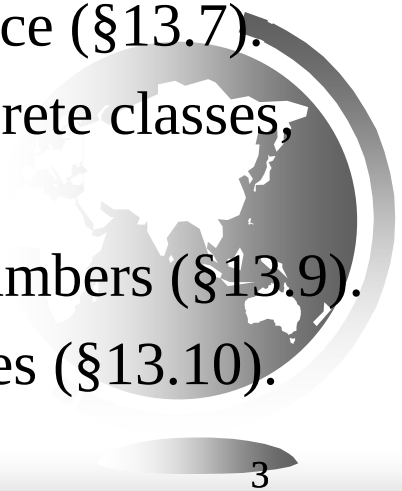
12 Abstract Classes and Interfaces

Motivations

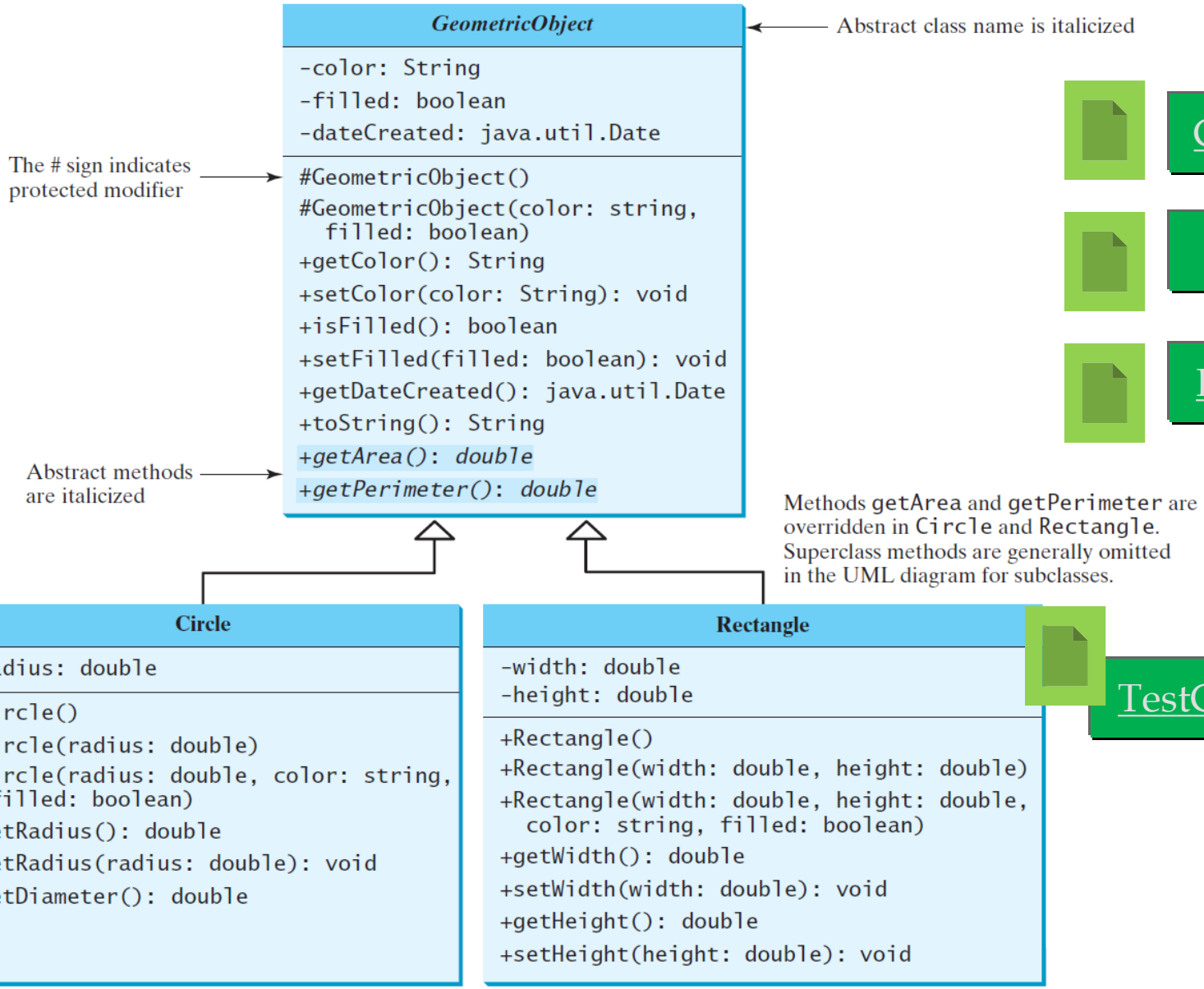
- You have learned how to write simple programs to create and display GUI components. Can you write the code to respond to user actions, such as clicking a button to perform an action?
- In order to write such code, you have to know about interfaces. An *interface* is for defining common behavior for classes (including unrelated classes). Before discussing interfaces, we introduce a closely related subject: abstract classes.

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



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

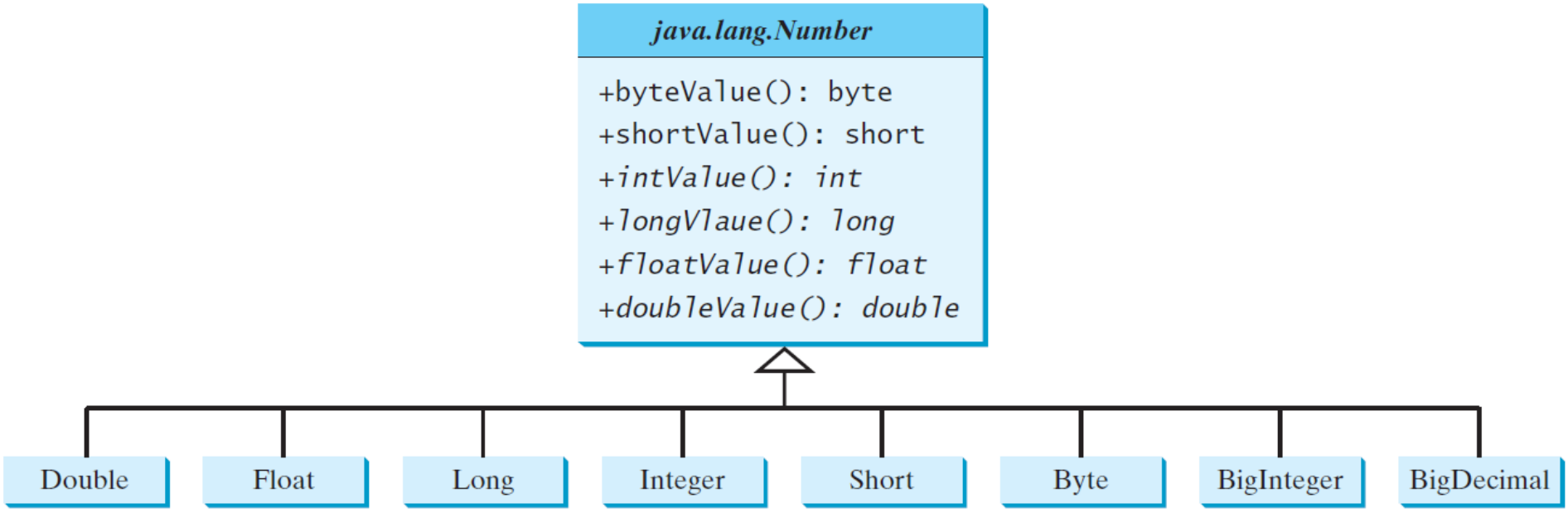
A subclass can override a method from its superclass to define it abstract. This is rare, but useful when the implementation of the method in the superclass becomes invalid in the subclass. In this case, the subclass must be defined abstract.

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];
```

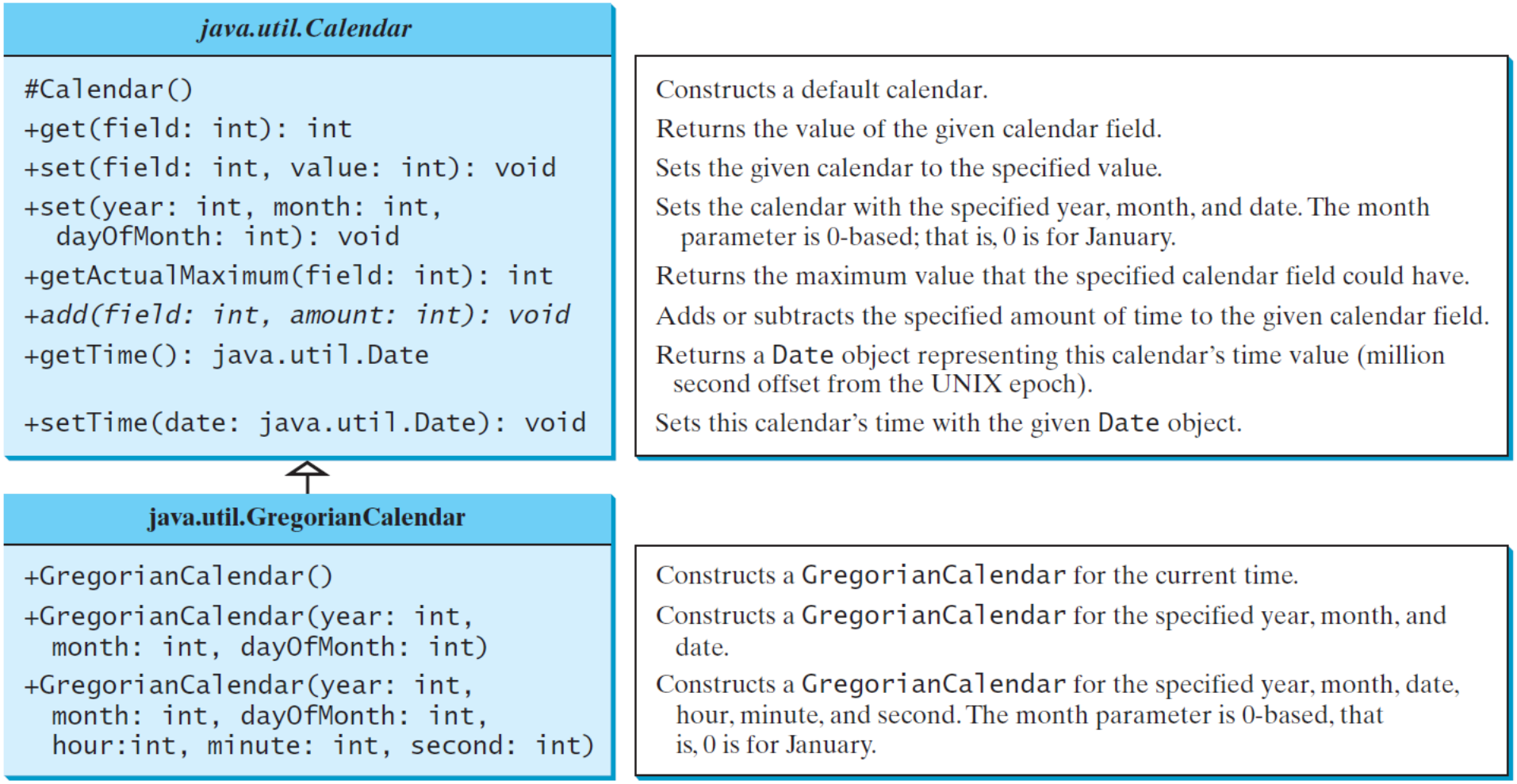
Case Study: the Abstract Number Class



LargestNumbers

Run

The Abstract Calendar Class and Its GregorianCalendar subclass



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 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).

Getting Date/Time Information from Calendar

- calendar.set(Calender.DAY_OF_MONTH,1)，将calendar设置为当月的第一天
- calendar.add(Calendar.DAY_OF_MONTH,5)，给日历的当前时间加5天；calendar.add(Calendar.DAY_OF_MONTH,-5)，给日历的当前时间减5天；
- calendar.getActualMaximum(Calendar.DAY_OF_MONTH)，如果是三月的calendar，则返回31

TestCalendar

Run

LocalDate

- Java的时间日期API一直以来都是被诟病的东西，为了解决这一问题，Java 8中引入了新的时间日期API，其中包括LocalDate、LocalTime、LocalDateTime、Clock、Instant等类，这些的类的设计都使用了不变模式，因此是线程安全的设计。

```
// 取当前日期:
LocalDate today = LocalDate.now(); // -> 2014-12-24
// 根据年月日取日期:
LocalDate crischristmas = LocalDate.of(2014, 12, 25); // -> 2014-12-25
// 根据字符串取:
LocalDate endOfFeb = LocalDate.parse("2014-02-28"); // 严格按照ISO yyyy-MM-dd验证，02写成2都不行，当然也有一个重载方法允许自己定义格式
LocalDate.parse("2014-02-29"); // 无效日期无法通过：DateTimeParseException: Invalid date
```

```
// 取本月第1天:
LocalDate firstDayOfThisMonth = today.with(TemporalAdjusters.firstDayOfMonth()); // 2017-03-01
// 取本月第2天:
LocalDate secondDayOfThisMonth = today.withDayOfMonth(2); // 2017-03-02
// 取本月最后一天，再也不用计算是28，29，30还是31:
LocalDate lastDayOfThisMonth = today.with(TemporalAdjusters.lastDayOfMonth()); // 2017-12-31
// 取下一天:
LocalDate firstDayOf2015 = lastDayOfThisMonth.plusDays(1); // 变成了2018-01-01
// 取2017年1月第一个周一，用Calendar要死掉很多脑细胞:
LocalDate firstMondayOf2015 = LocalDate.parse("2017-01-01").with(TemporalAdjusters.firstInMonth(DayOfWeek.MONDAY)); // 2017-01-02
```

Interfaces

What is an interface?

Why is an interface useful?

How do you define an interface?

How do you use an interface?

What is an interface? Why is an interface useful?

An interface is a classlike 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

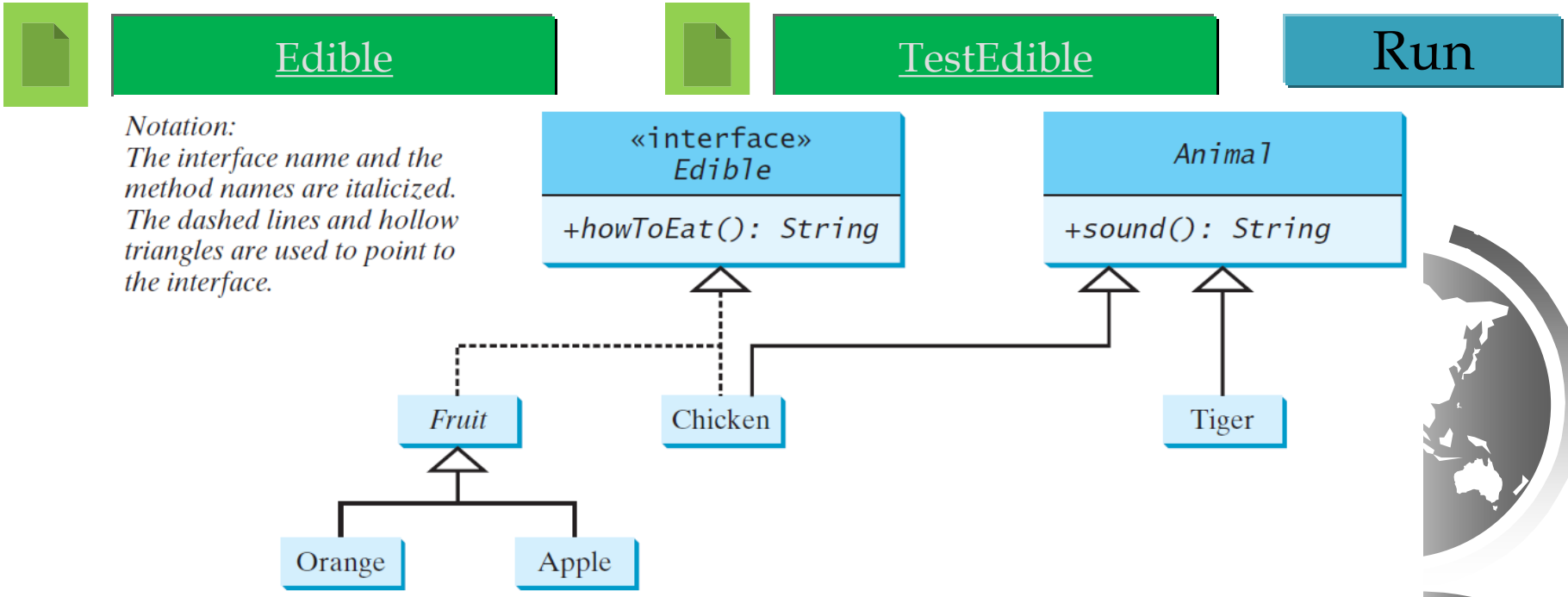
An interface is treated like a special class in Java. Each interface is compiled into a separate bytecode file, just like a regular class.

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).



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

```
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

```
1  System.out.println(new Integer(3).compareTo(new Integer(5)));
2  System.out.println("ABC".compareTo("ABE"));
3  java.util.Date date1 = new java.util.Date(2013, 1, 1);
4  java.util.Date date2 = new java.util.Date(2012, 1, 1);
5  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
```

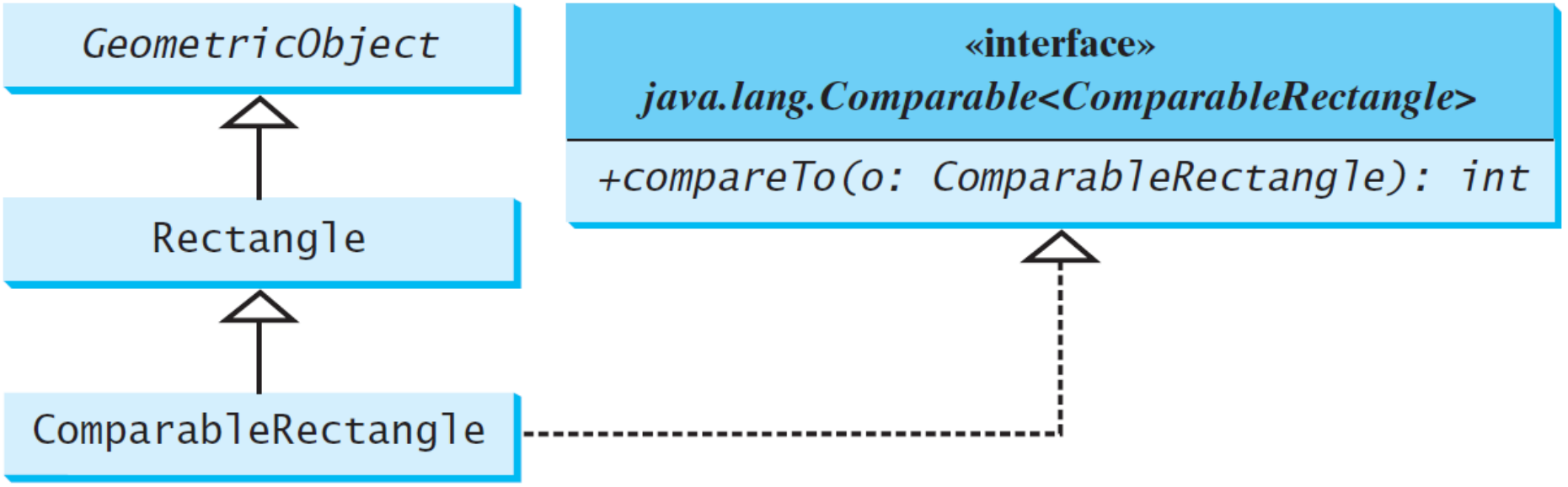
```
s instanceof String
s instanceof Object
s instanceof Comparable
```

```
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



ComparableRectangle

SortRectangles

Run

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

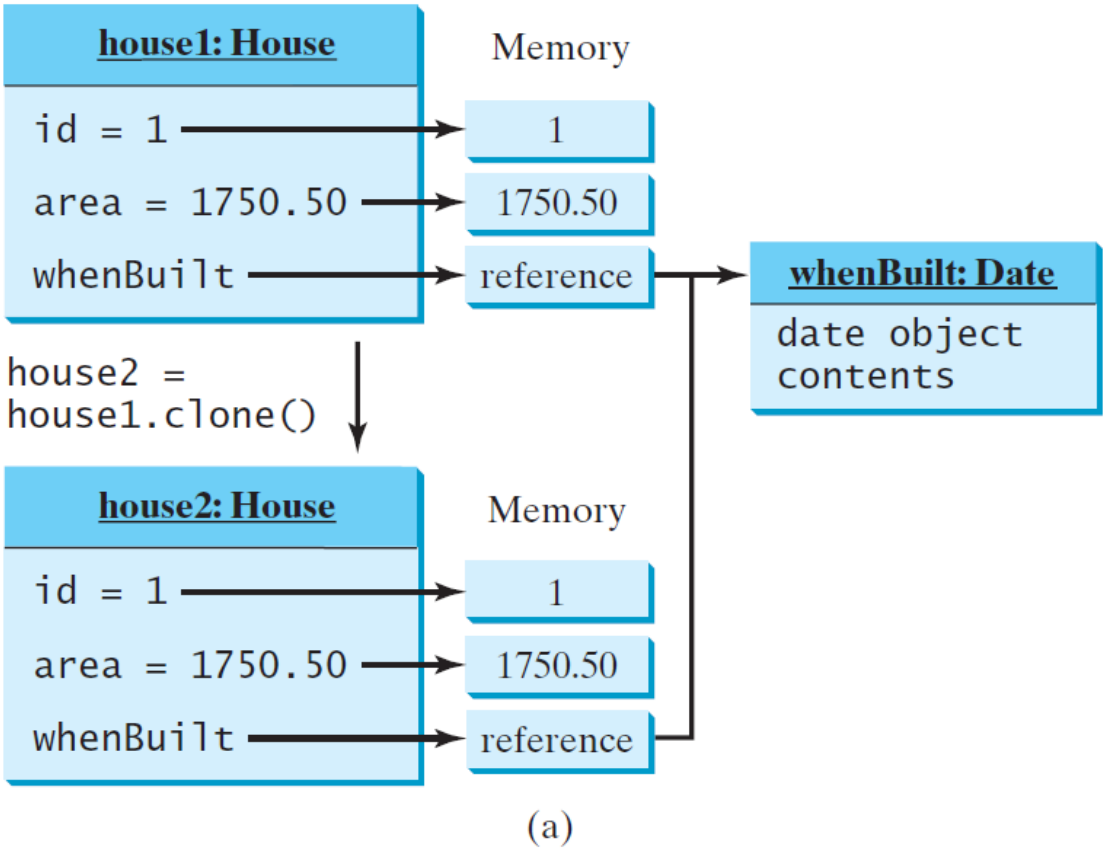
```
@Override /** Override the protected clone method defined in
  the Object class, and strengthen its accessibility */
public Object clone() {
  try {
    return super.clone();
  }
  catch (CloneNotSupportedException ex) {
    return null;
  }
}
```

Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);

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

Shallow Copy

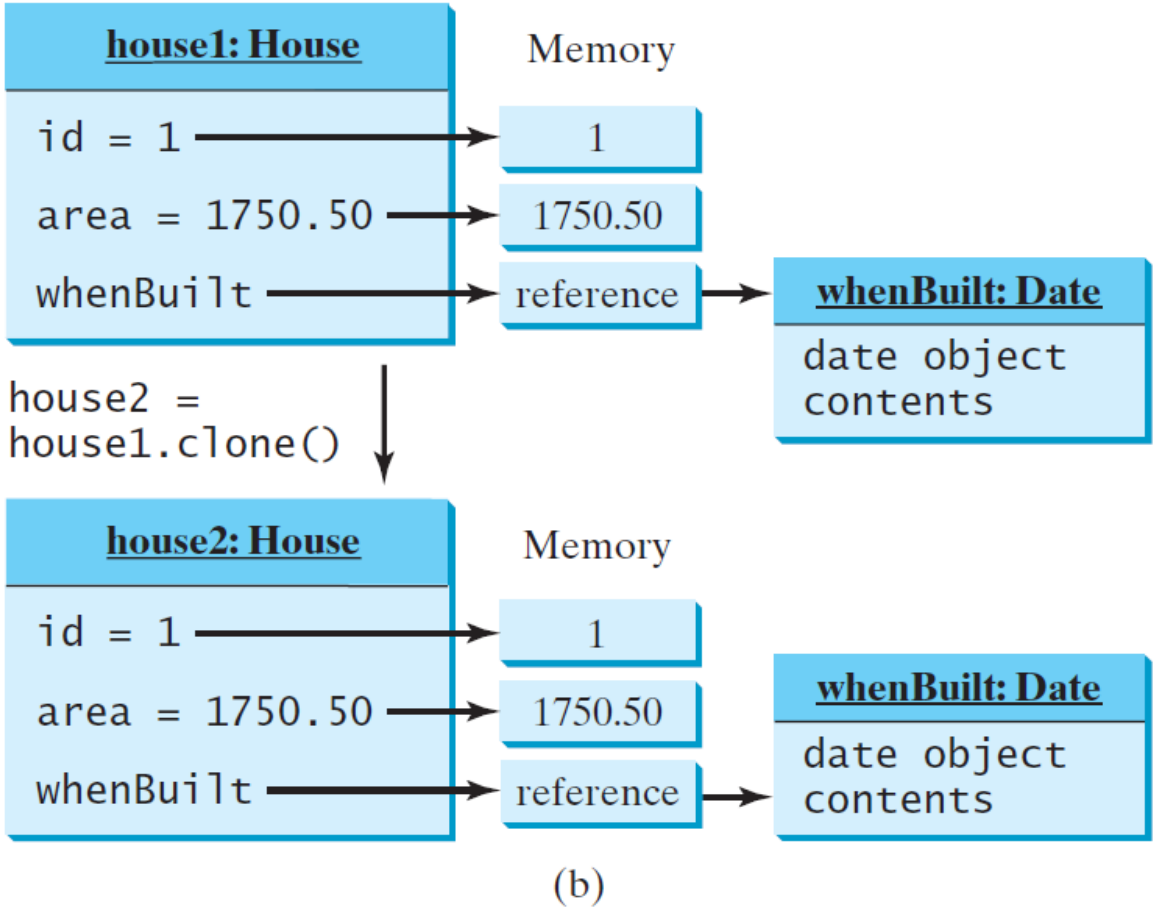


Shallow vs. Deep Copy

House house1 = new House(1, 1750.50);

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

Deep Copy



Shallow vs. Deep Copy

```
Public Object clone()throws CloneNotSupportedException{
        House houseClone = (House)super.clone();
        houseClone.whenBuilt =
(java.util.Date)(whenBuilt.clone());
        return houseClone;
}
```

如何Deep Copy Stack?

```
import java.util.Arrays;

public class Stack implements Cloneable {
    private Object[] elements;
    private int size = 0;
    private static final int DEFAULT_INITIAL_CAPACITY = 16;

    public Stack() {
        this.elements = new Object[DEFAULT_INITIAL_CAPACITY];
    }

    public void push(Object e) {
        ensureCapacity();
        elements[size++] = e;
    }

    public Object pop() {
        if (size == 0)
            throw new EmptyStackException();
        Object result = elements[--size];
        elements[size] = null; // Eliminate obsolete reference
        return result;
    }

    public boolean isEmpty() {
        return size == 0;
    }

    // Ensure space for at least one more element.
    private void ensureCapacity() {
        if (elements.length == size)
            elements = Arrays.copyOf(elements, 2 * size + 1);
    }

}
```

```
@Override
public Stack clone() {
    try {
        Stack result = (Stack) super.clone();
        result.elements = elements.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        throw new AssertionError();
    }
}
```

如何Deep Copy HashTable?

```
public class HashTable implements Cloneable {
    private Entry[] buckets;

    private static class Entry{
        Object key;
        Object value;
        Entry next;

        Entry(Object key, Object value, Entry next) {
            this.key = key;
            this.value = value;
            this.next = next;
        }
    }
    ......
```

```
//递归clone这个HashTable数组，如下:
public HashTable clone(){
    HashTable result = new HashTable();
    try {
        result = (HashTable) super.clone();
        result.buckets = buckets.clone();
        return result;
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
    return result;
}
```

是否正确？

```
public HashTable  clone(){
    HashTable  result = new HashTable();
    try {
        result = (HashTable ) super.clone();
        result.buckets = new Entry[buckets.length];
        for(int i=0;i<buckets.length;i++){
            result.buckets[i] = buckets[i].deepCopy();
        }
        return result;
    } catch (CloneNotSupportedException e) {
        e.printStackTrace();
    }
    return result;
}
```

```
private static class Entry{
    Object key;
    Object value;
    Entry next;

    Entry(Object key, Object value, Entry next) {
        this.key = key;
        this.value = value;
        this.next = next;
    }

   Entry deepCopy(){
        Entry result = new Entry(key,value,next);
        for(Entry p=result;p.next !=null; p=p.next){
            p.next=new Entry(p.next.key,p.next.value,p.next.next);
    }
        return result;
}
```

Interfaces vs. Abstract Classes

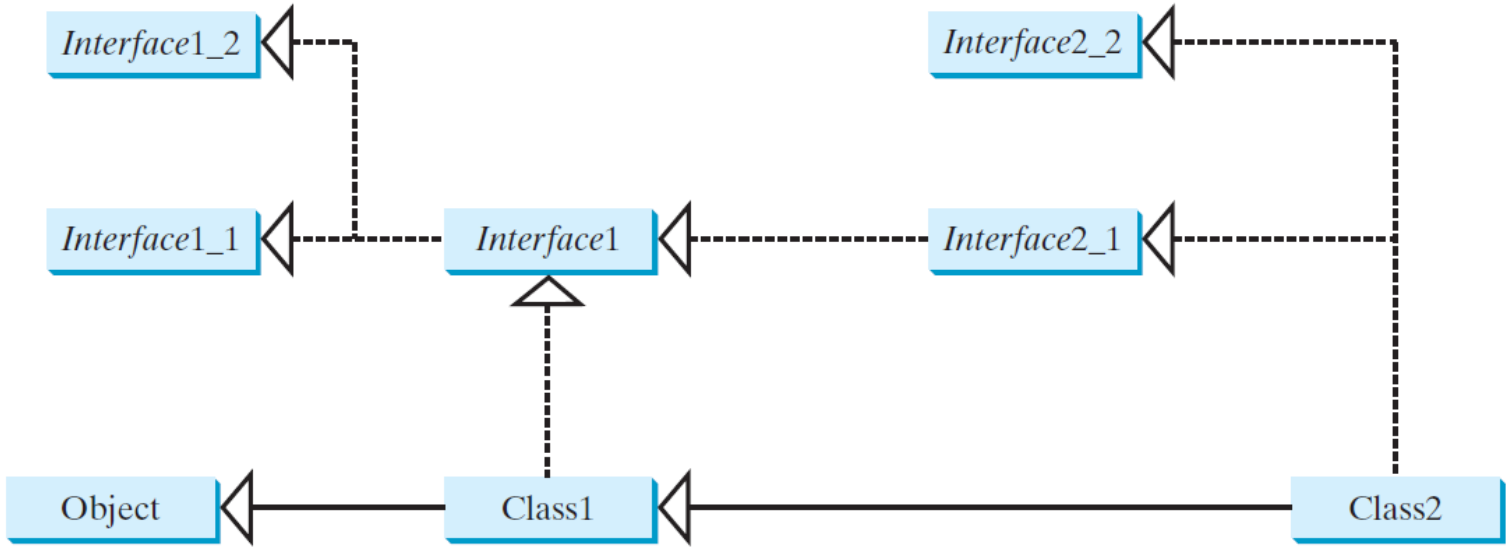
In an interface, the data must be constants; an abstract class can have all types of data.

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.

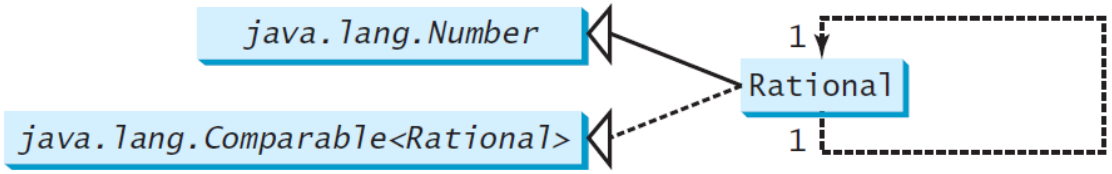
Whether to use an interface or a class?

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 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



Rational
-numerator: long -denominator: long
+Rational() +Rational(numerator: long, denominator: long) +getNumerator(): long +getDenominator(): long +add(secondRational: Rational): Rational +subtract(secondRational: Rational): Rational +multiply(secondRational: Rational): Rational +divide(secondRational: Rational): Rational +toString(): String -gcd(n: long, d: long): long

The numerator of this rational number.
The denominator of this rational number.
Creates a rational number with numerator 0 and denominator 1.
Creates a rational number with a specified numerator and denominator.
Returns the numerator of this rational number.
Returns the denominator of this rational number.
Returns the addition of this rational number with another.
Returns the subtraction of this rational number with another.
Returns the multiplication of this rational number with another.
Returns the division of this rational number with another.
Returns a string in the form "numerator/denominator." Returns the numerator if denominator is 1.
Returns the greatest common divisor of n and d.

Rational

TestRationalClass

Run

Designing a Class

(Coherence) 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.

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 equals method and the toString method defined in the Object 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.

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. The gcd method in the Rational class is private, for example, because it is 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.

10个面向对象的设计原则

- **原则1：DRY(Don't repeat yourself)**

  即不要写重复的代码，而是用“abstraction”类来抽象公有的东西。如果你需要多次用到一个硬编码值，那么可以设为公共常量;如果你要在两个以上的地方使用一个代码块，那么可以将它设为一个独立的方法。SOLID设计原则的优点是易于维护，但要注意，不要滥用，duplicate 不是针对代码，而是针对功能。这意味着，即使用公共代码来验证OrderID和SSN，二者也不会是相同的。使用公共代码来实现两个不同的功能，其实就是近似地把这两个功能永远捆绑到了一起，如果OrderID改变了其格式，SSN验证代码也会中断。因此要慎用这种组合，不要随意捆绑类似但不相关的功能。

10个面向对象的设计原则

- **原则2：封装变化**

在软件领域中唯一不变的就是“Change”，因此封装你认为或猜测未来将发生变化的代码。OOPS设计模式的优点在于易于测试和维护封装的代码。如果你使用Java编码，可以默认私有化变量和方法，并逐步增加访问权限，比如从private到protected和public.有几种Java设计模式也使用封装，比如Factory设计模式是封装“对象创建”，其灵活性使得之后引进新代码不会对现有的代码造成影响。

工厂模式

10个面向对象的设计原则

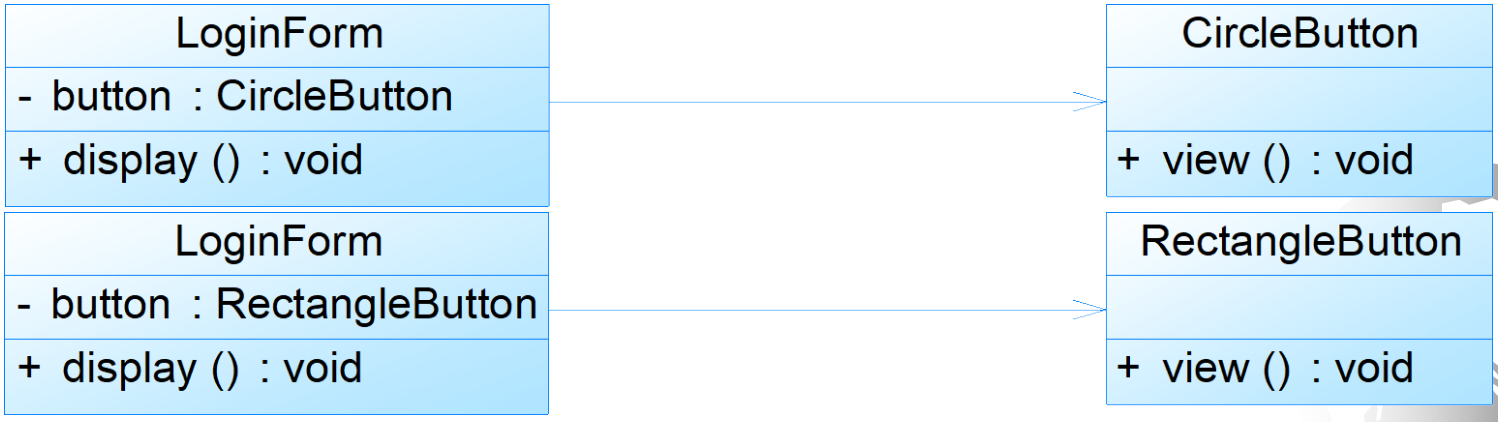
- **原则3：开闭原则**

  一个软件实体应当对扩展开放，对修改关闭。也就是说在设计一个模块的时候，应当使这个模块可以在不被修改的前提下被扩展，即实现在不修改源代码的情况下改变这个模块的行为。

- 其英文定义为：
  - Software entities should be open for extension, but closed for modification.

10个面向对象的设计原则

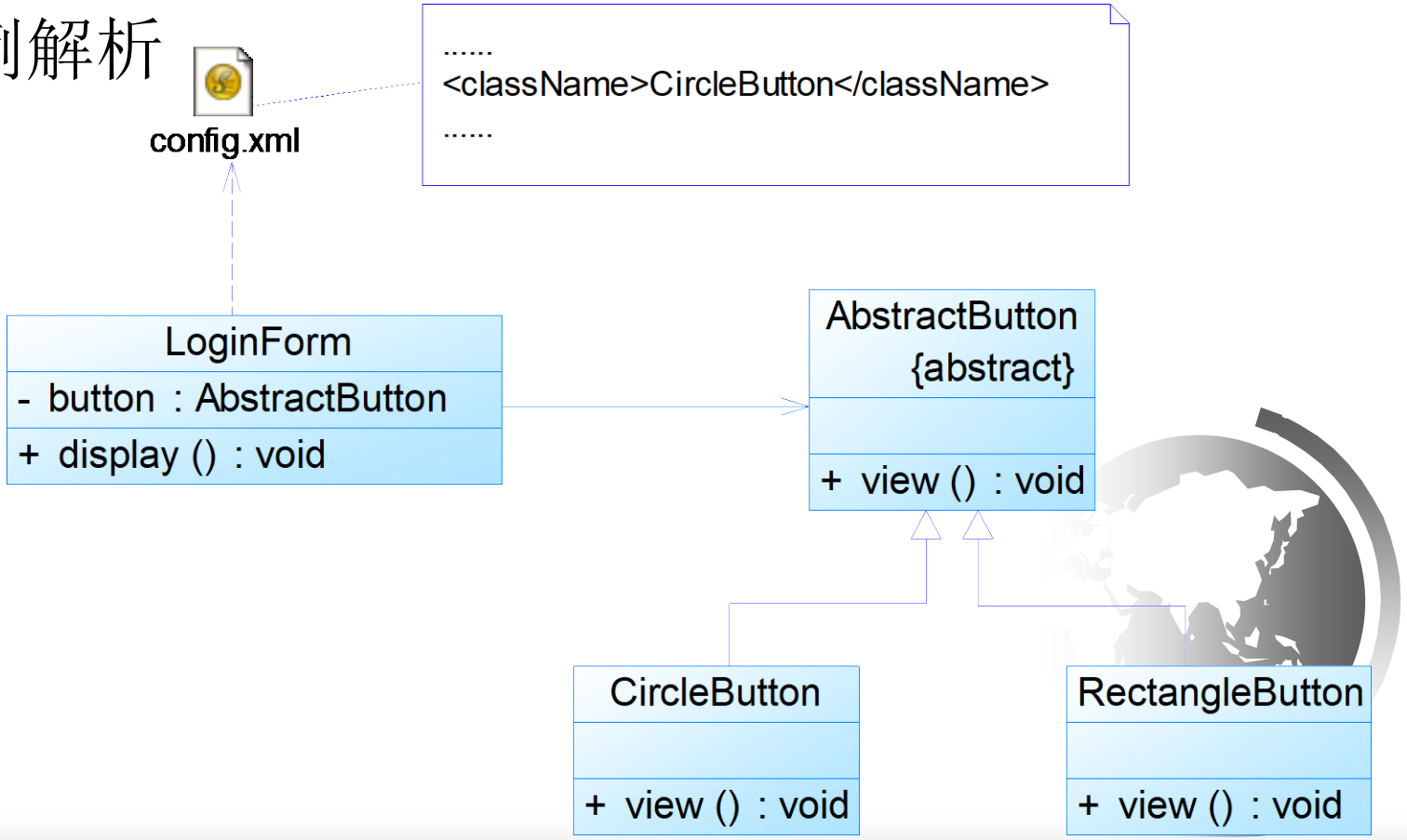
- 开闭原则实例
  - 实例说明
    - 某图形界面系统提供了各种不同形状的按钮，客户端代码可针对这些按钮进行编程，用户可能会改变需求要求使用不同的按钮，原始设计方案如图所示：



    - 现对该系统进行重构，使之满足开闭原则的要求。

10个面向对象的设计原则

- 开闭原则实例
  - 实例解析



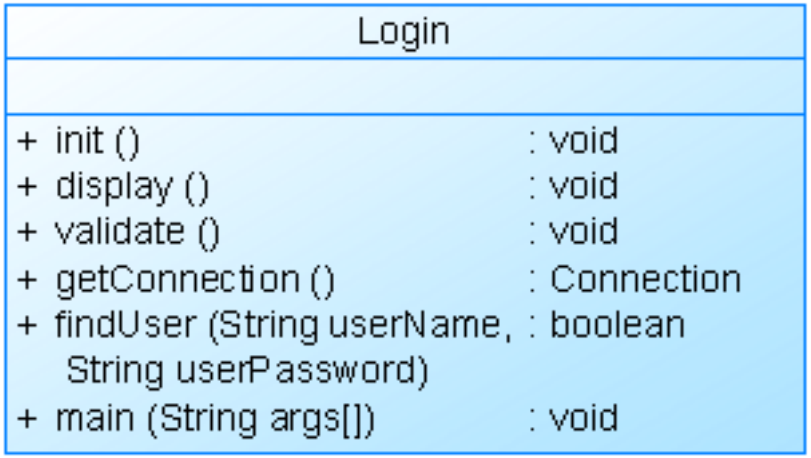
10个面向对象的设计原则

- **原则4：单一职责原则**

  类被修改的几率很大，因此应该专注于单一的功能。如果你把多个功能放在同一个类中，功能之间就形成了关联，改变其中一个功能，有可能中止另一个功能，这时就需要新一轮的测试来避免可能出现的问题。

10个面向对象的设计原则

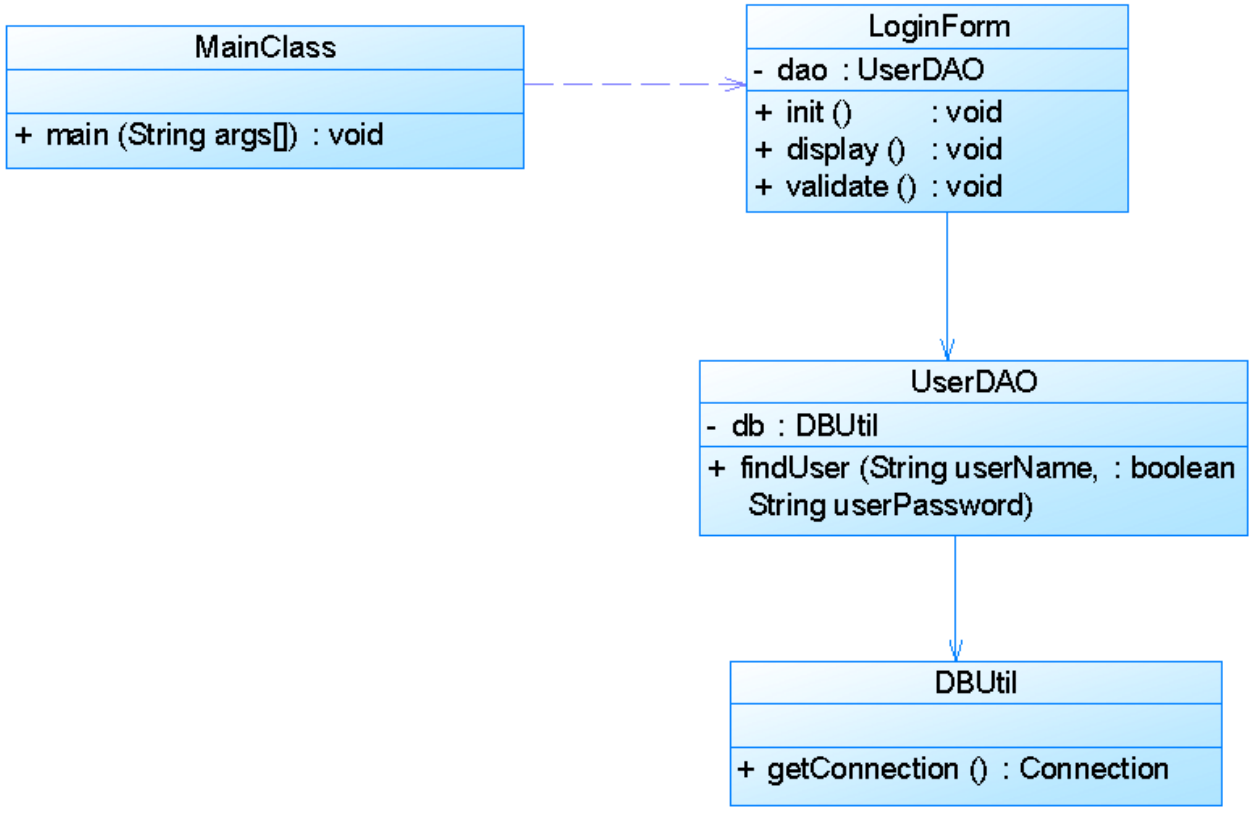
- 单一职责原则实例
  - 实例说明
    - 某基于Java的C/S系统的“登录功能”通过如下登录类(Login)实现：



    - 现使用单一职责原则对其进行重构。

10个面向对象的设计原则

- 单一职责原则实例
  - 实例解析



10个面向对象的设计原则

- **原则5：依赖注入或倒置原则**
  - 依赖**倒置**原则(Dependence Inversion Principle, DIP)的定义如下：
    - 高层模块不应该依赖低层模块，它们都应该依赖抽象。抽象不应该依赖于细节，细节应该依赖于抽象。
  - 其英文定义为：
    - High level modules should not depend upon low level modules, both should depend upon abstractions. Abstractions should not depend upon details, details should depend upon abstractions.
  - 另一种表述为：
    - 要针对接口编程，不要针对实现编程。
  - 其英文定义为：
    - Program to an interface, not an implementation.

10个面向对象的设计原则

- **原则5：依赖注入或倒置原则**

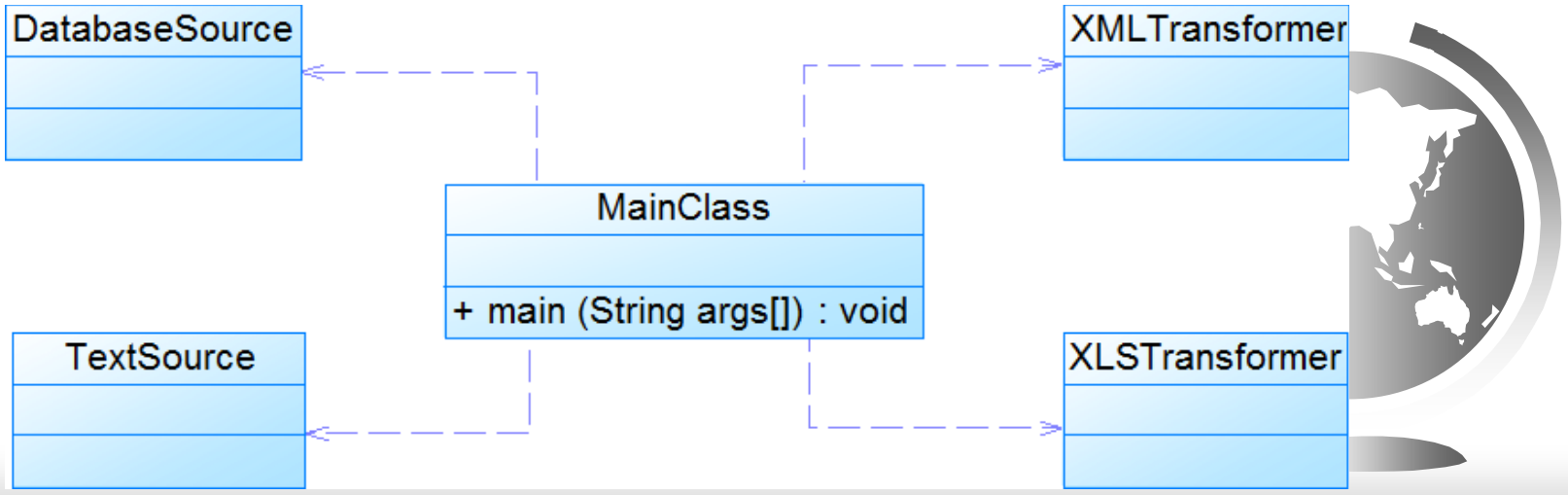
这个设计原则的亮点在于任何被DI框架注入的类很容易用mock对象进行测试和维护，因为对象创建代码集中在框架中，客户端代码也不混乱。有很多方式可以实现依赖倒置，比如像AspectJ等的AOP(Aspect Oriented programming)框架使用的字节码技术，或Spring框架使用的代理等。

10个面向对象的设计原则

- 依赖倒转原则分析
  - 依赖注入
    - 构造注入(Constructor Injection)：通过构造函数注入实例变量。
    - 设值注入(Setter Injection)：通过Setter方法注入实例变量。
    - 接口注入(Interface Injection)：通过接口方法注入实例变量。

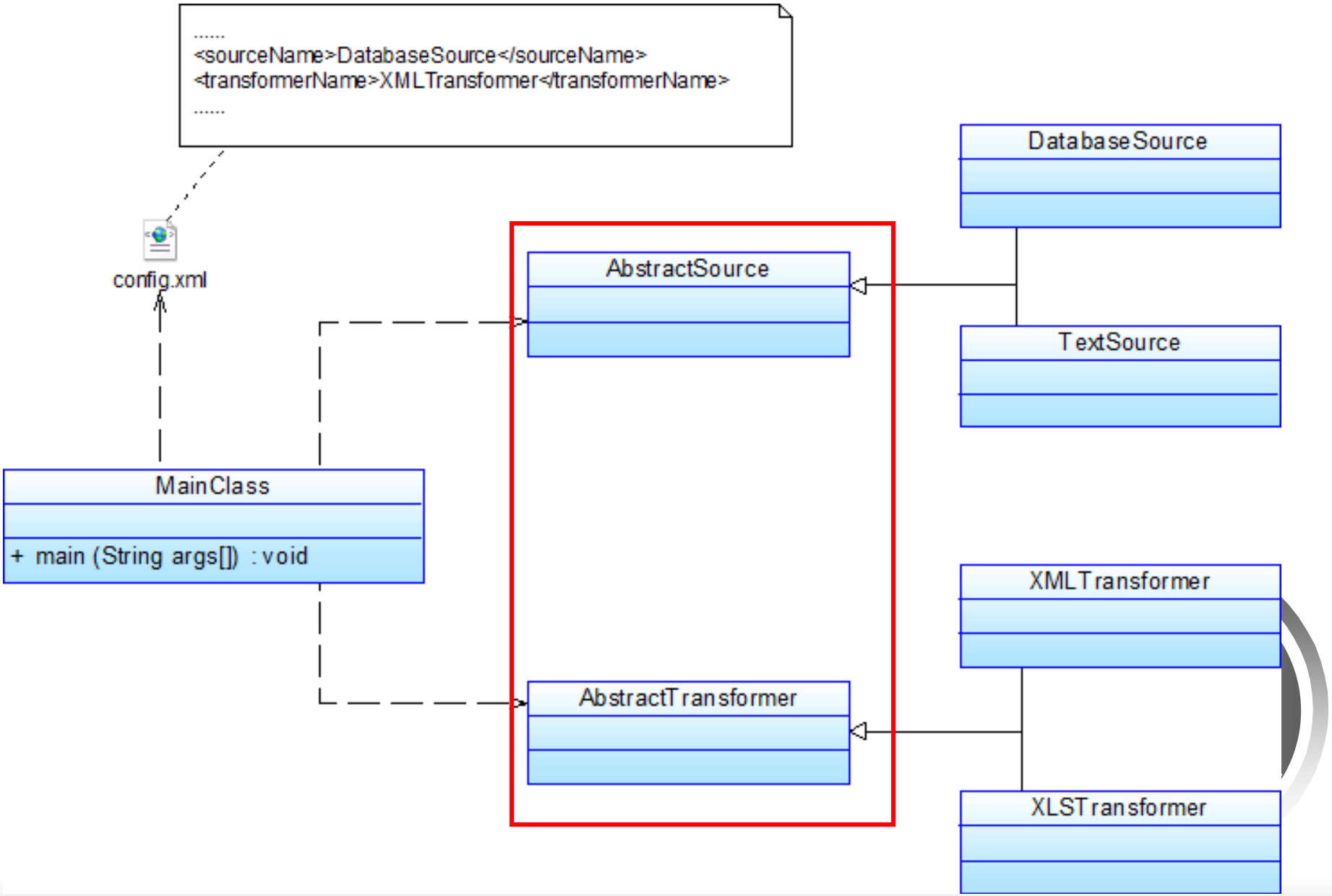
10个面向对象的设计原则

- 依赖倒转原则实例
  - 实例说明
    - 某系统提供一个数据转换模块，可以将来自不同数据源的数据转换成多种格式，如可以转换来自数据库的数据(DatabaseSource)、也可以转换来自文本文件的数据(TextSource)，转换后的格式可以是XML文件(XMLTransformer)、也可以是XLS文件(XLSTransformer)等。



10个面向对象的设计原则

- 依赖倒转原则实例
  - 实例说明
    - 由于需求的变化，该系统可能需要增加新的数据源或者新的文件格式，每增加一个新的类型的数据源或者新的类型的文件格式，客户类MainClass都需要修改源代码，以便使用新的类，但违背了开闭原则。现使用依赖倒转原则对其进行重构。


.....
<sourceName>DatabaseSource</sourceName>
<transformerName>XMLTransformer</transformerName>
.....
config.xml
DatabaseSource
AbstractSource
TextSource
MainClass
+ main (String args[]) : void
XMLTransformer
AbstractTransformer
XLSTransformer

10个面向对象的设计原则

- **原则6：合成复用原则**
  - 合成复用原则定义
    - 合成复用原则(Composite Reuse Principle, CRP)又称为组合/聚合复用原则(Composition/ Aggregate Reuse Principle, CARP)，其定义如下：
      - 尽量使用对象组合，而不是继承来达到复用的目的。
    - 其英文定义为：
      - Favor composition of objects over inheritance as a reuse mechanism.

10个面向对象的设计原则

- 合成复用原则分析
  - 合成复用原则就是指在一个新的对象里通过关联关系（包括组合关系和聚合关系）来使用一些已有的对象，使之成为新对象的一部分；新对象通过委派调用已有对象的方法达到复用其已有功能的目的。简言之：要尽量使用组合/聚合关系，少用继承。

10个面向对象的设计原则

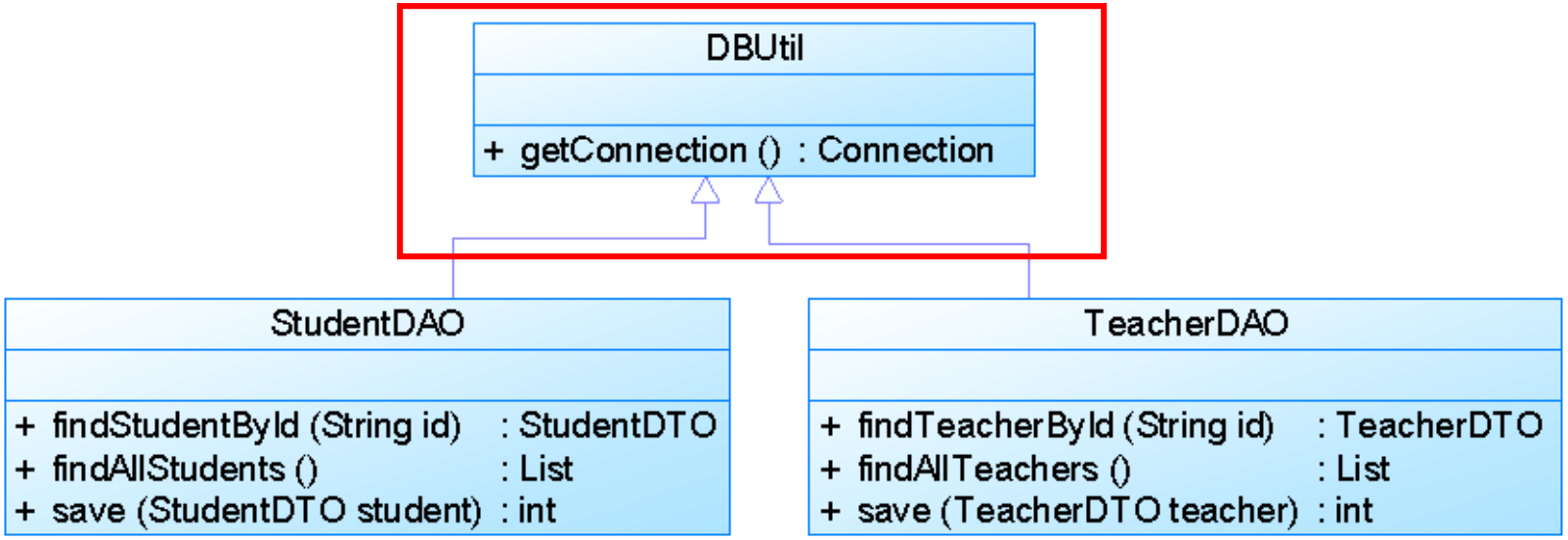
- 合成复用原则分析
  - 在面向对象设计中，可以通过两种基本方法在不同的环境中复用已有的设计和实现，即通过组合/聚合关系或通过继承。
    - 继承复用：实现简单，易于扩展。破坏系统的封装性；从基类继承而来的实现是静态的，不可能在运行时发生改变，没有足够的灵活性；只能在有限的环境中使用。（“白箱”复用）
    - 组合/聚合复用：耦合度相对较低，选择性地调用成员对象的操作；可以在运行时动态进行。（“黑箱”复用）

10个面向对象的设计原则

- 合成复用原则分析
  - 组合/聚合可以使系统更加灵活，类与类之间的耦合度降低，一个类的变化对其他类造成的影响相对较少，因此一般首选使用组合/聚合来实现复用；其次才考虑继承，在使用继承时，需要严格遵循里氏代换原则，有效使用继承会有助于对问题的理解，降低复杂度，而滥用继承反而会增加系统构建和维护的难度以及系统的复杂度，因此需要慎重使用继承复用。

10个面向对象的设计原则

- 合成复用原则实例
  - 实例说明
    - 某教学管理系统部分数据库访问类设计如图所示：

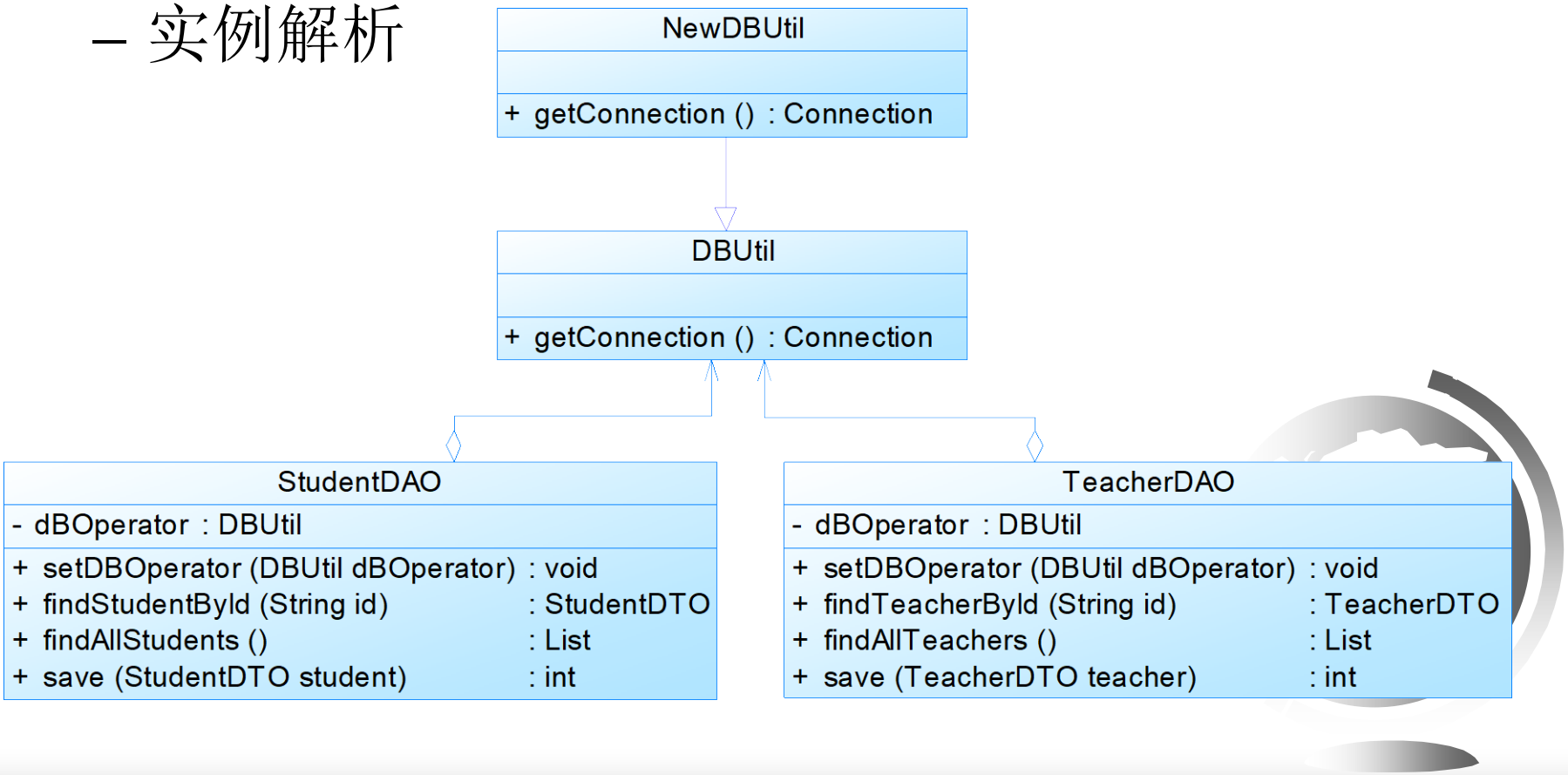


10个面向对象的设计原则

- 合成复用原则实例
  - 实例说明
    - 如果需要更换数据库连接方式，如原来采用JDBC连接数据库，现在采用数据库连接池连接，则需要修改DBUtil类源代码。如果StudentDAO采用JDBC连接，但是TeacherDAO采用连接池连接，则需要增加一个新的DBUtil类，并修改StudentDAO或TeacherDAO的源代码，使之继承新的数据库连接类，这将违背开闭原则，系统扩展性较差。
    - 现使用合成复用原则对其进行重构。

10个面向对象的设计原则

- 合成复用原则实例
  - 实例解析



10个面向对象的设计原则

- **原则7：里氏代换原则(LSP)**

  根据该原则，子类必须能够替换掉它们的基类，也就是说使用基类的方法或函数能够顺利地引用子类对象。LSP原则与单一职责原则和接口分离原则密切相关，如果一个类比子类具备更多功能，很有可能某些功能会失效，这就违反了LSP原则。为了遵循该设计原则，派生类或子类必须增强功能。

10个面向对象的设计原则

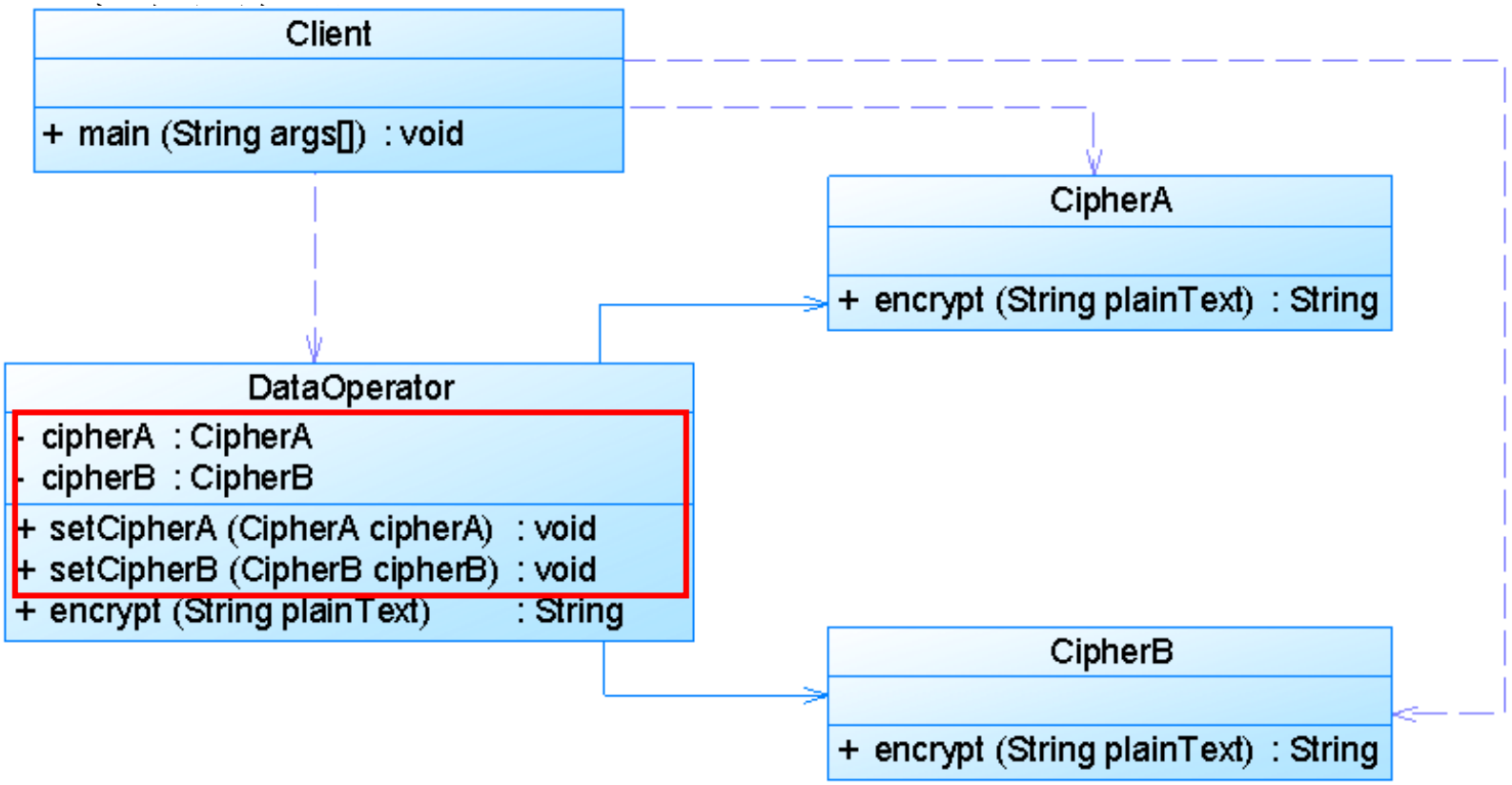
- 里氏代换原则分析

喜欢动物→喜欢猫

因为猫是动物 ☺

10个面向对象的设计原则

- 里氏代换原则实例

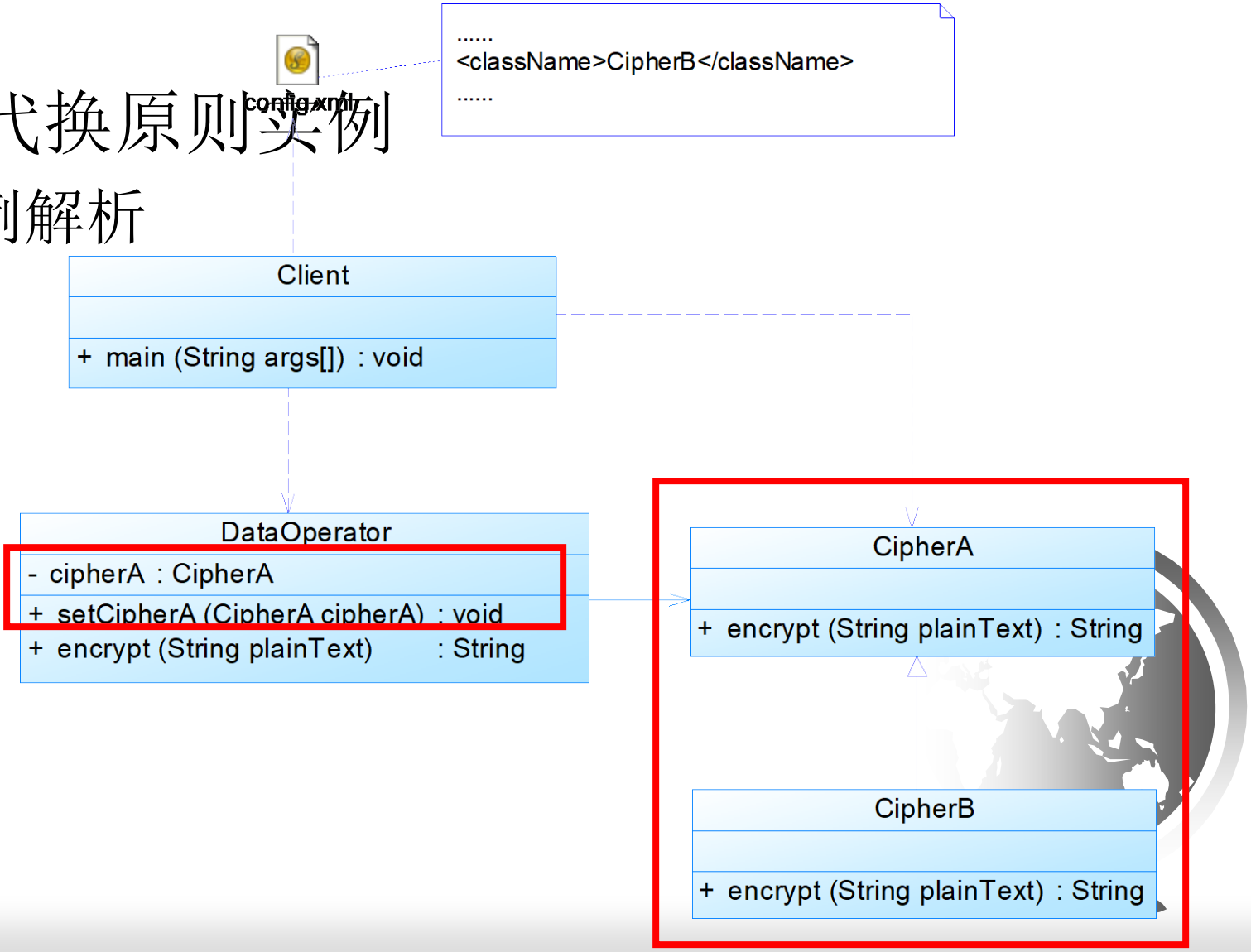


10个面向对象的设计原则

- 里氏代换原则实例
  - 实例说明
    - 如果需要更换一个加密算法类或者增加并使用一个新的加密算法类，如将CipherA改为CipherB，则需要修改客户类Client和数据操作类DataOperator的源代码，违背了开闭原则。
    - 现使用里氏代换原则对其进行重构，使得系统可以灵活扩展，符合开闭原则。

10个面向对象的设计原则

- 里氏代换原则实例
  - 实例解析



10个面向对象的设计原则

- **原则8：**接口隔离**原则**
  - 接口隔离原则定义
    - 接口隔离原则(Interface Segregation Principle, ISP)的定义如下：
      - 客户端不应该依赖那些它不需要的接口。
    - 其英文定义为：
      - Clients should not be forced to depend upon interfaces that they do not use.
    - 注意，在该定义中的接口指的是所定义的方法。
    - 另一种定义方法如下：
      - 一旦一个接口太大，则需要将它分割成一些更细小的接口，使用该接口的客户端仅需知道与之相关的方法即可。
    - 其英文定义为：
      - Once an interface has gotten too 'fat' it needs to be split into smaller and more specific interfaces so that any clients of the interface will only know about the methods that pertain to them.

10个面向对象的设计原则

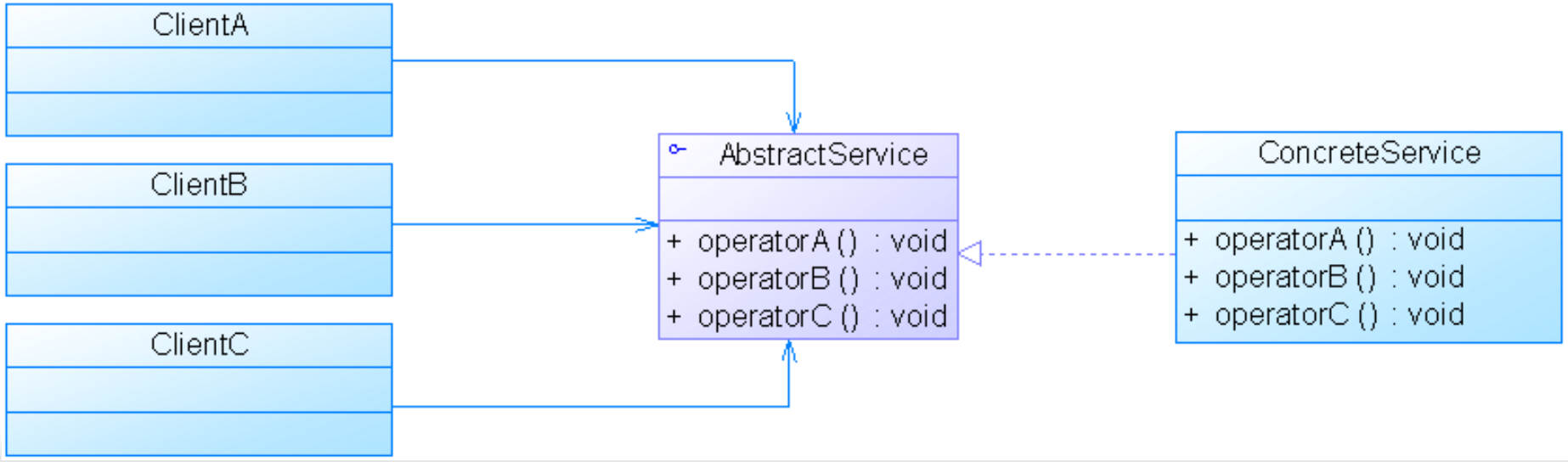
- 接口隔离原则分析
  - 接口隔离原则是指使用多个专门的接口，而不使用单一的总接口。每一个接口应该承担一种相对独立的角色，不多不少，不干不该干的事，该干的事都要干。
    - (1) 一个接口就只代表一个角色，每个角色都有它特定的一个接口，此时这个原则可以叫做“角色隔离原则”。
    - (2) 接口仅仅提供客户端需要的行为，即所需的方法，客户端不需要的行为则隐藏起来，应当为客户端提供尽可能小的单独的接口，而不要提供大的总接口。

10个面向对象的设计原则

- 接口隔离原则分析
  - 使用接口隔离原则拆分接口时，首先必须满足单一职责原则，将一组相关的操作定义在一个接口中，且在满足高内聚的前提下，接口中的方法越少越好。
  - 可以在进行系统设计时采用定制服务的方式，即为不同的客户端提供宽窄不同的接口，只提供用户需要的行为，而隐藏用户不需要的行为。

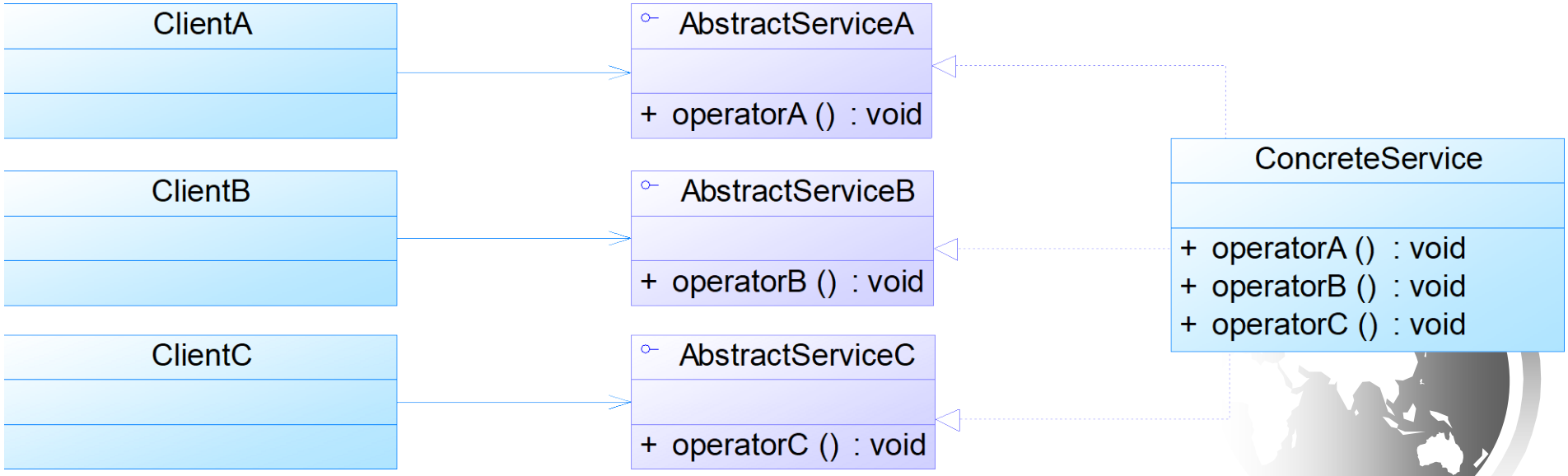
10个面向对象的设计原则

- 接口隔离原则实例
  - 实例说明
    - 下图展示了一个拥有多个客户类的系统，在系统中定义了一个巨大的接口（胖接口）AbstractService来服务所有的客户类。可以使用接口隔离原则对其进行重构。



10个面向对象的设计原则

- 接口隔离原则实例
  - 实例解析



10个面向对象的设计原则

- **原则9：针对接口编程，而不是针对实现编程**

  该原则可以使代码更加灵活，以便可以在任何接口实现中使用。因此，在Java中最好使用变量接口类型、方法返回类型、方法参数类型等。《Effective Java》 和《head first design pattern》书中也有提到。

10个面向对象的设计原则

- **原则10：委托原则**

  该原则最典型的例子是Java中的equals() 和 hashCode() 方法。为了平等地比较两个对象，我们用类本身而不是客户端类来做比较。这个设计原则的好处是没有重复的代码，而且很容易对其进行修改。