, RadioEnabled {
    public void printLocation() {
      System.out.println("Location");
    }
    public void startRadio() {
      System.out.println("Radio is ON!");
    }
    public void stopRadio() {
      System.out.println("Radio is OFF!");
    }
}
```

```java
// what is the output?
Cellphone ciaoMi = new Cellphone();
ciaoMi.printLocation();
ciaoMi.startRadio();
GPSEnabled samsu = new Cellphone();
samsu.printLocation();
samsu.startRadio();
```

Quiz time

- Create an interface of AIEnabled
- The interface should include the following methods:
  - turnonChatbot(): turn on chatbot module
  - turnoffChatbot(): turn off chatbot module
  - checkChatbot(): check if chatbot module is on
- Implement AIEnabled in Cellphone
- Test your code

Solution: AIEnabled.java

```
public interface AIEnabled {

      // turn on chatbot module
      void turnonChatbot();

      // turn off chatbot module
      void turnoffChatbot();

      // check if chatbot module is on
      boolean checkChatbot();

}
```

Solution: Cellphone.java

```
public class Cellphone implements AIEnabled {

// ...

  boolean chatbotModule = false;
  public void turnonChatbot() { chatbotModule = true; }

  public void turnoffChatbot() { chatbotModule = false; }

  public boolean checkChatbot() { return chatbotModule; }

}
```

Test the code

```
AIEnabled ciaoMi = new Cellphone();
System.out.println(ciaoMi.checkChatbot());
ciaoMi.turnonChatbot();
System.out.println(ciaoMi.checkChatbot());
ciaoMi.turnoffChatbot();
System.out.println(ciaoMi.checkChatbot());
```

`default` interface methods (Java 8+)

A default method provides a default implementation for the method in the interface. A class that implements the interface may simply:

- use the default implementation for the method, or
- override the method with a new implementation.

This feature enables you to add a new method to an existing interface with a default implementation without having to rewrite the code for the existing classes that implement this interface.

default interface methods (Java 8+): example

Interfaces for Cellphone: Say, you want to add a new method for interface RadioEnabled

```java
public interface RadioEnabled {
    public void startRadio();
    public void stopRadio();

    public void recordRadio(); // let's add this

}
```

What might happen?

```java
public class Cellphone implements RadioEnabled {
    public void startRadio() {
      // start radio
    }
    public void stopRadio() {
      // stop radio
    }
}
```

default interface methods (Java 8+): example

Interfaces for Cellphone: Say, you want to add a new method for interface RadioEnabled

```
public interface RadioEnabled {
    public void startRadio();
    public void stopRadio();
    default public void recordRadio() {
            System.out.println("Recording radio..");
    }
}
```

```
public class Cellphone implements RadioEnabled {
    public void startRadio() {
      // start radio
    }
    public void stopRadio() {
      // stop radio
    }
}
```

Don't do this!

```
public interface GPSEnabled {
    public void printLocation();
}

public interface RadioEnabled {
    public int printLocation();
    public void startRadio();
    public void stopRadio();
}

public class Cellphone implements GPSEnabled, RadioEnabled {
    public void printLocation() {
      // return location
    }

    // ...

}
```

Don't do this!

```
public interface GPSEnabled {
    public void printLocation();
}

public interface RadioEnabled {
    public int printLocation();
    public void startRadio();
    public void stopRadio();
}

public class Cellphone implements GPSEnabled, RadioEnabled {
    public void printLocation() {
      // return location
    }

    // ...

}
```

What's wrong?
Name collision: Overlapped methods with different return types!

Quiz time: List, ArrayList, and LinkedList

```
public static void main(String[] args) {

        List<String> lst1 = new ArrayList<String>();
        lst1.add("A");lst1.add("B");lst1.add("C");
        System.out.println(lst1.contains("A"));
        ((ArrayList<String>) lst1).ensureCapacity(100);
        System.out.println(lst1);

        List<String> lst2 = new LinkedList<String>();
        lst2.add("A");lst2.add("B");lst2.add("C");
        System.out.println(lst2.contains("A"));
        ((LinkedList<String>) lst2).addFirst("9");
        System.out.println(lst2);
}
```

Both AL and LL are lists, but implemented differently:
https://dzone.com/storage/temp/895349-arraylist-linkedlistt.png

What can you observe?

Quiz time: List, ArrayList, and LinkedList

```
public static void main(String[] args) {

        List<String> lst1 = new ArrayList<String>();
        lst1.add("A");lst1.add("B");lst1.add("C");
        System.out.println(lst1.contains("A"));
        ((ArrayList<String>) lst1).ensureCapacity(100);
        System.out.println(lst1);

        List<String> lst2 = new LinkedList<String>();
        lst2.add("A");lst2.add("B");lst2.add("C");
        System.out.println(lst2.contains("A"));
        ((LinkedList<String>) lst2).addFirst("9");
        System.out.println(lst2);
}
```

Both AL and LL are lists, but implemented differently:
https://dzone.com/storage/temp/895349-arraylist-linkedlistt.png

Any other methods that exist in LinkedList only?

Quiz time: List, ArrayList, and LinkedList

```
public static void main(String[] args) {

        List<String> lst1 = new ArrayList<String>();
        lst1.add("A");lst1.add("B");lst1.add("C");
        System.out.println(lst1.contains("A"));
        ((ArrayList<String>) lst1).ensureCapacity(100);
        System.out.println(lst1);

        List<String> lst2 = new LinkedList<String>();
        lst2.add("A");lst2.add("B");lst2.add("C");
        System.out.println(lst2.contains("A"));
        ((LinkedList<String>) lst2).addFirst("9");
        System.out.println(lst2);
}
```

Both AL and LL are lists, but implemented differently:
https://dzone.com/storage/temp/895349-arraylist-linkedlistt.png

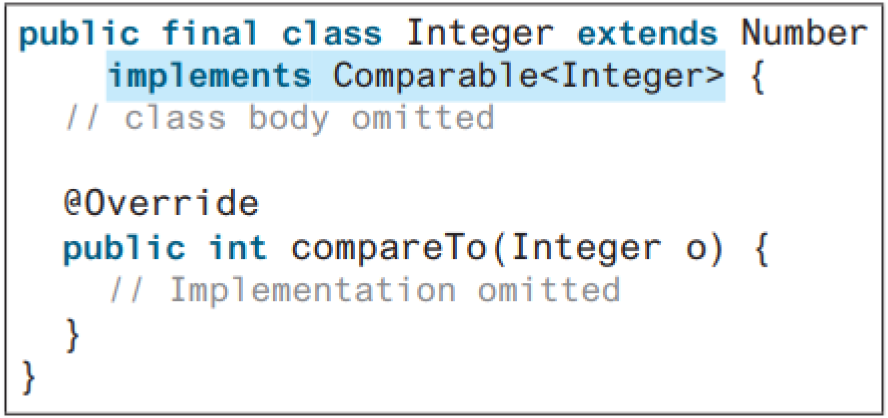
Any other classes implementing List?

A tour to source code of:
List, ArrayList, and LinkedList

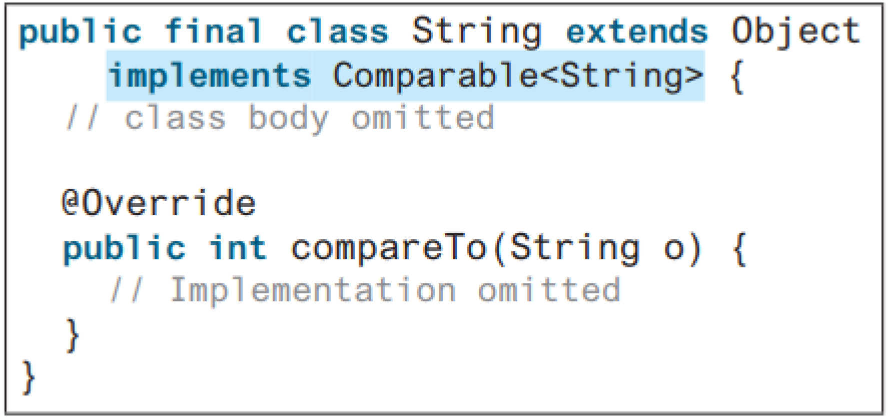
- http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java/util/List.java
- http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java/util/ArrayList.java
- http://hg.openjdk.java.net/jdk8/jdk8/jdk/file/tip/src/share/classes/java/util/LinkedList.java

The Comparable Interface

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

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



```
public final class Integer extends Number
    implements Comparable<Integer> {
  // class body omitted

  @Override
  public int compareTo(Integer o) {
    // Implementation omitted
  }
}
```



```
public final class String extends Object
    implements Comparable<String> {
  // class body omitted

  @Override
  public int compareTo(String o) {
    // Implementation omitted
  }
}
```

Example: The Comparable Interface

- What is the output?

```
System.out.println(new Integer(3).compareTo(new Integer(5)));
System.out.println("ABC".compareTo("ABC"));
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));
```

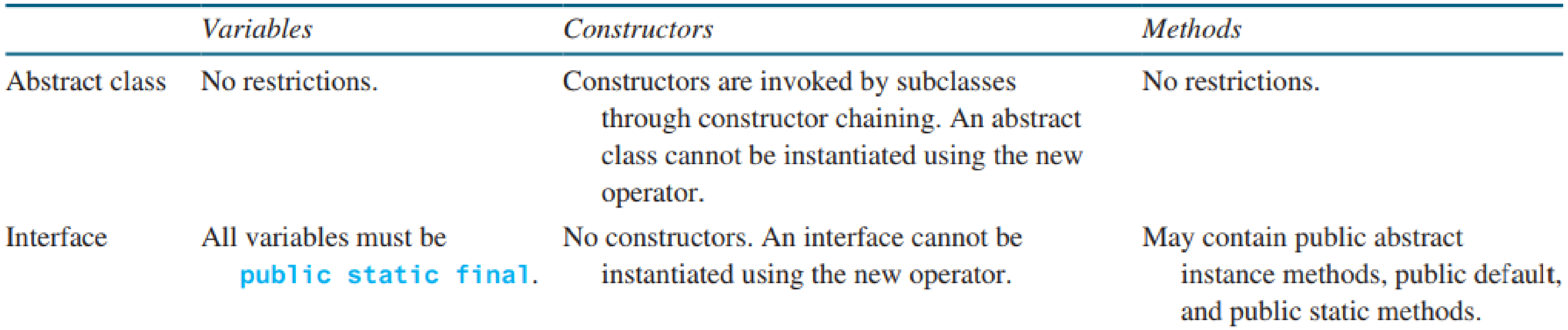
- Have a look at Comparable interface in JavaDoc, and try to create an Employee class implementing the Comparable interface, such that employees can be compared based on the order of their instantiations (employees created first get more priorities than those created later).
  From your implementation, demonstrate how you can sort an Employee list, based on the employee instantiation order.

Abstract Classes vs Interface

Variables, Constructors, and Methods

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.	May contain public abstract instance methods, public default, and public static methods.

Inheritance of abstract class and interface

- Java allows only single inheritance for class extension, but allows multiple extensions for interfaces.

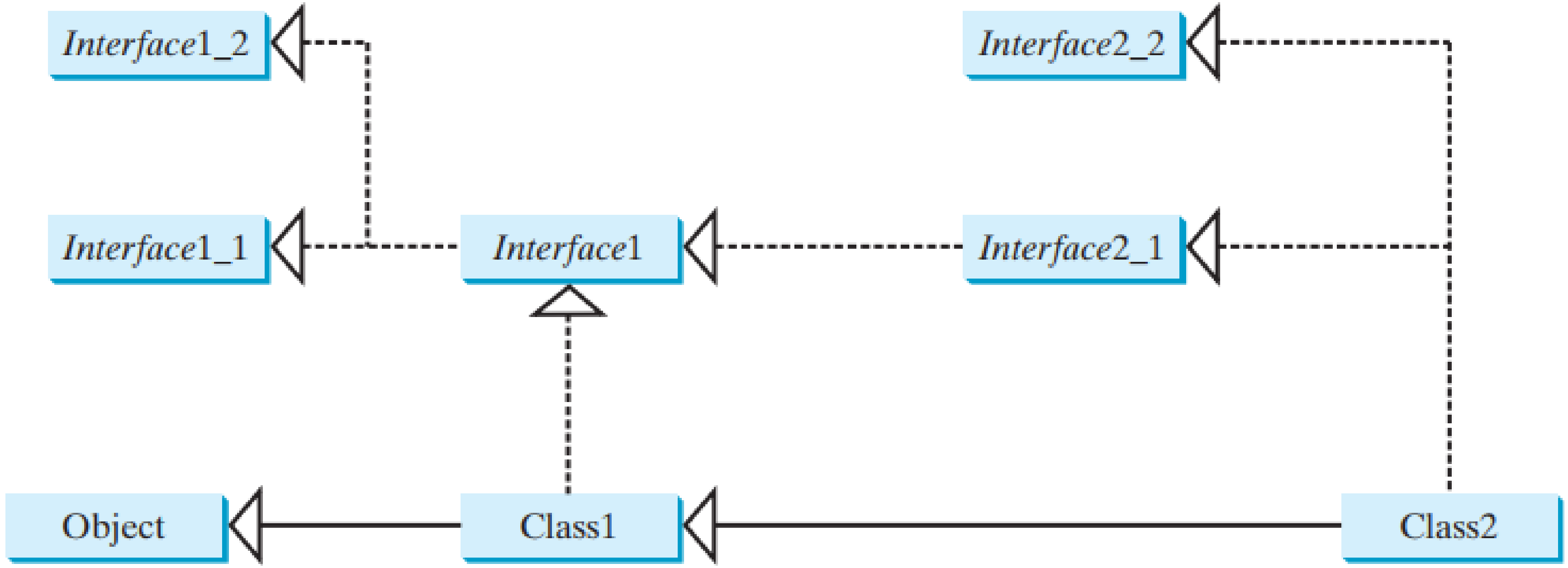
```
public class NewClass extends BaseClass
    implements Interface1, ... , InterfaceN {
  ...
}
```

- An interface can inherit other interfaces using the **extends** keyword. Such an interface is called a subinterface.

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

Inheritance of abstract class and interface (cont.)

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.



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 (or 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.