3.4 Oriented Programming (OOP)

一级标题，二级标题，三级标题，四级标题，注，拓展，不懂

MIT 6.031：12

Safe from bugs	Easy to understand	Ready for change
Correct today and correct in the unknown future.	Communicating clearly with future programmers, including future you.	Designed to accommodate change without rewriting.

3.4.1 object, class, fields, method, package

Object（对象）：

Object 表示现实中的某个对象。

数据成员（fields）→ 现实事物的属性（states）.

方法成员（methods） → 现实事物的行为（behaviors）。

Class（类）：

类的定义：

类是一组对象的抽象,同样定义了 fields 和 methods。

一个类在形式上分为 type（类的名称）和 implementation（类的实现）。type 表明哪一个对象是这个类，implementation 表面以这个类定义的对象会具有什么样的属性。

类的声明:

Fields（数据成员）:

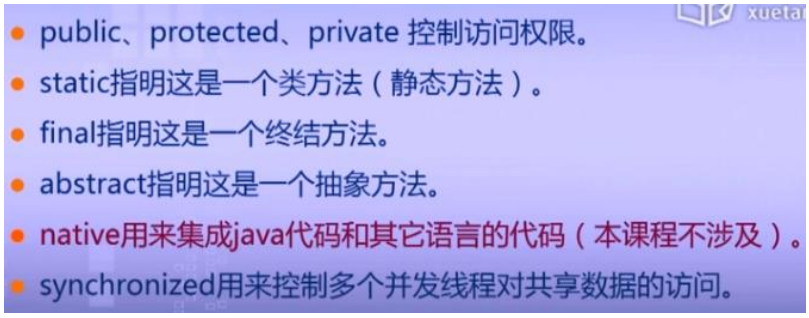
实例变量:

存在于类初始化的实例之中。

类变量:

类变量在类被初始化就被赋值, 该变量只在类中储存, 在该类创建的对象中并没有储存。

数据成员的声明:

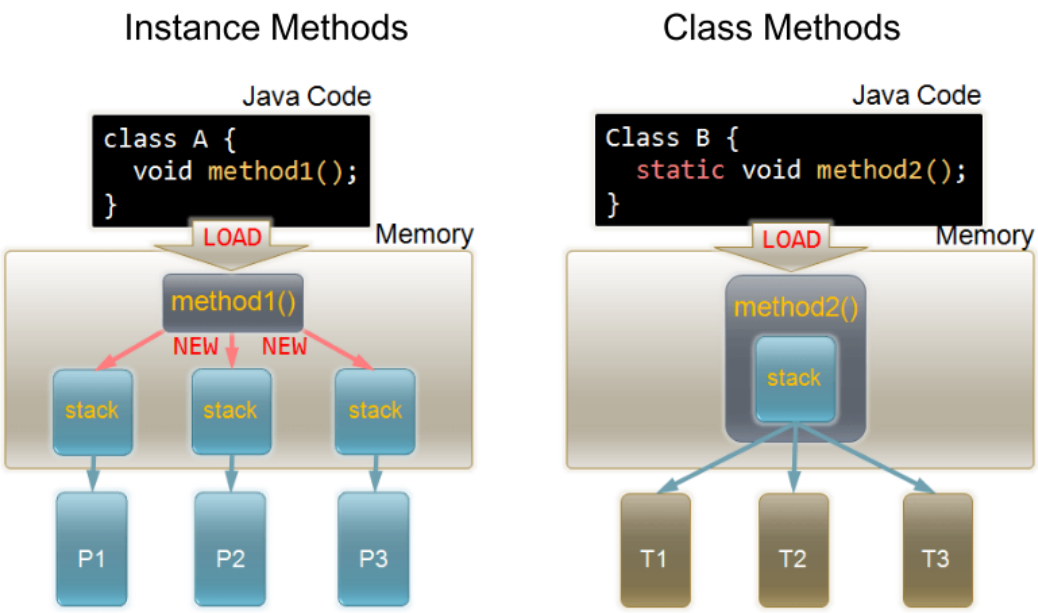


Methods（方法）:

class methods 类方法:

由这个类可以直接调用的方法。

在该方法内部, 不可以直接调用该类非静态数据成员, 因为非静态数据成员还未初始化。



instance methods 实例方法：

必须创建一个类的实例对象，才可以调用的方法。

方法的声明：

Package(包)：

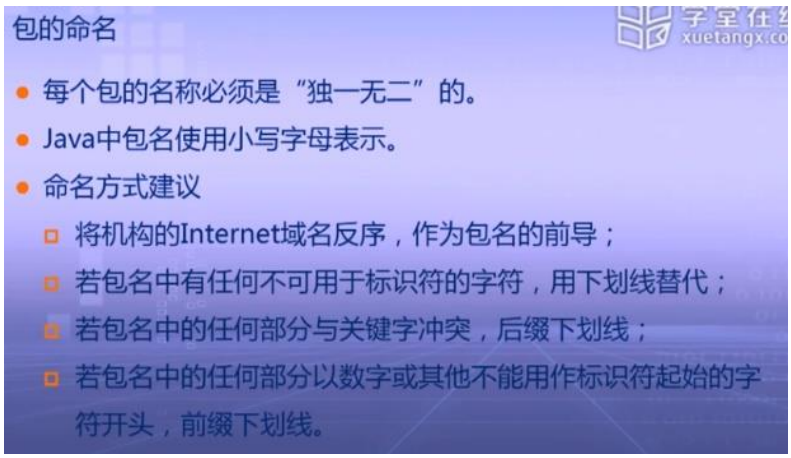
Definity：

包：是一组类的集合。

Characteristic：

可以管理类，利用包来划分命名空间避免类名冲突。也可以封装。

注：包的命名



3.4.2 interface（接口）

Definition:

An interface in Java is a list of method signatures without method bodies。Java 中的接口由一系列的没有具体实现，仅有方法签名的方法构成。

Implementation:

A class implements an interface if it declares the interface in its implements clause, and provides method bodies for all of the interface's methods. 一个类实现接口，必须给类加上 implements 关键字，并且在类内部提供接口中方法的具体实现。

Characteristic:

A. An interface can extend one or more others（接口之间可以继承与扩展）。
B. A class can implement multiple interfaces（一个类可以实现多个接口）。
C. 一个接口可以有多个实现类。

Advantage:

A. 隔离了 implementer 和 client

interface specifies the contract for the client and nothing more. The client can't create inadvertent dependencies on the ADT's rep, because fields can't be put in an interface at all. The implementation is kept well and truly separated, in a different class altogether.接口只为使用者提供“契约”（contract），client 不会对该 ADT 的内部实现产生依赖，因为内部实现值完全没有放在接口中，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）

B. 可以灵活的选用一个接口的多种实现方式

注: 接口的定义及性质使得其成为实现一个 ADT 的较好的选择，其定义中只有方法的签名，这一方面隔离了 implementer 和 client。一方面由方法定义的 ADT 的思想不谋而合。

注: 接口中不允许存在构造器，但是客户端需要知道该接口的某个具体实现类的名字，这就在一定程度上减弱了隔离，所以在 java6 之后，允许接口中定义某个静态方法进行构造初始化。

注: 接口与抽象类的不同是：抽象类只有一个抽象方法即可，但接口必须全部为抽象方法，但自 java6 开始，接口可以定义某些静态方法。另一个不同是：一个类只可以继承一个类，但可以实现多个接口。

Ex:

```java
/** 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);
}
```

```java
public class SimpleMyString implements MyString {

    private char[] a;

    /* Create an uninitialized SimpleMyString. */
    private SimpleMyString() {}

    /** 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' };
    }

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

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

    @Override public MyString substring(int start, int end) {
        SimpleMyString that = new SimpleMyString();
        that.a = new char[end - start];
        System.arraycopy(this.a, start, that.a, 0, end - start);
        return that;
    }
}
```

```java
public class FastMyString implements MyString {

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

    /* Create an uninitialized FastMyString. */
    private FastMyString() {}

    /** 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;
    }

    @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) {
        FastMyString that = new FastMyString();
        that.a = this.a;
        that.start = this.start + start;
        that.end = this.start + end;
        return that;
    }
}
```

3.4.2 Enumerations（枚举类型）

Definition

枚举类型应用于存在小部分的值，且该部分值被反复利用，将之整合为一个类型较为方便。

A. a new type name 类型名称
B. a set of named values which we write in all-caps because they are effectively public static final constants. 所圈定的枚举值，大写。
C. additional operations and fields. 额外的方法以及数据成员。

Ex:

```
public enum Month {
    // the values of the enumeration, written as calls to the private co
nstructor 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 enumeratio
n.

    // rep invariant:
    //   daysInMonth is the number of days in this month in a non-leap y
ear
```

```
    // abstraction function:
    //   AF(ordinal,daysInMonth) = the (ordinal+1)th month of the Gregor
ian 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 initiali
ze 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 !isLeap
Year)
     *                                          or leap year (if isLeap
Year)
     */
    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 startOfNextSemester() {
        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");
        }
    }
}
```

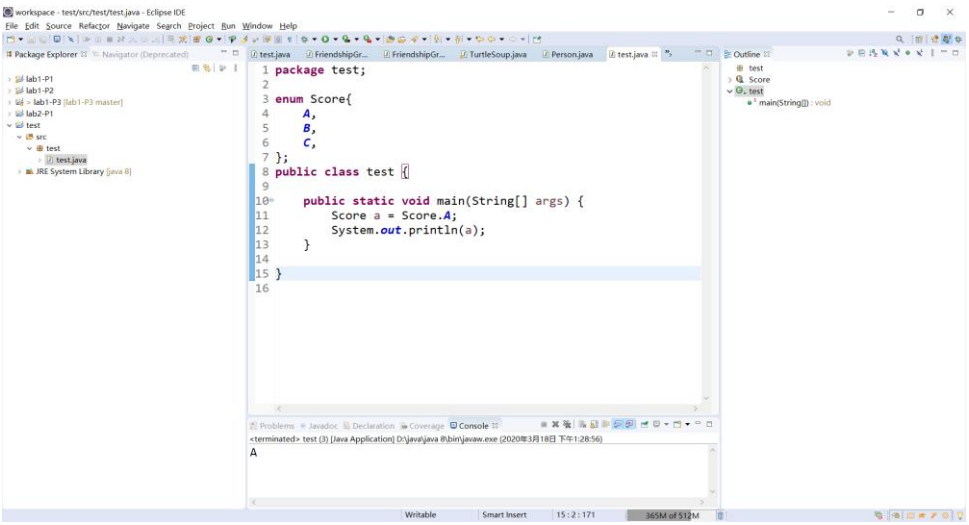
Ex:

```
public enum Planet {
   MERCURY (3.303e+23, 2.4397e6),
   VENUS   (4.869e+24, 6.0518e6),
   EARTH   (5.976e+24, 6.37814e6),
   MARS    (6.421e+23, 3.3972e6),
   JUPITER (1.9e+27,   7.1492e7),
   SATURN  (5.688e+26, 6.0268e7),
   URANUS  (8.686e+25, 2.5559e7),
   NEPTUNE (1.024e+26, 2.4746e7);

   private final double mass;   // in kilograms
   private final double radius; // in meters
   Planet(double mass, double radius) {
      this.mass = mass;
      this.radius = radius;
   }
```

```
   private double mass() { return mass; }
   private double radius() { return radius; }

   // universal gravitational constant  (m3 kg-1 s-2)
   public static final double G = 6.67300E-11;

   double surfaceGravity() {
      return G * mass / (radius * radius);
   }
   double surfaceWeight(double otherMass) {
      return otherMass * surfaceGravity();
   }
```

```
   public static void main(String[] args) {
     if (args.length != 1) {
       System.err.println("Usage: java Planet <earth_weight>");
        System.exit(-1);
     }
     double earthWeight = Double.parseDouble(args[0]);
     double mass = earthWeight/EARTH.surfaceGravity();
     for (Planet p : Planet.values())
       System.out.printf("Your weight on %s is %f%n",
                         p, p.surfaceWeight(mass));
   }
}
```

Ex:



注：在某种程度上，枚举值十分的类似于基本数据类型，一个枚举值只会在内存中创建一个内存空间，所有的引用都会指向该内存空间。故而可以利用 equals 和==进行直接的比较。

注：枚举类型都隐含继承自 java.lang.Enum，因此枚举类型不能再继承其他任何类。

注：枚举类型的构造方法必须是包内私有的或者私有的，定义在枚举开头的常量会被自动创建，不能显式地调用枚举类的构造方法

注：枚举类型支持 switch 语句进行判断。

Ex:

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

注：枚举类型常见的操作

All `enum` types also have some automatically-provided operations, defined by `Enum`:

- `ordinal()` is the index of the value in the enumeration, so `JANUARY.ordinal()` returns 0.
- `compareTo()` compares two values based on their ordinal numbers.
- `name()` returns the name of the value's constant as a string, e.g. `JANUARY.name()` returns `"JANUARY"`.
- `toString()` has the same behavior as `name()`.

拓展：访问权限

作用域	当前类	同一包内	子孙类(不同包)	其他包
public	√	√	√	√
protected	√	√	√	×
default	√	√	×	×
private	√	×	×	×

拓展：final 前缀于不同类型的含义

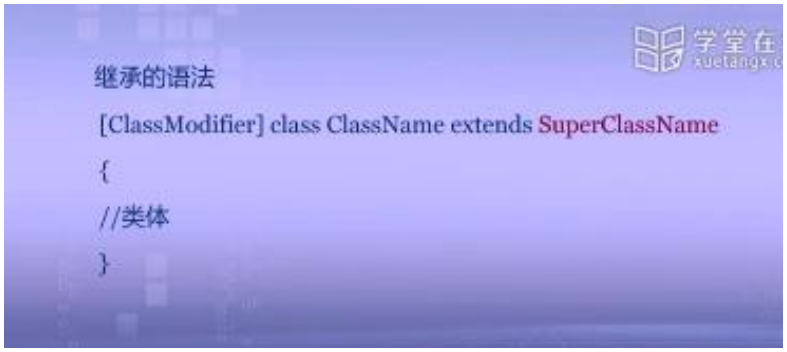
1. A final field: prevents reassignment to the field after initialization.初始化后该变量的值不可以被修改
2. A final method: prevents overriding the method.该方法不可以被重写
3. A final class: prevents extending the class.

3.4.3 Inheritance(继承)

Definition

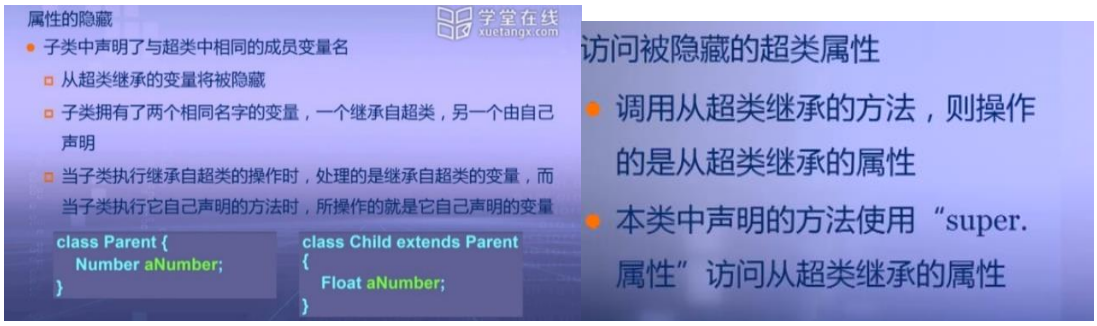
根据已有类来定义新类，新类拥有已有类的所有功能

声明



注：超类是所有子类的公共属性与方法的集合，而子类是超类的特殊化。

拓展：继承时属性的隐藏



3.4.4 Overriding (覆盖/重写)

Definition

Method overriding is a language feature that allows a subclass or child class to provide a specific implementation of a method that is already provided by one of its superclasses or parent classes.方法的覆盖是对超类中的方法进行重写，二者使用同样的签名。

Ex :

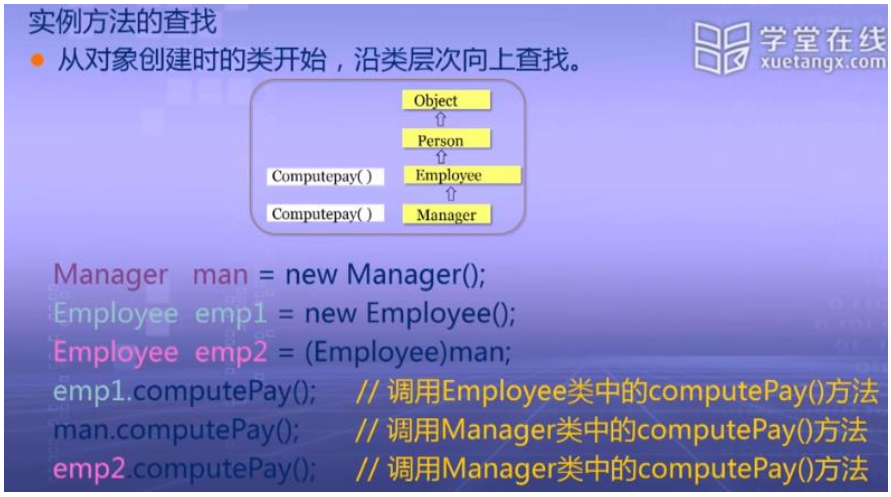
```
class Thought {
    public void message() {
        System.out.println("Thought.");
    }
}

public class Advice extends Thought {
    @Override  // @Override annotation in Java 5 is optional but helpful.
    public void message() {
        System.out.println("Advice.");
        super.message();  // Invoke parent's version of method.
    }
}

Thought parking = new Thought();
parking.message();  // Prints "Thought."

Thought dates = new Advice();
dates.message();  // Prints “Advice. \n Thought."
```

注：方法的重写尽量保证与原意一致，当然也可以不保证，但是，秉着科学编程的原则，尽量保证一致性，如果需要编写不一致的算法，可以重新定义一个方法。

注：重写的方法的访问权限只能更大，不可以更小。

拓展：方法的查找

对于方法覆盖而言，其对于方法的查找在运行阶段进行。



3.4.5 abstract （抽象）

Abstract methods（抽象方法）

Definition

A method with a signature but without an implementation (also called abstract operation)

只有定义没有实现

声明

声明的语法形式为：

public abstract <returnType> <methodName>(...);

Abstract class（抽象类）

Definition

A class which contains at least one abstract method is called abstract class.至少含有一个抽象方法的类。

声明

类名前加“abstract”修饰。

注：一个抽象类中可以包含非抽象方法。

注：抽象类不可进行实例化，只有当一个子类完全实现了抽象类之中的方法，才可以进行实例化

拓展：抽象类与接口的差异

抽象类只有一个抽象方法就可以，而接口必须所有方法都是抽象方法，但是从 java8 开始，接口支持非抽象的方法，这时抽象类和接口的区别不是很大。但是仍存在一个巨大的的差别是一个类只能继承一个抽象类，但是一个类可以支持多个接口。

3.4.6 Polymorphism（多态）

Definition

多态是同一个行为具有多个不同表现形式或形态的能力。

Ad hoc polymorphism (特殊多态)： Overloading（方法重载）

Definition

when a function denotes different and potentially heterogeneous implementations depending on a limited range of individually specified types and combinations. Ad hoc polymorphism is supported in many languages using function overloading (功能重载).

调用一个方法可以根据调用参数的不同来调用函数体不同但名称相同的方法。

注：对于 Overloading（方法重载）而言，其对于方法的确定是在编译阶段进行完成的，直接根据参数列表就可以进行匹配。

注：方法的重载也可以发生在子类与超类之间，不过要与方法的重写进行区别。差别在于方法的签名。

Ex:

```
public class OverloadExample {
        public static void main(String args[]) {
                System.out.println(add("C","D"));
                System.out.println(add("C","D","E"));
                System.out.println(add(2,3));
        }
        public static String add(String c, String d) {
                return c.concat(d);
        }
        public static String add(String c, String d, String e){
                return c.concat(d).concat(e);
        }
        public static int add(int a, int b) {
                return a+b;
        }
}
```

拓展：Overriding 和 Overloading 的对比

	Overloading	Overriding
Argument list	**Must change**	**Must not change**
Return type	**Can Change**	**Not change or subtype**
Exceptions	**Can Change**	**Can reduce or eliminate Must not throw new or broader checked exception**
Access	**Can Change**	**Must not make more restrictive (can be less restrictive)**
Invocation	**Reference type determines which overloaded version (based on declared argument types) is selected. Happens at compile time. The actual method that's invoked is still a virtual method invocation that happens at runtime, but the compiler will always know the signature of the method that is to be invoked. So at runtime, the argument match will have already been nailed down, just not the actual class in which the method lives**	**Object type (in other words, the type of the actual instance on the heap) determines which method is selected Happens at runtime.**

Parametric polymorphism（参数多态）：Generics（泛型）

Definition

Parametric polymorphism is obtained when a function works uniformly on a range of types; these types normally exhibit some common structure. 参数多态性是指方法针对多种类型时具有同样的行为 （这里的多种类型具有通用的结构），此时可使用
统一的类型表达多种类型

注：在运行时根据具体指定类型确定(编译成 class 文件时，会用指定类型替换类型变量“擦除”)

generic class

definition

A class is generic if it declares one or more type variables. 一个类中如果声明了一个或多个泛型变量，则为泛型类

Ex:

```
public class Pair<E> {
   private final E first, second;
   public Pair(E first, E second) {
      this.first = first;
      this.second = second;
   }
   public E first() { return first; }
   public E second() { return second; }
}
```

Client:

```
Pair<String> p = new Pair<>("Hello", "world");
String result = p.first();
```

```
public class PapersJar<T> {

        private List<T> itemList = new ArrayList<>();

        public void add(T item) {
                itemList.add(item);
        }

        public T get(int index) {
                return (T) itemList.get(index);
        }

        public static void main(String args[]) {
                PapersJar<String> papersStr = new PapersJar<>();
                papersStr.add("Lion");
                String str = papersStr.get(0);
                System.out.println(str);

                PapersJar<Integer> papersInt = new PapersJar<>();
                papersInt.add(new Integer(100));
                Integer integerObj = papersInt.get(0);
                System.out.println(integerObj);
        }
}
```

- **Generic method in generic class 泛型类中的泛型方法**

```
class GenericTest<T>{
        //下面的T同所在类的类型变量一致，show1不是泛型方法
        public void show1(T t){
             System.out.println(t.toString());
         }

        //下面的E是新的类型变量，只适用于此方法，show2是泛型方法
        public <E> void show2(E t){
             System.out.println(t.toString());
        }

        //下面的T是新的类型变量，同类的类型变量无关（即使名字一样）
        //show3是泛型方法
        public <T> void show3(T t){
             System.out.println(t.toString());
        }
}
```

注：details in java Generics

- **Can have multiple type parameters**
  - e.g., `Map<E, F>, Map<String, Integer>`
- **Wildcards 通配符**
  - `List<?> list = new ArrayList<String>();`
  - `List<? extends Animal>`
  - `List<? super Animal>`
- **Generic type info is erased (i.e. compile-time only)**
  - Cannot use `instanceof()` to check generic type 运行时泛型消失了！
- **Cannot create Generic arrays**
  - `Pair<String>[] foo = new Pair<String>[42]; // won't compile`

范型只存在于编译阶段

拓展：泛型类中的静态方法

Static method cannot use generic variable defined in class level. 静态方法 不能使用所在泛型类定义的泛型变量

If static methods are to use generics, they must also be defined as generic methods. 如果静态方法要使用泛型的话，必须将静态方法也定义成泛型方法。

Subtyping Polymorphism

Definition

子类型多态：不同类型的对象可以统一的处理而无需区分

“B is a subtype of A” means “every B is an A.” In terms of specifications: “every B satisfies the specification for A.”（B 是 A 的子类型就是每一个 B 都是 A，也就是 B 的规约要不比 A 的规约弱）

A subtype is simply a subset of the supertype: ArrayList and LinkedList are subtypes of List.一个子类型就是超类的的一个子集，比如 ArrayList 和 LinkList 就是 List 的子类型。

注：But the compiler cannot check that we haven't weakened the specification in other ways:

A. strengthening the precondition on some inputs to a method:
B. weakening a postcondition,
C. weakening a guarantee that the interface abstract type advertises to clients.

编译器不会帮助我们检查子类型于超类型的规约强弱。

Ex:

An example

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

The stronger requirement `width = height` violates the contract of the interface ➔ Stronger precondition

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

This postcondition requires different behavior, incompatible with the original spec.

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

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

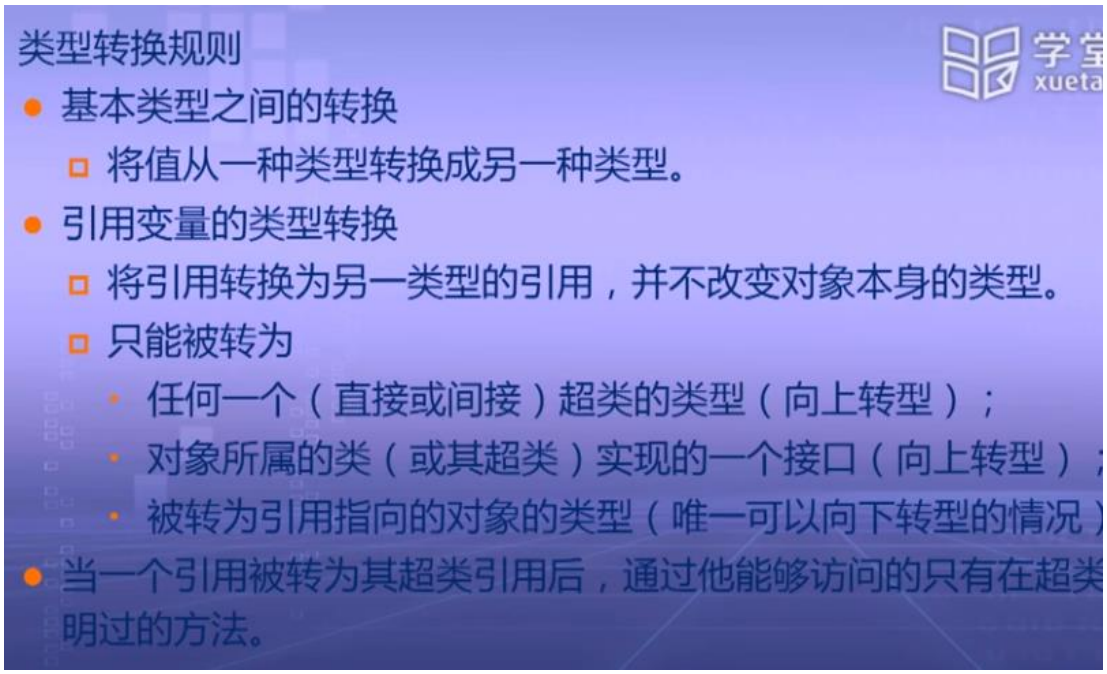
A weaker postcondition

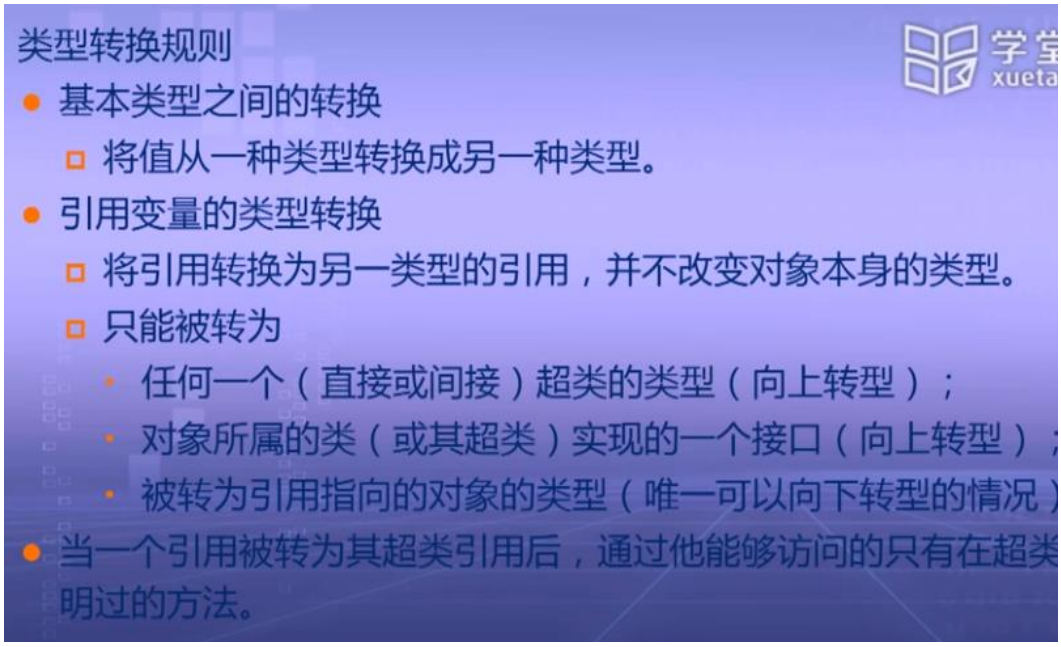
```java
/** If width = height, set this square's dimensions to width x height.
  *  Otherwise, new dimensions are unspecified. */
public void setSize(int width, int height) { ... }
```

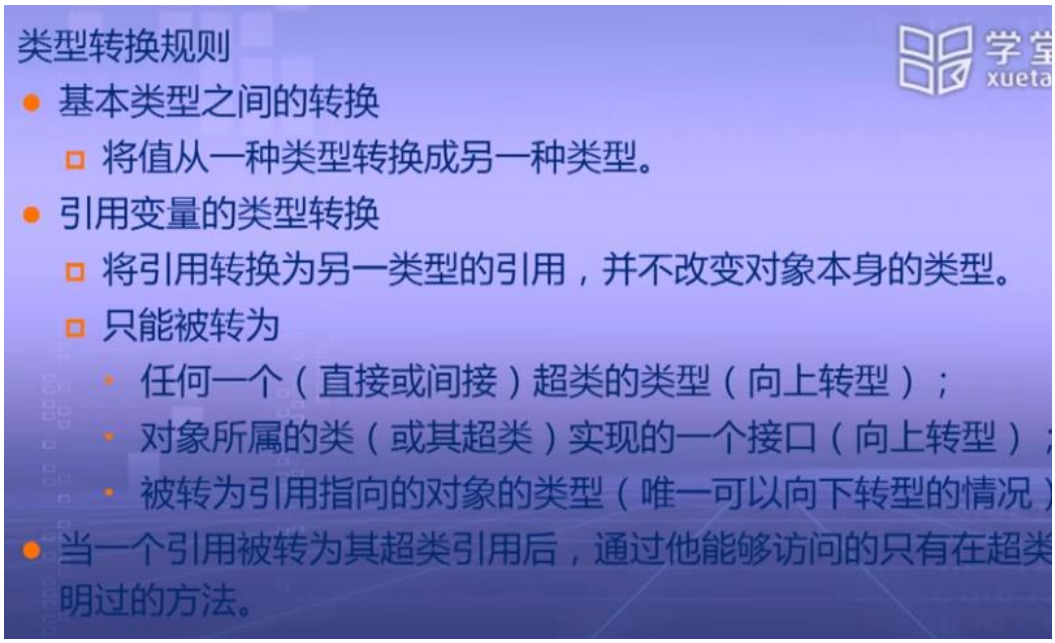
Overload !

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

拓展：类型转换







注：这一部分关于方法的访问问题可以参见方法的查找。

拓展：Object 类中的方法

clone方法

- 使用clone方法复制对象
  - 覆盖clone方法：在Object 类中被定义为protected，所以需要覆盖为public。
  - 实现Cloneable 接口，赋予一个对象被克隆的能力(cloneability)

```
class MyObject implements Cloneable
{ //...
}
```

Object类的主要方法

- public final Class getClass()
  - 获取当前对象所属的类信息，返回Class对象。
- public String toString()
  - 返回表示当前对象本身有关信息的字符串对象。
- public boolean equals(Object obj)
  - 比较两个对象引用是否指向同一对象，是则返回true，否则返回false。
- protected Object clone( )
  - 复制当前对象，并返回这个副本。
- public int hashCode()
  - 返回该对象的哈希代码值。
- protected void finalize() throws Throwable
  - 在对象被回收时执行，通常完成的资源释放工作。

equals 方法

- Object类中的 equals() 方法的定义如下：

```
public boolean equals(Object x) {
return this == x;
}
```

getClass方法

- final 方法，返回一个Class对象，用来代表对象所属的类。
- 通过Class 对象，可以查询类的各种信息：比如名字、超类、实现接口的名字等。
- 例如：

```
void PrintClassName(Object obj) {
  System.out.println("The Object's class is " +
  obj.getClass().getName());
}
```

getClass方法

- final 方法，返回一个Class对象，用来代表对象所属的类。
- 通过Class 对象，可以查询类的各种信息：比如名字、超类、实现接口的名字等。
- 例如：

```
void PrintClassName(Object obj) {
  System.out.println("The Object's class is " +
  obj.getClass().getName());
}
```


finalize方法

- 在对象被垃圾回收器回收之前，系统自动调用对象的finalize方法。
- 如果要覆盖finalize方法，覆盖方法的最后必须调用super.finalize。

拓展：总结如何写不可变与可变类

How to write an immutable class

A. Don't provide any mutators 不提供变值器
B. Ensure that no methods may be overridden 确保没有方法可以被重写
C. Make all fields final 不可变
D. make all fields private 私有
E. Ensure security of any mutable components (avoid rep exposure) 避免表示泄露
F. Implement toString(), hashCode(), clone(), equals(), etc.

When to make classes mutable， If class must be mutable, minimize mutability

A. Constructors should fully initialize instance
B. Avoid reinitialize methods
C. 尽量降低方法和参数的访问等级
D. 避免表示泄露，在 get 和 set 等处添加防御式编程。