本次课程的目标

本次课程的主题是接口：将抽象数据类型中的实现与抽象接口分离开，并在Java中运用 `interface` 强制这种分离。

在这次课程后，你应该能够定义ADT的接口，并能够写出对应的实现类。

译者注：本次阅读少部分说法基于*Javase8*及以后的版本。参考：*Java 8 Interface Changes – static method, default method*

接口

Java中的 `interface` （接口）是一种表示抽象数据类型的好方法。接口中是一连串的方法标识，但是没有方法体（定义）。如果想要写一个类来实现接口，我们必须给类加上 `implements` 关键字，并且在类内部提供接口中方法的定义。所以接口+实现类也是Java中定义抽象数据类型的一种方法。

这种做法的一个优点就是接口只为使用者提供"契约"（contract），而使用者只需要读懂这个接口即可使用该ADT，他也不需要依赖ADT特定的实现/表示，因为实例化的变量不能放在接口中（具体实现被分离在另外的类中）。

接口的另一个优点就是它允许了一种抽象类型能够有多种实现/表示，即一个接口可以有多个实现类（译者注：一个类也可以同时实现多个接口）。而当一个类型只用一个类来实现时，我们很难改变它的内部表示。例如之前阅读中的 `MyString` 这个例子，我们对 `MyString` 实现了两种表示方法，但是这两个类就不能同时存在于一个程序中。

Java的静态检查会发现没有实现接口的错误，例如，如果程序员忘记实现接口中的某一个方法或者返回了一个错误的类型，编译器就会在编译期报错。不幸的是，编译器不会去检查我们的方法是否遵循了接口中的文档注释。

> 关于定义接口的细节，请参考 Java Tutorials section on interfaces.

阅读小练习

Java interfaces

思考下面这个Java接口和实现类，它们尝试实现一个不可变的集合类型：

```
    /** Represents an immutable set of elements of type E. */
    public interface Set<E> {
        /** make an empty set */
A       public Set();
        /** @return true if this set contains e as a member */
        public boolean contains(E e);
        /** @return a set which is the union of this and that */
B       public ArraySet<E> union(Set<E> that);
    }

    /** Implementation of Set<E>. */
    public class ArraySet<E> implements Set<E> {
        /** make an empty set */
        public ArraySet() { ... }
        /** @return a set which is the union of this and that */
        public ArraySet<E> union(Set<E> that) { ... }
        /** add e to this set */
        public void add(E e) { ... }
    }
```

下面关于 `Set<E>` 和 `ArraySet<E>` 的说法哪一个是正确的？

`A` 标号处有问题，因为接口不能有构造方法。--> True

The line labeled `B` is a problem because `Set` mentions `ArraySet`, but `ArraySet` also mentions `Set`, which is circular. --> False

`B` 标号处有问题，因为它没有实现"表示独立"。--> True

`ArraySet` 并没有正确实现 `Set`，因为它缺失了 `contains()` 方法。--> True

`ArraySet` doesn't correctly implement `Set` because it includes a method that `Set` doesn't have. --> False

`ArraySet` 并没有正确实现 `Set`，因为 `ArraySet` 是可变的，但是 `Set` 是不可变的。--> True

子类型

回忆一下，我们之前说过类型就是值的集合。Java中的 `List` 类型是通过接口定义的，如果我们想一下 `List` 所有的可能值，它们都不是 `List` 对象：我们不能通过接口实例化对象——这些值都是 `ArrayList` 对象, 或 `LinkedList` 对象，或者是其他 `List` 实现类的对象。我们说，一个子类型就是父类型的子集，正如 `ArrayList` 和 `LinkedList` 是 `List` 的子类型一样。

"B是A的子类型"就意味着"每一个B都是A"，换句话说，"每一个B都满足了A的规格说明"。

这也意味着B的规格说明至少强于A的规格说明。当我们声明一个接口的实现类时，编译器会尝试做这样的检查：它会检查类是否全部实现了接口中规定的函数，并且检查这些函数的标识是否对的上。

但是编译器不会检查我们是否通过其他形式弱化了规格说明：例如强化了某个方法输入的前置条件，或弱化了接口对于用户的保证（后置条件）。如果你在Java中定义了一个子类型——我们这里是实现接口——你必须要确保子类型的规格说明至少要比父类型强。

阅读小练习

Immutable shapes

让我们为矩形定义一个接口：

```
/** An immutable rectangle. */
public interface ImmutableRectangle {
    /** @return the width of this rectangle */
    public int getWidth();
    /** @return the height of this rectangle */
    public int getHeight();
}
```

而每一个正方形类型都是矩形类型：

```
/** An immutable square. */
public class ImmutableSquare {
    private final int side;
    /** Make a new side x side square. */
```

```
    public ImmutableSquare(int side) { this.side = side; }
    /** @return the width of this square */
    public int getWidth() { return side; }
    /** @return the height of this square */
    public int getHeight() { return side; }
}
```

`ImmutableSquare.getWidth()` 是否满足了 `ImmutableRectangle.getWidth()` 的规格说明? --> Yes

`ImmutableSquare.getHeight()` 是否满足了 `ImmutableRectangle.getHeight()` 的规格说明? -->Yes

`ImmutableSquare` 的规格说明是否满足了（至少强于） `ImmutableRectangle` 的规格说明? --> Yes

Mutable shapes

```
/** A mutable rectangle. */
public interface MutableRectangle {
    // ... same methods as above ...
    /** Set this rectangle's dimensions to width x height. */
    public void setSize(int width, int height);
}
```

现在每一个正方形类型还是矩形类型吗?

```
/** A mutable square. */
public class MutableSquare {
    private int side;
    // ... same constructor and methods as above ...
    // TODO implement setSize(..)
}
```

对于下面的每一个 `MutableSquare.setSize(..)` 实现，请判断它是否合理:

```
/** Set this square's dimensions to width x height.
 *  Requires width = height. */
public void setSize(int width, int height) { ... }
```

--> No – stronger precondition

```
/** Set this square's dimensions to width x height.
 *  @throws BadSizeException if width != height */
public void setSize(int width, int height) throws BadSizeException { ... }
```

--> Specifications are incomparable

```
/** If width = height, set this square's dimensions to width x height.
 *  Otherwise, new dimensions are unspecified. */
public void setSize(int width, int height) { ... }
```

--> No – weaker postcondition

```
/** Set this square's dimensions to side x side. */
public void setSize(int side) { ... }
```

--> Specifications are incomparable

例子：`MyString`

现在我们再来看一看 `MyString` 这个例子，这次我们使用接口来定义这个ADT，以便创建多种实现类：

```
/** MyString represents an immutable sequence of characters. */
public interface MyString {

    // We'll skip this creator operation for now
    // /** @param b a boolean value
    //  *  @return string representation of b, either "true" or "false" */
    // public static MyString valueOf(boolean b) { ... }

    /** @return number of characters in this string */
    public int length();

    /** @param i character position (requires 0 <= i < string length)
     *  @return character at position i */
    public char charAt(int i);

    /** Get the substring between start (inclusive) and end (exclusive).
     *  @param start starting index
     *  @param end ending index.  Requires 0 <= start <= end <= string length.
     *  @return string consisting of charAt(start)...charAt(end-1) */
    public MyString substring(int start, int end);
}
```

现在我们先跳过 `valueOf` 这个方法，用我们在"抽象数据类型"中学习到的知识去实现这个接口。

下面是我们的第一种实现类：

```
public class SimpleMyString implements MyString {

    private char[] a;

    /** Create a string representation of b, either "true" or "false".
     *  @param b a boolean value */
    public SimpleMyString(boolean b) {
        a = b ? new char[] { 't', 'r', 'u', 'e' }
              : new char[] { 'f', 'a', 'l', 's', 'e' };
    }

    // private constructor, used internally by producer operations
    private SimpleMyString(char[] a) {
        this.a = a;
    }

    @Override public int length() { return a.length; }

    @Override public char charAt(int i) { return a[i]; }

    @Override public MyString substring(int start, int end) {
        char[] subArray = new char[end - start];
```

```
        System.arraycopy(this.a, start, subArray, 0, end - start);
        return new SimpleMyString(subArray);
    }
}
```

而下面是我们优化过的实现类：

```
public class FastMyString implements MyString {

    private char[] a;
    private int start;
    private int end;

    /** Create a string representation of b, either "true" or "false".
     *  @param b a boolean value */
    public FastMyString(boolean b) {
        a = b ? new char[] { 't', 'r', 'u', 'e' }
              : new char[] { 'f', 'a', 'l', 's', 'e' };
        start = 0;
        end = a.length;
    }

    // private constructor, used internally by producer operations.
    private FastMyString(char[] a, int start, int end) {
        this.a = a;
        this.start = start;
        this.end = end;
    }

    @Override public int length() { return end - start; }

    @Override public char charAt(int i) { return a[start + i]; }

    @Override public MyString substring(int start, int end) {
        return new FastMyString(this.a, this.start + start, this.end + end);
    }
}
```

- 与我们之前的实现相比，注意到之前的代码中 `valueOf` 是静态方法，但是在这里就不是了。而这里也使用了指向实例内部表示的 `this` 。
- 同时要注意到 `@Override` 的使用，这个词是通知编译器这个方法必须和其父类中的某个方法的标识完全一样（覆盖）。但是由于实现接口时编译器会自动检查我们的实现方法是否遵循了接口中的方法标识，这里的 `@Override` 更多是一种文档注释，它告诉读者这里的方法是为了实现某个接口，读者应该去阅读这个接口中的规格说明。同时，如果你没有对实现类（子类型）的规格说明进行强化，这里就不需要再写一遍规格说明了。（DRY原则）
- 另外注意到我们添加了一个私有的构造方法，它是为 `substring(..)` 这样的生产者服务的。它的参数是表示的域。我们之前并不需要写出构造方法，因为Java会在没有构造方法时自动构建一个空的构造方法，但是这里我们添加了一个接收 `boolean b` 的构造方法，所以就必须显式声明另一个为生产者服务的构造方法了。

那么使用者会如何用这个ADT呢？下面是一个例子：

```
MyString s = new FastMyString(true);
System.out.println("The first character is: " + s.charAt(0));
```

这似乎和我们用Java的聚合类型时的代码很像，例如：

```
List<String> s = new ArrayList<String>();
...
```

不幸的是，这种模式已经破坏了我们辛苦构建的抽象层次 。使用者必须知道具体实现类的名字。因为Java接口中不能包含构造方法，它们必须通过调用实现类的构造方法来获取接口类型的对象，而接口中是不可能含有构造方法的规格说明的。另外，由于接口中没有对构造方法进行说明，所以我们甚至无法保证不同的实现类会提供同样的构造方法。

幸运的是，Java8以后允许为接口定义静态方法，所以我们可以在接口 `MyString` 中通过静态的工厂方法来实现创建者 `valueOf` ：

```
public interface MyString {

    /** @param b a boolean value
     *  @return string representation of b, either "true" or "false" */
    public static MyString valueOf(boolean b) {
        return new FastMyString(true);
    }

    // ...
```

现在使用者可以在不破坏抽象层次的前提下使用ADT了：

```
MyString s = MyString.valueOf(true);
System.out.println("The first character is: " + s.charAt(0));
```

将实现完全英寸起来是一种"妥协"，因为有时候使用者会希望有对具体实现的选择权利。这也是为什么Java库中的 `ArrayList` 和 `LinkedList` "暴露"给了用户，因为这两个实现在 `get()` 和 `insert()` 这样的操作中会有性能上的差别。

阅读小练习

Code review

现在让我们来审查以下 `FastMyString` 实现，下面是对这个实现的一些批评，你认为哪一些是对的？

应该把抽象函数注释出来 --> True

应该把表示不变量注释出来 --> True

表示域应该使用关键词 `final` 以便它们不能被重新改变索引 --> True

The private constructor should be public so clients can use it to construct their own arbitrary strings --> False

The `charAt` specification should not expose that the rep contains individual characters --> False

`charAt` 应该对于大于字符串长度的 `i` 有更好的处理 --> True

例子: 泛型 `Set<E>`

Java中的聚合类型为"将接口和实现分离"提供了很好的例子。

现在我们来思考一下java聚合类型中的 `Set` 。 `Set` 是一个用来表示有着有限元素 `E` 的集合。这里是 `Set` 的一个简化的接口:

```
/** A mutable set.
 *  @param <E> type of elements in the set */
public interface Set<E> {
```

`Set` 是一个泛型类型（generic type）：这种类型的规格说明中用一个占位符（以后会被作为参数输入）表示具体类型，而不是分开为不同类型例如 `Set<String>` , `Set<Integer>` , 进行说明。我们只需要设计实现一个 `Set<E>` .

现在我们分别实现/声明这个ADT的各个操作，从创建者开始:

```
    // example creator operation
    /** Make an empty set.
     *  @param <E> type of elements in the set
     *  @return a new set instance, initially empty */
    public static <E> Set<E> make() { ... }
```

这里的 `make` 是作为一个静态工厂方法实现的。使用者会像这样调用它: `Set<String> strings = Set.make();` ，而编译器也会知道新的 `Set` 会是一个包含 `String` 对象元素的集合。（注意我们将 `<E>` 写在函数标识前面，因为 `make` 是一个静态方法，而 `<E>` 是它的泛型类型）。

```
    // example observer operations

    /** Get size of the set.
     *  @return the number of elements in this set */
    public int size();

    /** Test for membership.
     *  @param e an element
     *  @return true iff this set contains e */
    public boolean contains(E e);
```

接下来我们声明两个观察者。注意到规格说明中的提示，这里不应该提到具体某一个实现的细节或者它们的标识，而规格说明也应该适用于所有 `Set` ADT的实现。

```
    // example mutator operations

    /** Modifies this set by adding e to the set.
```

```
     *  @param e element to add */
    public void add(E e);

    /** Modifies this set by removing e, if found.
     *  If e is not found in the set, has no effect.
     *  @param e element to remove */
    public void remove(E e);
```

对于改造者的要求也和观察者一样，我们依然要在接口抽象的层次书写规格说明。

阅读参考：

- Lesson: Interfaces
- The Set Interface
- Set Implementations
- The List Interface
- List Implementations

阅读小练习

Collection interfaces & implementations

假设下面的代码都是逐次执行的，并且不能被编译的代码都会被注释掉。

这里的代码使用到了 `Collections` 中的两个方法，你可能需要阅读一些参考。请为下面的问题回答出最合理的答案。

```
Set<String> set = new HashSet<String>();
```

`set` 现在指向： --> 一个HashSet对象

```
set = Collections.unmodifiableSet(set);
```

`set` 现在指向： --> 一个实现了 `Set` 接口的对象

```
set = Collections.singleton("glorp");
```

`set` 现在指向： --> 一个实现了 `Set` 接口的对象

```
set = new Set<String>();
```

`set` 现在指向： --> 这一行不能被编译

```
List<String> list = set;
```

`set` 现在指向： --> 这一行不能被编译

泛型接口的实现

假设现在我们要实现上面的 `Set<E>` 接口。我们既可以使用一个非泛型的实现（用一个特定的类型替代 `E` ），也可以使用一个泛型实现（保留类型占位符）。

首先我们来看看泛型接口的非泛型实现：

在 抽象函数 & 表示不变量 我们实现了 `CharSet` 类型,它被用来表示字符的集合。其中 `CharSet1` / `2` / `3` 这三种实现类都是 `Set` 接口 的子类型，它们的声明如下：

```
public class CharSet implements Set<Character>
```

当在Set声明中提到 `E` 时， `Charset` 的实现将类型占位符 `E` 替换为了 `Character` ：

```
public interface Set<E> {

    // ...

    /**
     * Test for membership.
     * @param e an element
     * @return true iff this set contains e
     */
    public boolean contains(E e);

    /**
     * Modifies this set by adding e to the set.
     * @param e element to add
     */
    public void add(E e);

    // ...
}
```

```
public class CharSet1 implements Set<Character> {

    private String s = "";


    // ...


    @Override
    public boolean contains(Character e) {
        checkRep();
        return s.indexOf(e) != -1;
    }

    @Override
    public void add(Character e) {
        if (!contains(e)) s += e;
        checkRep();
    }
    // ...
```

```
}
```

`CharSet1` / `2` / `3` 的实现方法不适用于任意类型的元素，例如，由于它使用的是 `String` 成员，`Set<Integer>` 这种集合就无法直接表示。

接着我们再来看看泛型接口的泛型实现：

我们也可以在实现 `Set<E>` 接口的时候不对 `E` 选择一个特定的类型。在这种情况下，我们会让使用者决定 `E` 到底是什么。例如，Java的 `HashSet` 就是这种实现，它的声明像这样：

```
public interface Set<E> {

    // ...
```

```
public class HashSet<E> implements Set<E> {

    // ...
```

一个泛型实现只能依靠接口规格说明中对类型占位符的要求，我们会在以后的阅读中看到 `HashSet` 是如何依靠每一个类型都要求实现的操作来实现它自己的，因为它没办法依赖于特定类型的操作。

为什么要使用接口?

在Java代码中，接口被用的很广泛（但也不是所有类都是接口的实现），这里列出来了几个使用接口的好处：

- 接口对于编译器和读者来说都是重要的文档：接口不仅会帮助编译器发现ADT实现过程中的错误，它也会帮助读者更容易/快速的理解ADT的操作——因为接口将ADT抽象到了更高的层次，用户不需要关心具体实现的各种方案。
- 允许进行性能上的权衡：接口使得ADT可以有不同的实现方案，而这些实现方案可能在不同环境下的性能或其他资源特性有很大差别。使用者可以根据自己的环境/需求选择合适的实现方案。但是，在我们选择特定的方案后，我们依旧要保持代码的表示独立性，即当ADT发生（内部）改变或更换实现方案后代码依然能正常运行。
- 通过未决定的规格说明给实现者以定义方法的自由：例如，当把一个有限集合转化为一个列表的时候，有一些实现可能是使用较慢的方法，但是它们确保这些元素在列表中是排好序的；而其他的实现可能是不管这些元素转换后在列表中的排序，但是它们的速度更快。
- 一个类具有多种"视角"：在Java中，一个类可以同时实现多个接口，例如，一个能够显示列表的窗口部件就可能是一个同时实现了窗口和列表这两个接口的类。这反映的是多种ADT特性同时存在的特殊情况。
- 允许不同信任度的实现：另一个多次实现一个接口的原因在于，你可以写一个简单但是非常可靠的实现，也可以写一个很"炫"但是bug存在的几率（稳定性）高一些的实现。而使用者可以根据

实际情况选择相应的方案。

阅读小练习

假设你有一个有理数的类型，它现在是以类来表示的：

```
public class Rational {
    ...
}
```

现在你决定将 `Rational` 换成Java接口，同时定义了一个实现类 `IntFraction` ：

```
public interface Rational {
    ...
}

public class IntFraction implements Rational {
    ...
}
```

对于下面之前 `Rational` 类中的代码，请你判定它们对应的身份，以及应该出现在新的接口或者新的实现类中？

Interface + implementation 1

```
private int numerator;
private int denominator;
```

这段代码是（选中所有正确答案）：

- [ ] 抽象函数
- [ ] 创建者
- [ ] 改造者
- [ ] 观察者
- [ ] 生产者
- [x] （成员）表示
- [ ] 表示不变量
- [ ] 规格说明

它应该位于：

- [ ] 接口
- [x] 实现类

- [ ] 都有

Interface + implementation 2

```
//   denominator > 0
//   numerator/denominator is in reduced form
```

这段代码是（选中所有正确答案）：

- [ ] 抽象函数
- [ ] 创建者
- [ ] 改造者
- [ ] 观察者
- [ ] 生产者
- [ ] （成员）表示
- [x] 表示不变量
- [ ] 规格说明

它应该位于：

- [ ] 接口
- [x] 实现类
- [ ] 都有

Interface + implementation 3

```
//   AF(numerator, denominator) = numerator / denominator
```

这段代码是（选中所有正确答案）：

- [x] 抽象函数
- [ ] 创建者
- [ ] 改造者
- [ ] 观察者
- [ ] 生产者
- [ ] （成员）表示

- [ ] 表示不变量
- [ ] 规格说明

它应该位于：

- [ ] 接口
- [x] 实现类
- [ ] 都有

Interface + implementation 4

```
    /**
     * @param that another Rational
     * @return a Rational equal to (this / that)
     */
```

这段代码是（选中所有正确答案）：

- [ ] 抽象函数
- [ ] 创建者
- [ ] 改造者
- [ ] 观察者
- [x] 生产者
- [ ] （成员）表示
- [ ] 表示不变量
- [ ] 规格说明

它应该位于：

- [x] 接口
- [ ] 实现类
- [ ] 都有

Interface + implementation 5

```
    public boolean isZero()
```

这段代码是（选中所有正确答案）：

- [ ] 抽象函数
- [ ] 创建者
- [ ] 改造者
- [x] 观察者
- [ ] 生产者
- [ ] （成员）表示
- [ ] 表示不变量
- [ ] 规格说明

它应该位于：

- [ ] 接口
- [ ] 实现类
- [x] 都有

Interface + implementation 6

```
    return numer == 0;
```

这段代码是（选中所有正确答案）：

- [ ] 抽象函数
- [ ] 创建者
- [ ] 改造者
- [x] 观察者
- [ ] 生产者
- [ ] （成员）表示
- [ ] 表示不变量
- [ ] 规格说明

它应该位于：

- [ ] 接口
- [x] 实现类
- [ ] 都有

枚举

有时候一个ADT的值域是一个很小的有限集，例如：

- 一年中的月份: January, February, …
- 一周中的天数: Monday, Tuesday, …
- 方向: north, south, east, west
- 画线时的line caps : butt, round, square

这样的类型往往会被用来组成更复杂的类型（例如 `DateTime` 或者 `Latitude` ），或者作为一个改某个方法的行为的参数使用（例如 `drawline` ）。

当值域很小且有限时，将所有的值定义为被命名的常量是有意义的，这被称为枚举(**enumeration**)。JAVA用 `enum` 使得枚举变得方便：

```
public enum Month { JANUARY, FEBRUARY, MARCH, ..., DECEMBER };
```

这个 `enum` 定义类一种新的类型名， `Month` ，这和使用 `class` 以及 `interface` 定义新类型名时是一样的。它也定义了一个被命名的值的集合，由于这些值实际上是 `public static final` ,所以我们将这个集合中的每个值的每个字母都大写。所以你可以这么写：

```
Month thisMonth = MARCH;
```

这种思想被称为枚举，因为你显式地列出了一个集合中的所有元素，并且JAVA为每个元素都分配了数字作为代表它们的值。

在枚举类型最简单的使用场景中，你需要的唯一操作是比较两个值是否相等：

```
if (day.equals(SATURDAY) || day.equals(SUNDAY)) {
    System.out.println("It's the weekend");
}
```

你可能也会看到这样的代码，它使用 `==` 而不是 `equals()` ：

```
if (day == SATURDAY || day == SUNDAY) {
    System.out.println("It's the weekend");
}
```

如果使用 `String` 类型来表示天数，那么这个代码是不安全的，因为 `==` 检测两边的表达式是否引

用的是同一个对象，对于任意的两个字符串 `"Saturday"` 来说，这是不一定的。这也是为什么我们总是在比较两个对象时使用 `equals()` 的原因。但是使用枚举类型的好处之一就是：实际上只有一个对象来表示枚举类型的每个取值，且用户不可能创建更多的对象（没有构造者方法！）所以对于枚举类型来说， `==` 和 `equals()` 的效果是一样的。

在这个意义上，使用枚举就像使用原式的 `int` 常量一样。JAVA甚至支持在 `switch` 语句中使用枚举类型（ `switch` 在其他情况下只允许使用原式的整型，而不能是对象）：

```
switch (direction) {
    case NORTH: return "polar bears";
    case SOUTH: return "penguins";
    case EAST:  return "elephants";
    case WEST:  return "llamas";
}
```

但是和 `int` 值不同的是，JAVA对枚举类型有更多的静态检查:

```
Month firstMonth = MONDAY; // static error: MONDAY has type DayOfWeek, not type Month
```

一个 `enum` 声明中可以包含所有能在 `class` 声明中常用字段和方法。所以你可以为这个ADT定义额外的操作，并且还定义你自己的表示（成员变量）。这里是一个声明了一个成员变量、一个观察者和一个生产者的枚举类型的例子：

```
public enum Month {
    // the values of the enumeration, written as calls to the private constructor below
    JANUARY(31),
    FEBRUARY(28),
    MARCH(31),
    APRIL(30),
    MAY(31),
    JUNE(30),
    JULY(31),
    AUGUST(31),
    SEPTEMBER(30),
    OCTOBER(31),
    NOVEMBER(30),
    DECEMBER(31);

    // rep
    private final int daysInMonth;

    // enums also have an automatic, invisible rep field:
    //   private final int ordinal;
    // which takes on values 0, 1, ... for each value in the enumeration.

    // rep invariant:
    //   daysInMonth is the number of days in this month in a non-leap year
    // abstraction function:
    //   AF(ordinal,daysInMonth) = the (ordinal+1)th month of the Gregorian calendar
    // safety from rep exposure:
    //   all fields are private, final, and have immutable types

    // Make a Month value. Not visible to clients, only used to initialize the
```

```
    // constants above.
    private Month(int daysInMonth) {
        this.daysInMonth = daysInMonth;
    }

    /**
     * @param isLeapYear true iff the year under consideration is a leap year
     * @return number of days in this month in a normal year (if !isLeapYear)
     *                                          or leap year (if isLeapYear)
     */
    public int getDaysInMonth(boolean isLeapYear) {
        if (this == FEBRUARY && isLeapYear) {
            return daysInMonth+1;
        } else {
            return daysInMonth;
        }
    }

    /**
     * @return first month of the semester after this month
     */
    public Month nextSemester() {
        switch (this) {
            case JANUARY:
                return FEBRUARY;
            case FEBRUARY:   // cases with no break or return
            case MARCH:      // fall through to the next case
            case APRIL:
            case MAY:
                return JUNE;
            case JUNE:
            case JULY:
            case AUGUST:
                return SEPTEMBER;
            case SEPTEMBER:
            case OCTOBER:
            case NOVEMBER:
            case DECEMBER:
                return JANUARY;
            default:
                throw new RuntimeException("can't get here");
        }
    }
}
```

所有的 `enum` 类型也都有一些内置的(automatically-provided)操作，这些操作在 `Enum` 中定义：

- `ordinal()` 是某个值在枚举类型中的索引值，因此 `JANUARY.ordinal()` 返回 0.
- `compareTo()` 基于两个值的索引值来比较两个值.
- `name()` 返回字符串形式表示的当前枚举类型值，例如， `JANUARY.name()` 返回 `"JANUARY"` .
- `toString()` 和 `name()` 是一样的.

> 阅读JAVA教程中的**Enum Types** （1页）和 **Nested Classes**（1页）

阅读测试

Semester

考虑这三种可选的方式来命名你将要注册的Semester：

- 用一个字符串字面量:

```
startRegistrationFor("Fall", 2023);
```

- 用一个命名的 `String` 类型常量:

```
public static final String FALL = "Fall";
...
startRegistrationFor(FALL, 2023);
```

- 用一个枚举类型的值：

```
public enum Semester { IAP, SPRING, SUMMER, FALL };
...
startRegistrationFor(FALL, 2023);
```

下列关于每个方案的优缺点叙述正确的是：

- [x] 使用字符串字面量的方案不会快速报错，因为用户可能拼写错误的学期，而不会得到这样的静态错误信息: `startRegistrationFor("FAll", 2023)`
- [ ] The named string constant approach isn't safe from bugs, because the name can be reassigned: `FALL = "Spring"`
- [x] 命名的字符串常量方案不会快速报错，因为用户可能直接用不正确的字符串字面量来调用，但是却不会得到静态错误： `startRegistrationFor("Autumn", 2023)` .
- [ ] The enumeration approach isn't safe from bugs, because the client can define new semesters without getting a static error: `startRegistrationFor(new Semester("Autumn"), 2023)` .
- [ ] The enumeration approach isn't safe from bugs, because the client can substitute a different enumeration type without getting a static error: `startRegistrationFor(JANUARY, 2023)` .

抽象数据类型在Java中的实现

现在我们完成了对"抽象数据类型"中"Java中ADT实现"的理解：

ADT 角度	**Java**实现	例子	

抽象数据类型	类	`String`
	接口 + 类	`List` and `ArrayList`
	枚举（Enum）	`DayOfWeek`
创建者操作	构造方法	`ArrayList()`
	静态（工厂）方法	`Collections.singletonList()` , `Arrays.asList()`
	常量	`BigInteger.ZERO`
观察者操作	实例方法	`List.get()`
	静态方法	`Collections.max()`
生产者操作	实例方法	`String.trim()`
	静态方法	`Collections.unmodifiableList()`
改造者操作	实例方法	`List.add()`
	静态方法	`Collections.copy()`
（成员）表示	`private` /私有域	

总结

抽象数据类型是由它支持的操作集合所定义的，而Java中的结构能够帮助我们形式化这种思想。

这能够使我们的代码：

- 远离**bug.** 一个ADT是由它的操作集合定义的，而接口就是做了这件事情。当使用者使用接口类型时，静态检查能够确保它们只使用了接口规定的方法。如果实现类写出了/暴露了其他方法——或者更糟糕，暴露了内部表示——，使用者也不会依赖于这些操作。当我们实现一个接口时，编译器会确保所有的方法标识都得到实现。
- 易于理解**.** 使用者和维护者都知道在哪里寻找ADT的规格说明。因为接口没有实例成员或者实例方法的函数体，所以它能更容易的将具体实现从规格说明中分离开。
- 可改动**.** 我们可以轻松地为已有的接口添加新的实现类。如果我们认为静态工厂方法比类构造方法更合适，使用者将只会看到这个接口。这意味着我们可以调整接口中工厂方法的实现类而不用

改变使用者的代码。

Java的枚举类型能够定义一种只有少部分不可变值的ADT。和以前使用特殊的整数或者字符串相比，枚举类型能够帮助我们的代码：

- 远离**bug.** 静态检查能够确保使用者没有使用到规定集合外的值，或者是不同枚举类型的值。
- 易于理解**.** 将常量命名为枚举类型名字而非幻数（或其他字面量）能够更清晰的做自我注释。
- 可改动**.** 无