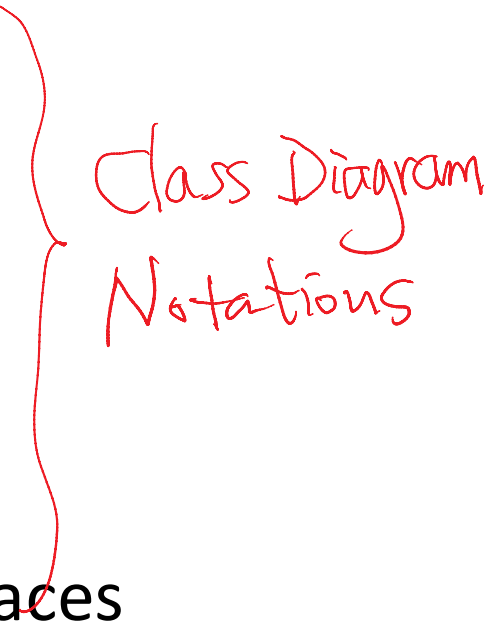
System Analysis and Design

L16. Static Object Modeling
(UML Class Diagrams)

Topics

- Design Class Diagram (Design Model)
- Static Object Modeling with UML Class Diagrams
- Class Diagram Notations
  - Attributes
  - Operations and Methods
  - Keywords and Stereotypes
  - Dependencies
  - Template Classes and Interfaces



静态对象建模的工作就是画下述的类图

Static Object Modeling

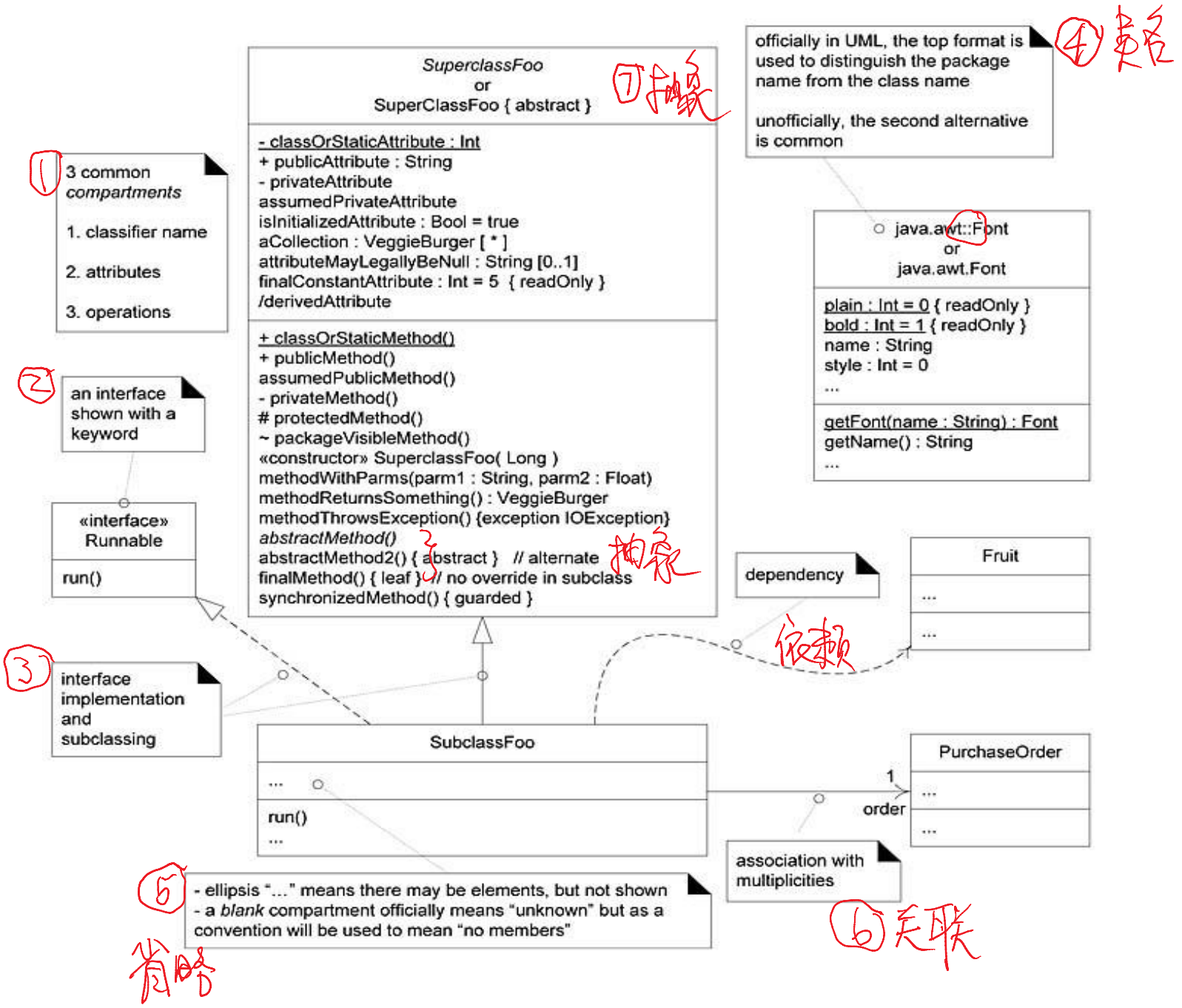
- Do the static object modeling by drawing class diagrams for the system to be designed.

类图

UML Class Diagrams

- They illustrate classes, interfaces, and their associations. 类图描述类及其关联关系
- They are used for static object modeling. 用于静态建模
- We've introduced this diagram while domain modeling, applying in a *conceptual perspective*. 前面以概念视角使用过类图用于领域建模
- We will summarizes more of the notation, irrespective of the perspective (conceptual or software). 我们下面讨论的类图相比方法与这一视角无关

Common Class Diagram Notation



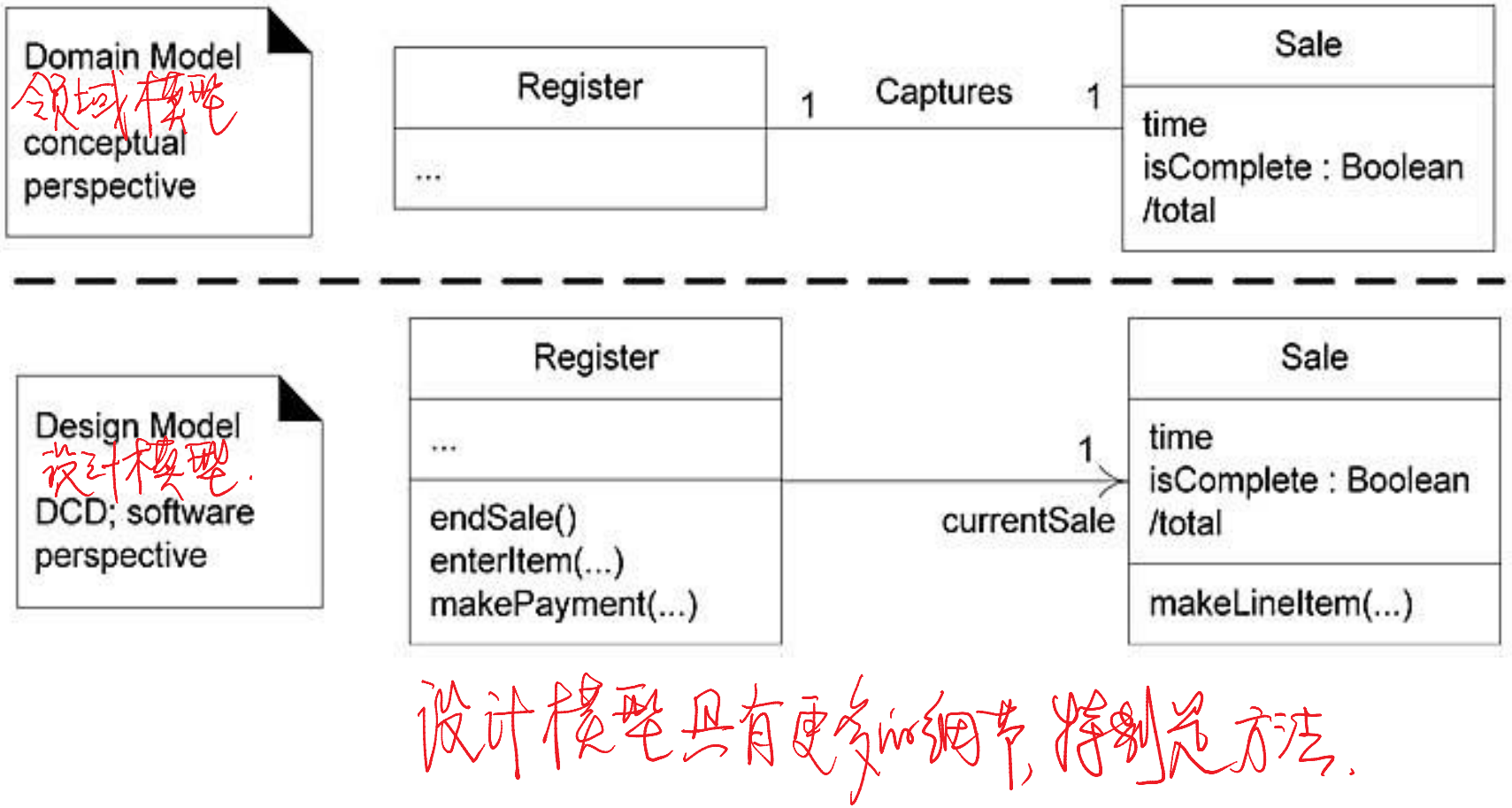
设计类图（设计模型）

Design Class Diagram (DCD)

- A class diagram can be used to visualize a domain model. （用于分析模型）
- We also need a unique term to clarify when the class diagram is used in a software or design perspective. 软件与设计视角
- A common modeling term for this purpose is **design class diagram (DCD)**

DCD是指设计模型，而前面讲述的领域模型是分析模型。

Design Class Diagram



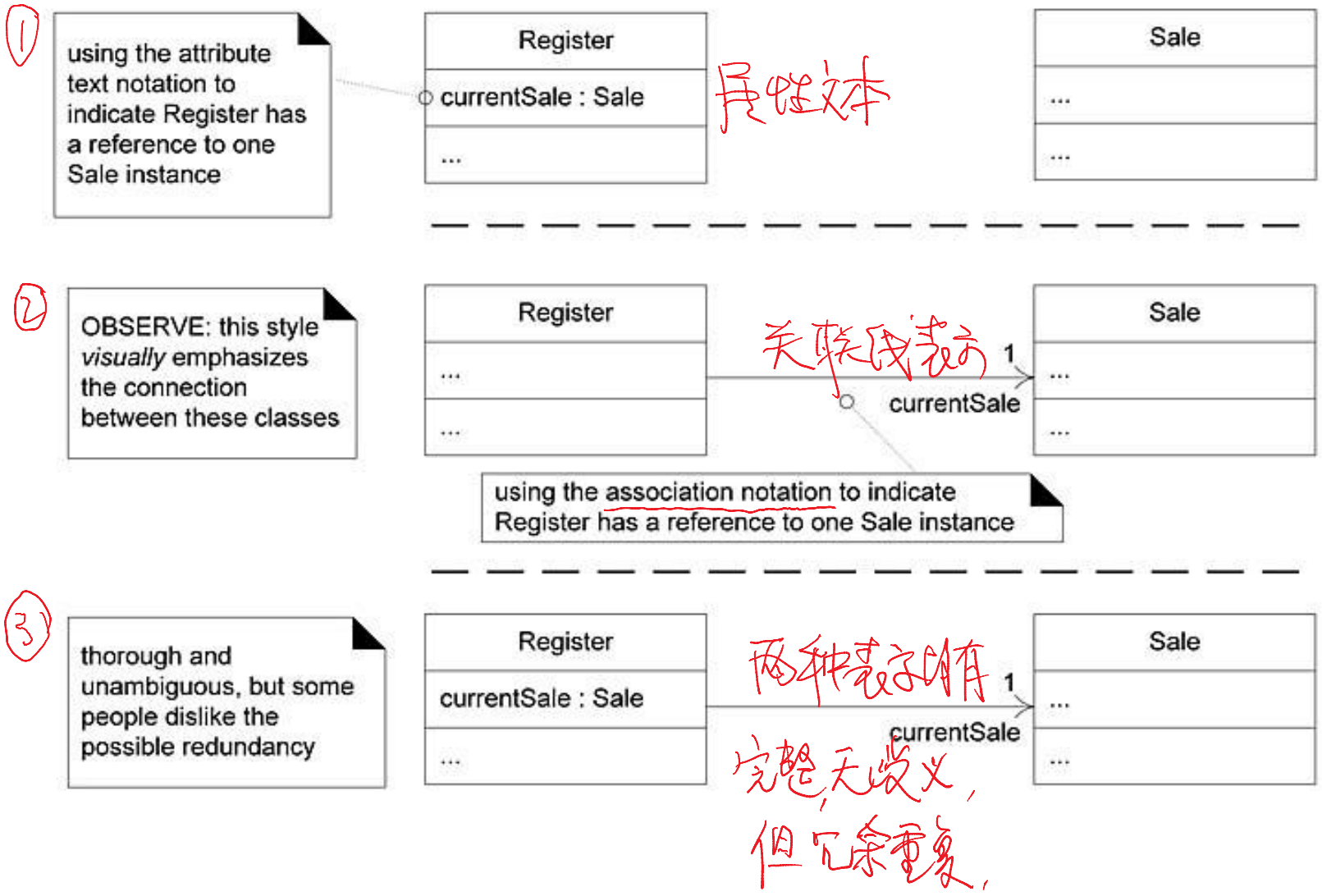
Classifier

- A UML **classifier** is “a model element that describes behavioral and structure features”.
- Classifiers can also be specialized.
- They are a generalization of many of the elements of the UML, including classes, interfaces, use cases, and actors.
- In class diagrams, the two most common classifiers are regular **classes** and **interfaces**.

Showing UML Attributes

- Attributes of a classifier (also called **structural properties** in the UML) are shown several ways:
  - **Attribute text notation** such as*: currentSale : Sale,*
  - **Association line notation**,
  - or both

Showing UML Attributes



Attribute Texts

- The full format of the attribute text notation is:

  <visibility> name : type multiplicity = default {property string}

- **Visibility** marks include + (public), - (private), and so forth
  - Attributes are assumed private if no visibility is given

有些建模中相反.

Attribute-as-association line in DCDs

关联用于表示属性时的用法.

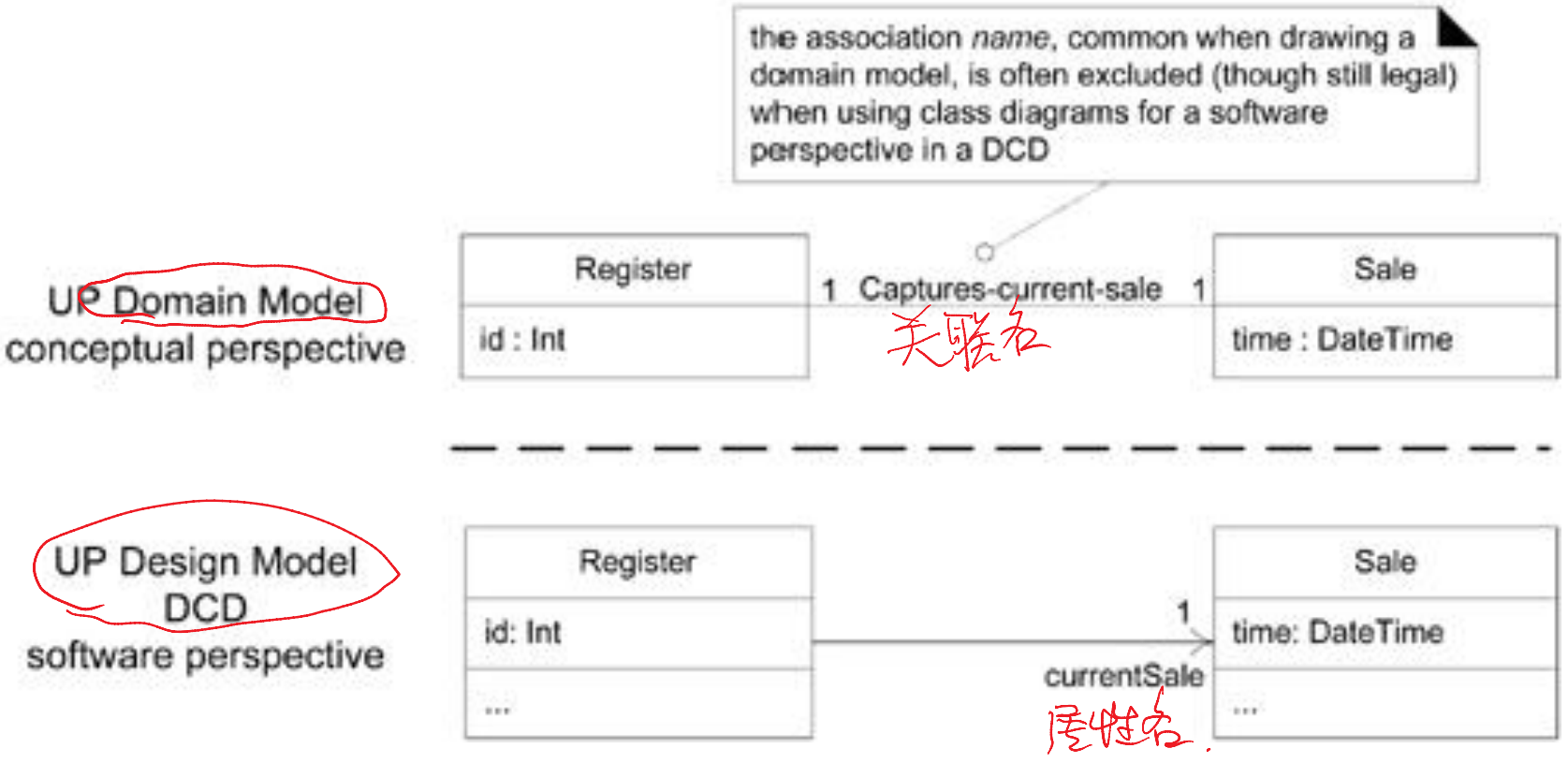
- A **navigability arrow** pointing from the source (*Register*) to target (*Sale*) object, indicating a *Register* object has an attribute of one *Sale*
- A multiplicity at the target end, but not the source end
- A **rolename** (*currentSale*) only at the target end to show the attribute name
- No association name

- The UML metamodel also allows multiplicity and rolenames at the *source* end, and also an association name, but they are not usually useful in the context of a DCD.

Association line in Domain Model

- When using class diagrams for a *domain model*
  - do show association names
  - but avoid navigation arrows,
- as a domain model is not a software perspective.

Idioms in association notation usage in different perspectives

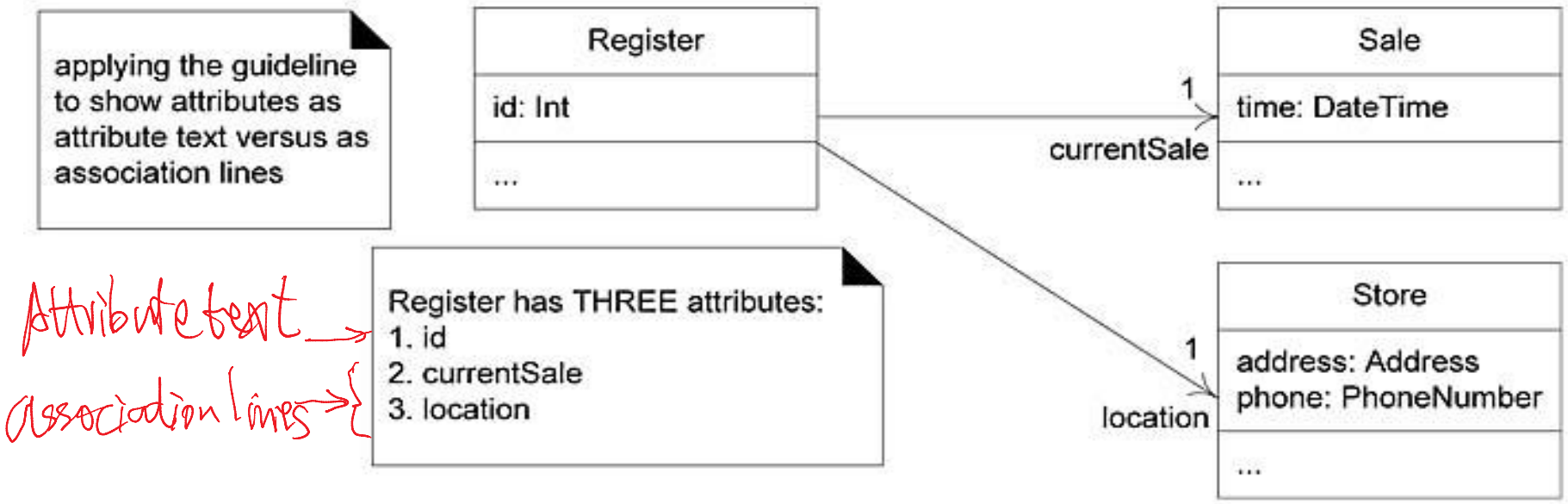


这不是新的表示法，只是细化。

- Note that this is not a new kind of association notation. This is an elaboration of the notation for use in the context of a software perspective DCD.

Attribute Text vs. Association Lines

- Use text notation for primitive data types, even things like Zip codes that may have components but are not complex
- Use association for everything else.
- Both are semantically equal



Attribute Text vs. Association Lines

- These different styles exist only in the UML surface notation;
- In code, they boil down to the same thing: the *Register* class has three attributes.

```
public class Register
{
   private int id;
   private Sale currentSale;
   private Store location;
   // …
}
```

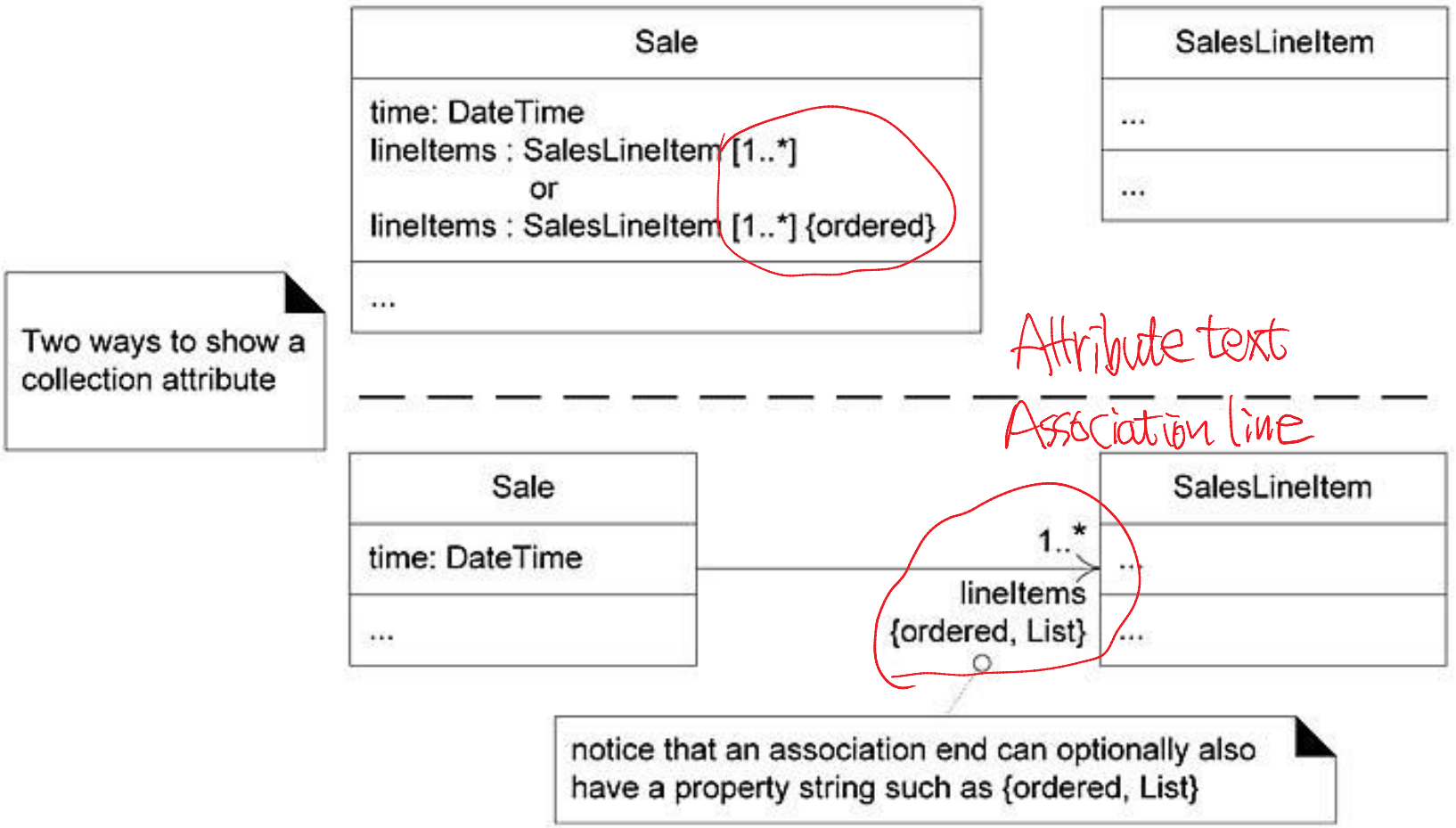
在代码上，两种是同一形式

Property String

- A **property string** such as *{ordered}* or *{ordered, List}* is possible for an attribute.
- *{ordered}* is a UML-defined **keyword** that implies the elements of the collection are ordered.
- Another related keyword is *{unique}*, implying a *set* of unique elements.
- The UML also supports user-defined keywords.
  - *{List}* is defined to mean the collection attribute *lineItems* will be implemented with an object implementing the *List* interface.

Two ways to show a collection attribute in the UML

集合结构属性的表示

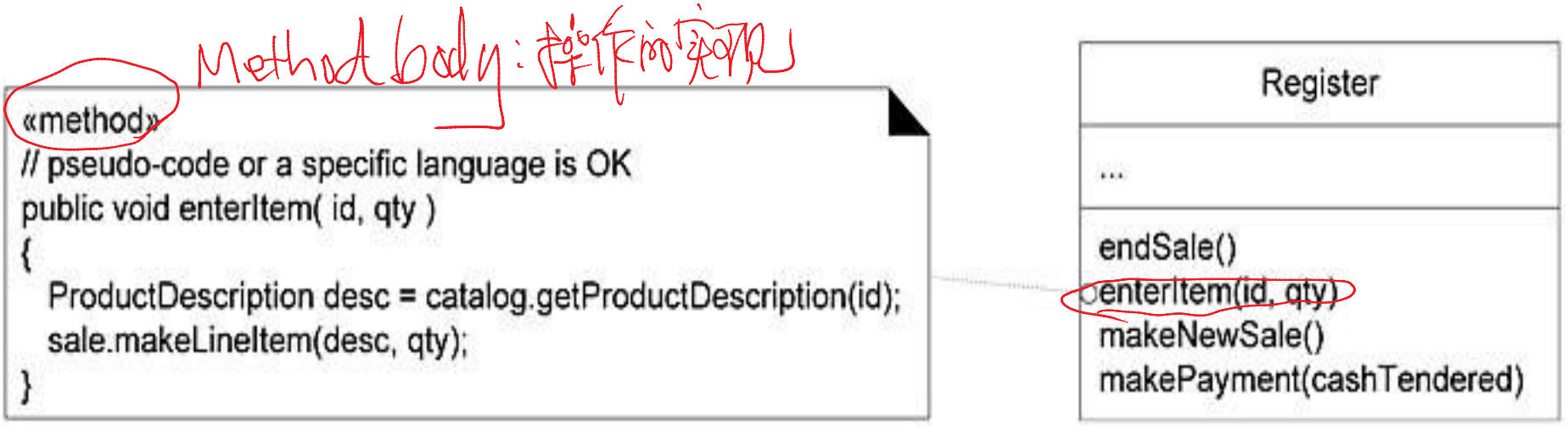


Note Symbols

- A UML **note symbol** is displayed as a dog-eared rectangle with a dashed line to the annotated element.
- A note symbol may represent several things, such as:
  - a UML **note** or **comment**, which by definition have no semantic impact 表示注解
  - a UML **constraint**, in which case it must be encased in braces '{…}' 表示约束
  - a **method** body—the implementation of a UML operation 表示方法定义

A method body in a class diagram

- A method is the implementation of an operation



Operations

- A UML **operation** is a signature *declaration*:
  visibility name (parameter-list) : return-type {property-string}
- The property string contains arbitrary additional information
  - exceptions
  - a set of *constraints* of pre-and post-conditions
  - if the operation is abstract
- The UML allows the operation signature to be written in any programming language.
  + getPlayer(name : String) : Player {exception IOException}
  public Player getPlayer(String name) throws IOException
- An operation is *not* a method (implementation).

How to Show Methods in Class Diagrams?

- A UML **method** is the implementation of an operation
- if constraints are defined, the method must satisfy them.
- A method may be illustrated several ways, including:
  - in interaction diagrams, by the details and sequence of messages
  - in class diagrams, with a UML note symbol stereotyped with «method»
- we are ***mixing static and dynamic views*** in the same diagram.

The Create operation

- The *create* message in an interaction diagram is normally interpreted as the invocation of the *new* operator and a constructor call
- In a DCD this *create* message will usually be mapped to a **constructor** definition
- You can use **<<constructor>>** to make this explicit
- **Accessing operations** such as *getPrice* and *setPrice* are often excluded (or filtered) from the class diagram

UML Keywords

- A UML **keyword** is a textual adornment to categorize a model element
- Most keywords are shown in guillemet (« ») but some are shown in curly braces, such as *{abstract}*
- Note that in UML 1, guillemet (« ») were only used for **stereotypes**. In UML 2, guillemets are used for both keywords and stereotypes.

Few Sample Predefined UML Keywords 预定义的关键字

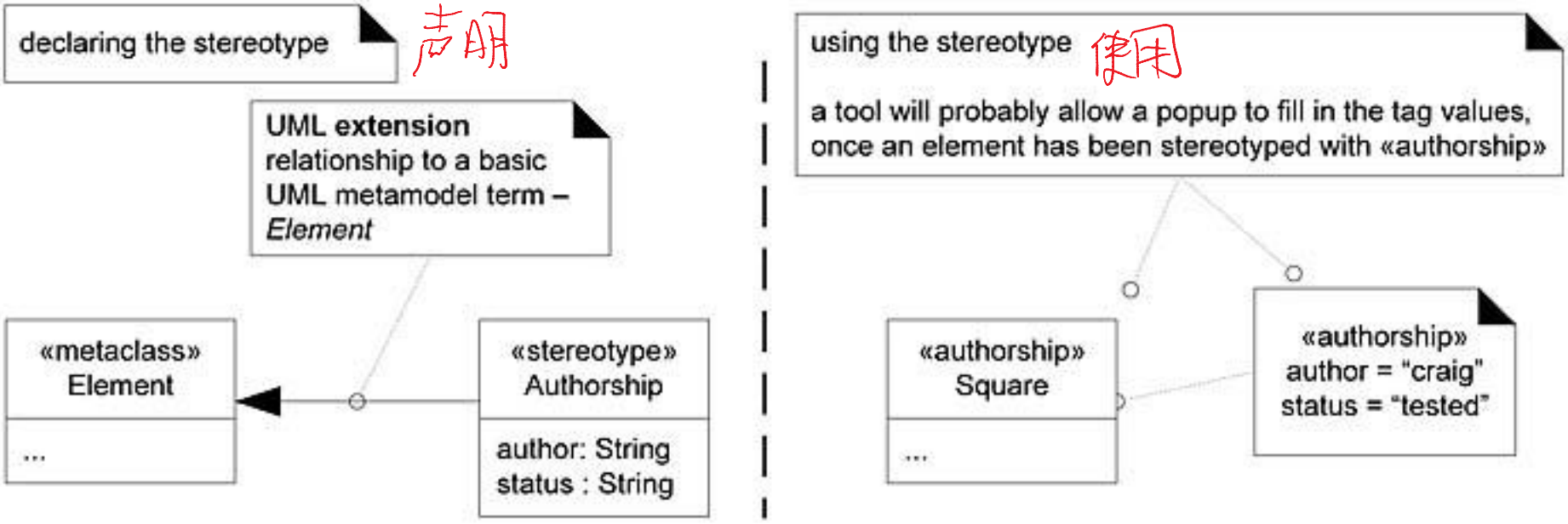
Keyword	Meaning	Example Usage
«actor»	classifier is an actor	in class diagram, above classifier name
«interface»	classifier is an interface	in class diagram, above classifier name
{abstract}	abstract element; can't be instantiated	in class diagrams, after classifier name or operation name
{ordered}	a set of objects have some imposed order	in class diagrams, at an association end

Stereotypes, Profiles, Tags

- Stereotypes are also shown in guillemets symbols . They are not keywords.
- A stereotype represents a refinement to an existing modeling concept and is defined within a UML **profile**
- A UML **profile** informally, a collection of related stereotypes, tags, and constraints to specialize the use of the UML for a specific domain or platform, such as a UML profile for project management or for data modeling.
- The UML predefines many stereotypes, such as «destroy», and also allows user-defined ones.
- Stereotypes provide an *extension mechanism* in the UML.

Stereotype Declaration and Use

- The stereotype declares a set of **tags**, using the attribute syntax.
- When an element (such as the *Square* class) is marked with a stereotype, all the tags apply to the element, and can be assigned values.

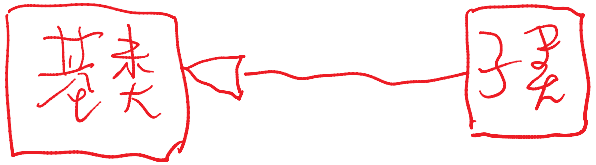


Properties and Property Strings

- A **property** is a named value denoting a characteristic of an element
- A property has semantic impact.
- Textual approach is to use the UML **property string** *{name1=value1, name2=value2}* format, such as *{abstract=true, visibility=public}*
- Note that *{abstract}* is both an example of a *constraint* and a property string.

Generalization 泛化关系

- **A taxonomic relationship** between a more general classifier and a more specific classifier 分类中上下的关系
- In a domain model conceptual-perspective class diagram, it implies the superclass is a **superset** and the subclass is a subset. 在领域模型中，表示超集与子集的关系
- In a DCD software-perspective class diagram, it implies OOPL **inheritance** from the superclass to subclass. 在DCD中表示继承关系
- **Generalization** in the UML is shown with a solid line and fat triangular arrow from the subclass to superclass



Abstract classes and operations

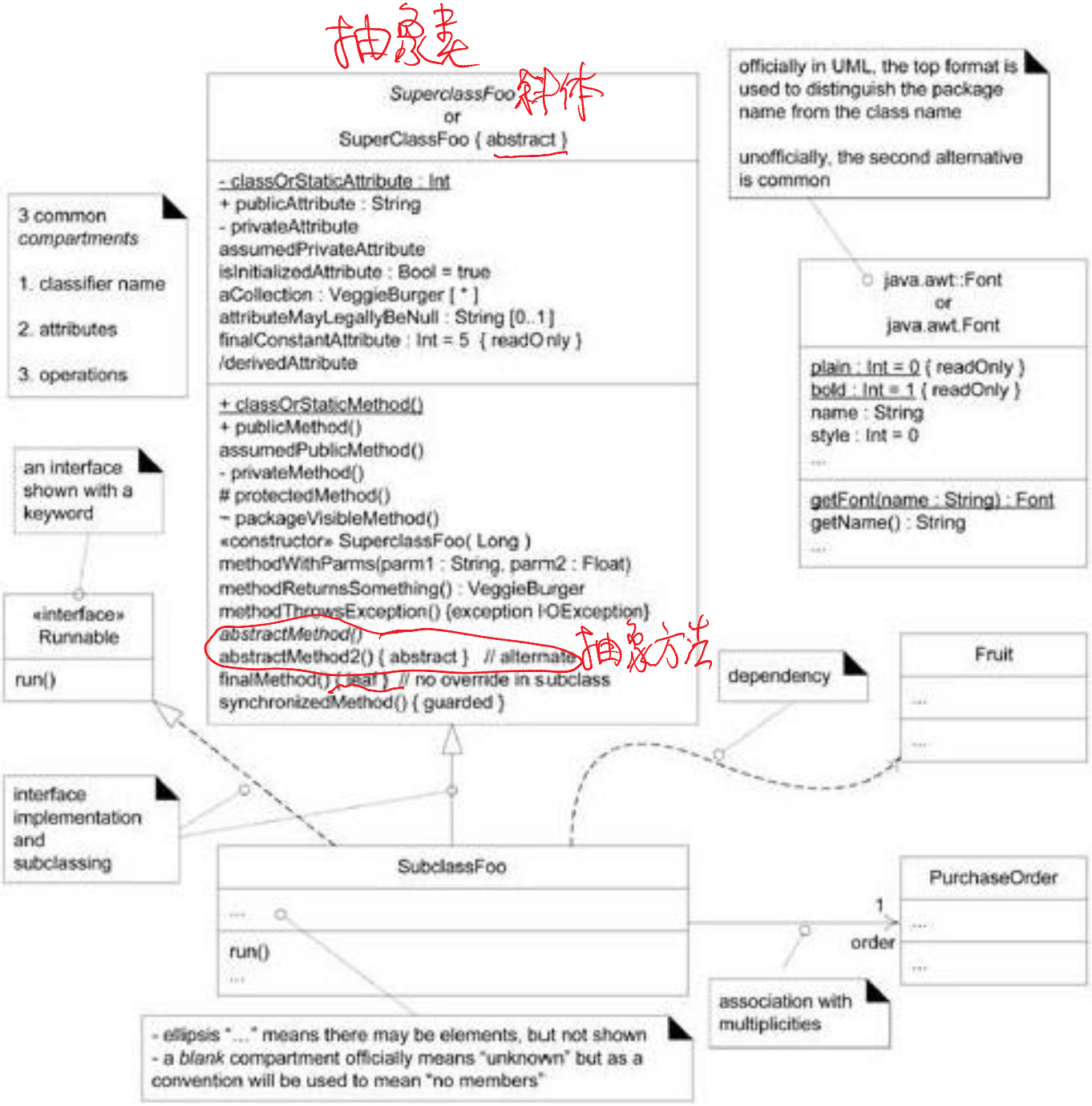
抽象类和抽象操作

- **Abstract classes** and operations can be shown either with an *{abstract}* tag (useful when sketching UML) or by italicizing the name (easy to support in a UML tool).

两种表示抽象的方法，

- The opposite case, **final classes** and operations that can't be overridden in subclasses, are shown with the *{leaf}* tag.

表示终结类，


SuperclassFoo
or
SuperClassFoo { abstract }
officially in UML, the top format is used to distinguish the package name from the class name
unofficially, the second alternative is common
3 common compartments
1. classifier name
2. attributes
3. operations
- classOrStaticAttribute : Int
+ publicAttribute : String
- privateAttribute
assumedPrivateAttribute
isInitializedAttribute : Bool = true
aCollection : VeggieBurger [*]
attributeMayLegallyBeNull : String [0..1]
finalConstantAttribute : Int = 5 { readOnly }
/derivedAttribute
+ classOrStaticMethod()
+ publicMethod()
assumedPublicMethod()
- privateMethod()
protectedMethod()
~ packageVisibleMethod()
«constructor» SuperclassFoo(Long)
methodWithParms(parm1 : String, parm2 : Float)
methodReturnsSomething() : VeggieBurger
methodThrowsException() {exception IOException}
abstractMethod()
abstractMethod2() { abstract } // alternate
finalMethod() { leaf } // no override in subclass
synchronizedMethod() { guarded }
java.awt::Font
or
java.awt.Font
plain : Int = 0 { readOnly }
bold : Int = 1 { readOnly }
name : String
style : Int = 0
...
getFont(name : String) : Font
getName() : String
...
an interface shown with a keyword
«interface»
Runnable
run()
dependency
Fruit
...
...
interface implementation and subclassing
SubclassFoo
...
run()
...
PurchaseOrder
...
...
1
order
association with multiplicities
- ellipsis "..." means there may be elements, but not shown
- a blank compartment officially means "unknown" but as a convention will be used to mean "no members"

Dependency 依赖关系

- Indicates that a **client** element has knowledge of another supplier element, and a change in the **supplier** could affect the client
- (**Supplier** and **client** are descriptive terms, not just classes)
- Dependency is illustrated with a dashed arrow line from the client to supplier. 带箭头的虚线表示
- Dependency lines are common on class and package diagrams

Types of Dependency

- Having an attribute of supplier type
- Send a message to a supplier
- Receiving a parameter of supplier type
- Supplier is a superclass or interface

什么时候使用依赖关系？

When to show a dependency?

- There's a special UML line to show
  - the superclass,
  - one to show implementation of an interface,
  - and one for attributes (the attribute-as-association line)

已有专门表示

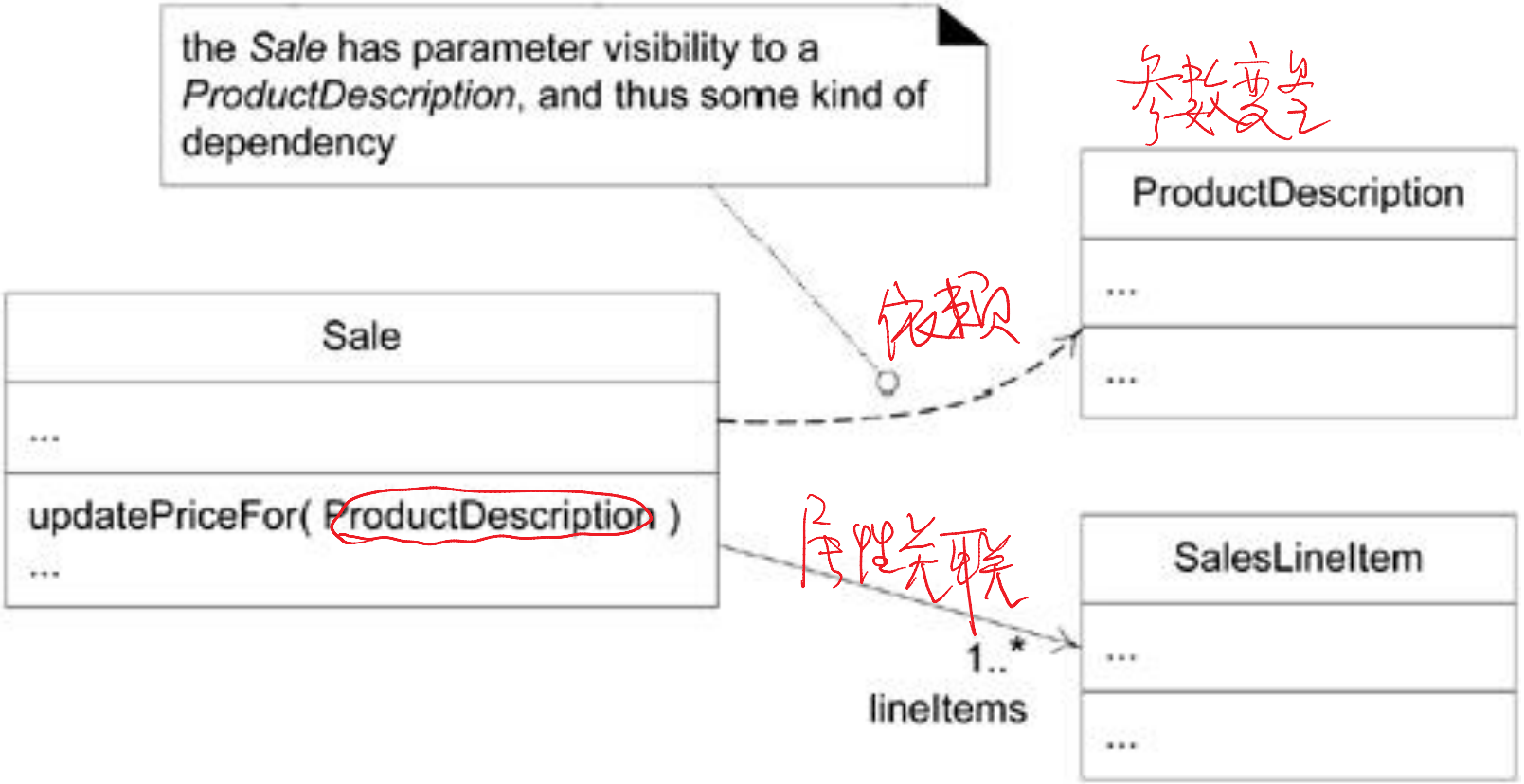
- Use dependency line to depict global, parameter variable, local variable, and static-method dependency

在全局变量、参数变量、局部变量、静态方法时，使用dependency.

Example: an *updatePriceFor* method in the *Sale* class

```
public class Sale
{
    public void updatePriceFor( ProductDescription description )
    {
        Money basePrice = description.getPrice();
        //…
    }
    // …
}
```

Showing Dependency

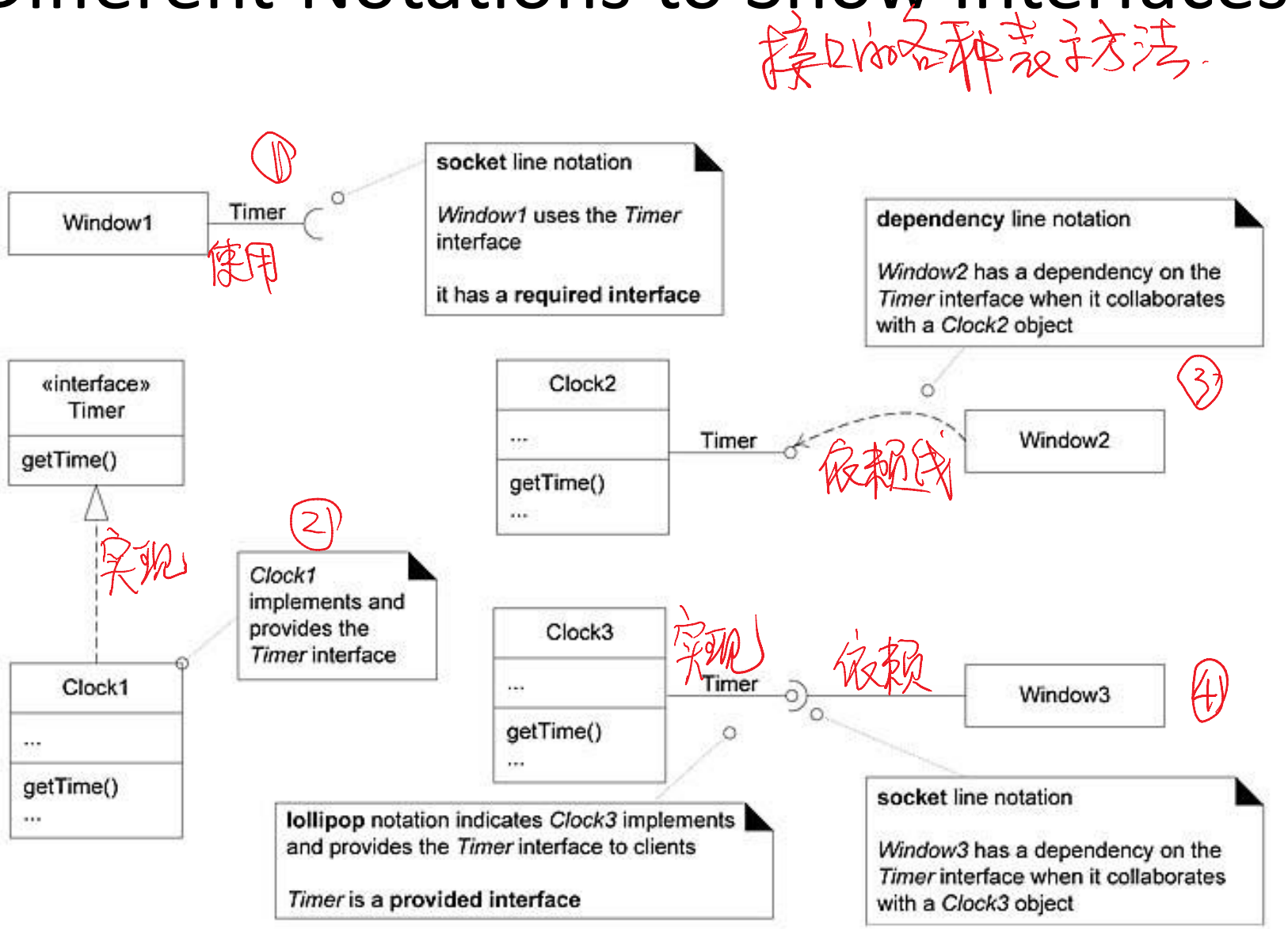


Optional Dependency Labels

- To show the type of dependency, or to help a tool with code generation, the dependency line can be labeled with keywords or stereotypes.

依赖关系可以标注上关键字或构造型，以强化语义关系。

Different Notations to Show Interfaces

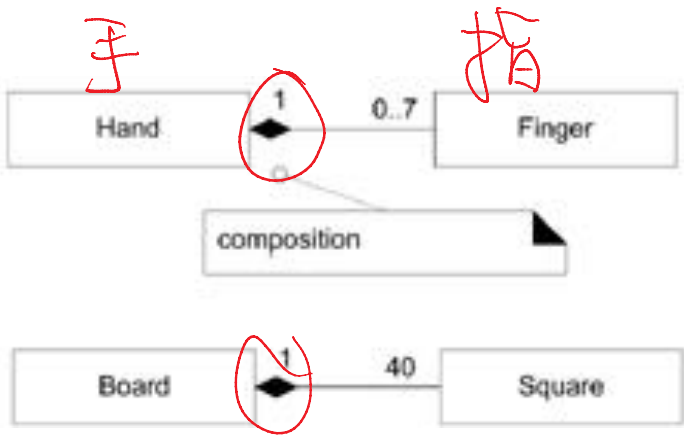


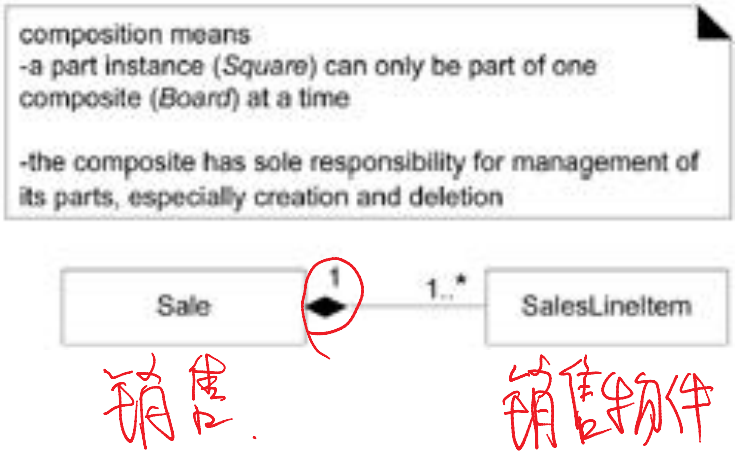
Composition Vs. Aggregation

- **Aggregation** is a vague association suggesting whole-part relationships. No meaningful semantics in UML versus association.
- **Composition**, or composite aggregation, is a strong kind of whole-part relationship.
  - Implies that an instance of the part belongs to only one composite instance of the whole

Use *Composition* When Appropriate

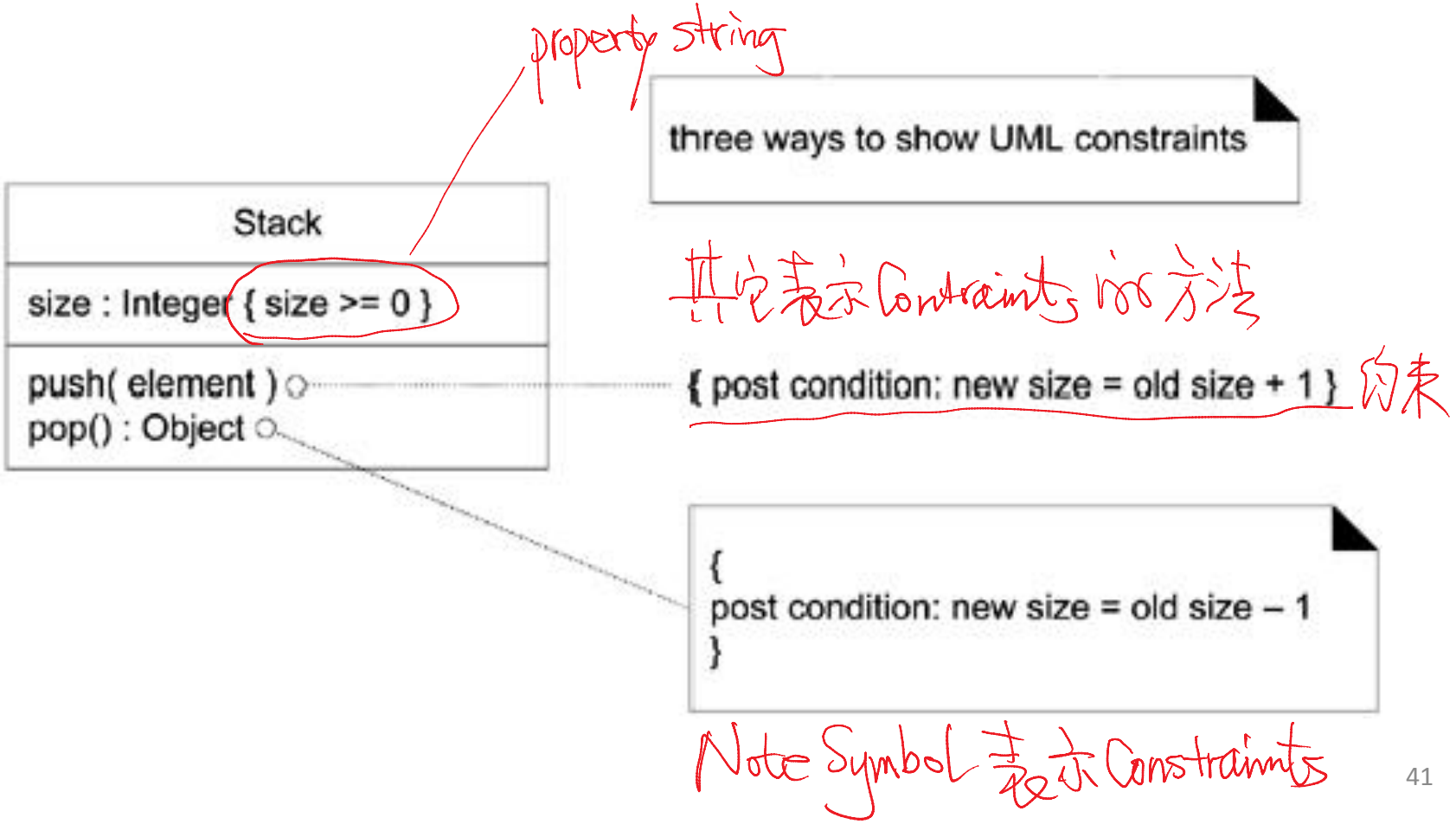
- For *example* the Monopoly *board* consists of 40 squares
- A *square* can be part of only one *board* at any time.





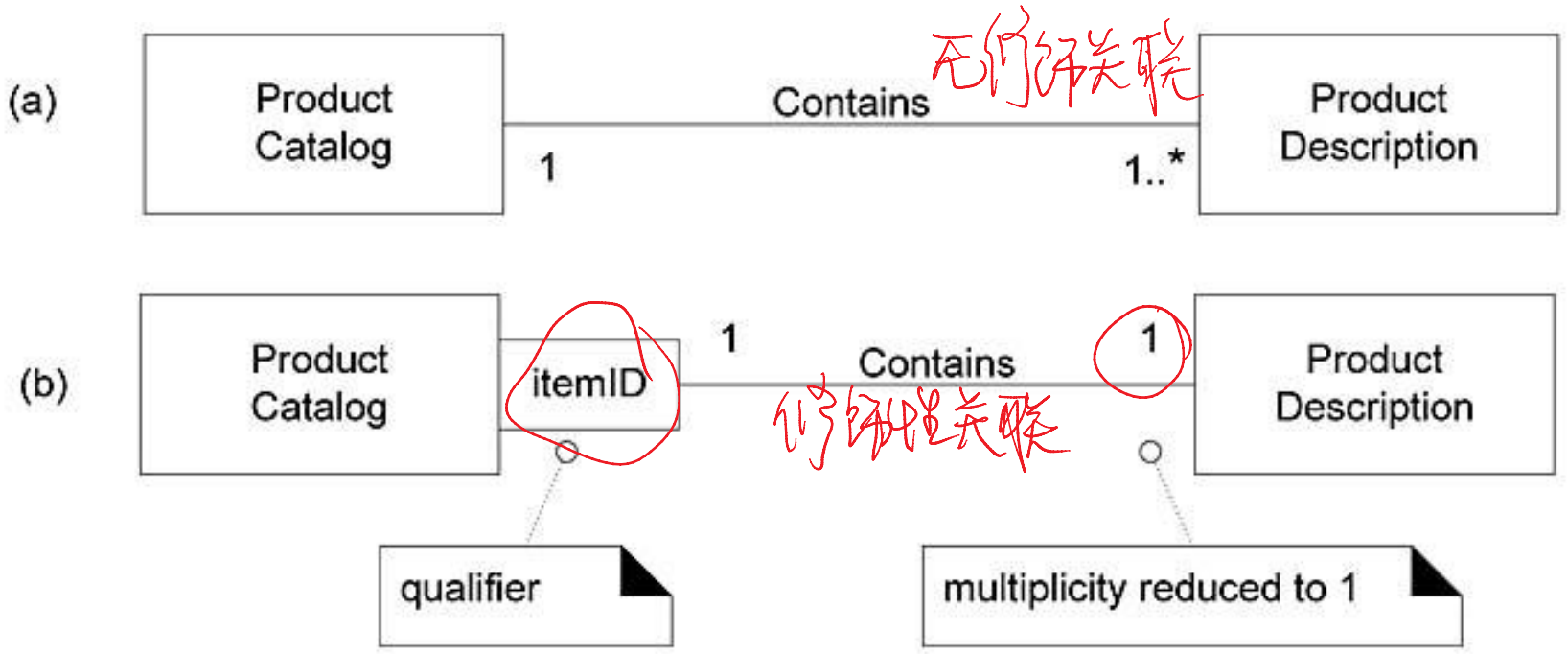
Constraints

- Constraints is a restriction on a UML element.
- Shown as text between braces {size > 0}



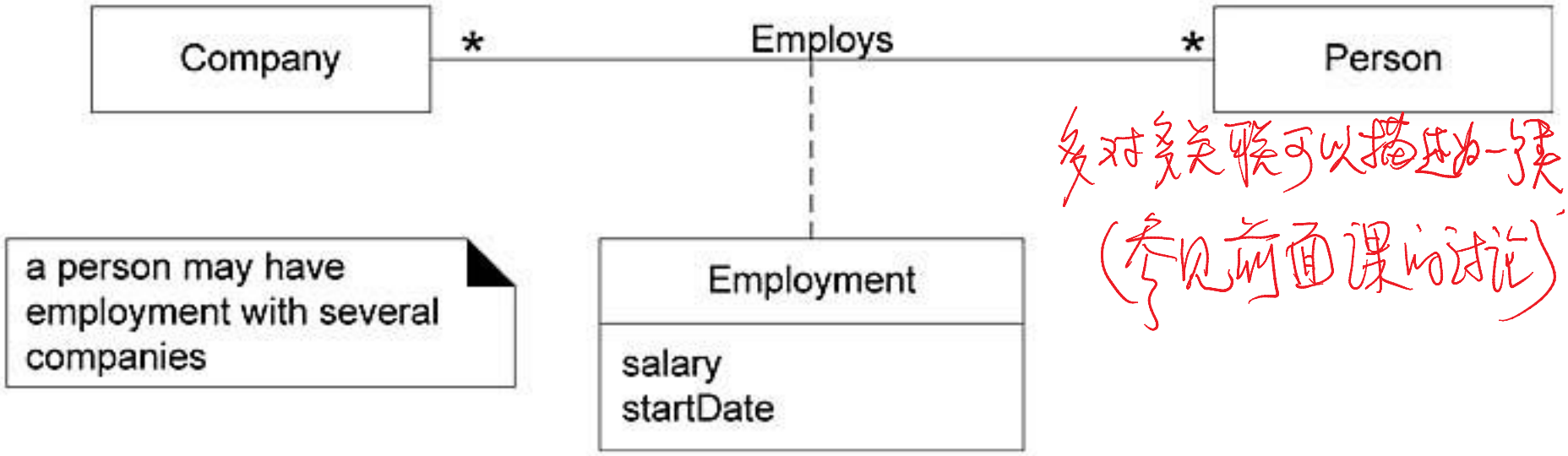
Qualified Association 修饰性关联

- Uses a qualifier to select an object or set of objects from a larger set 可以用修饰选择对象
- For example, if a *ProductCatalog* contains many *ProductDescriptions*, and each one can be selected by an *itemID*



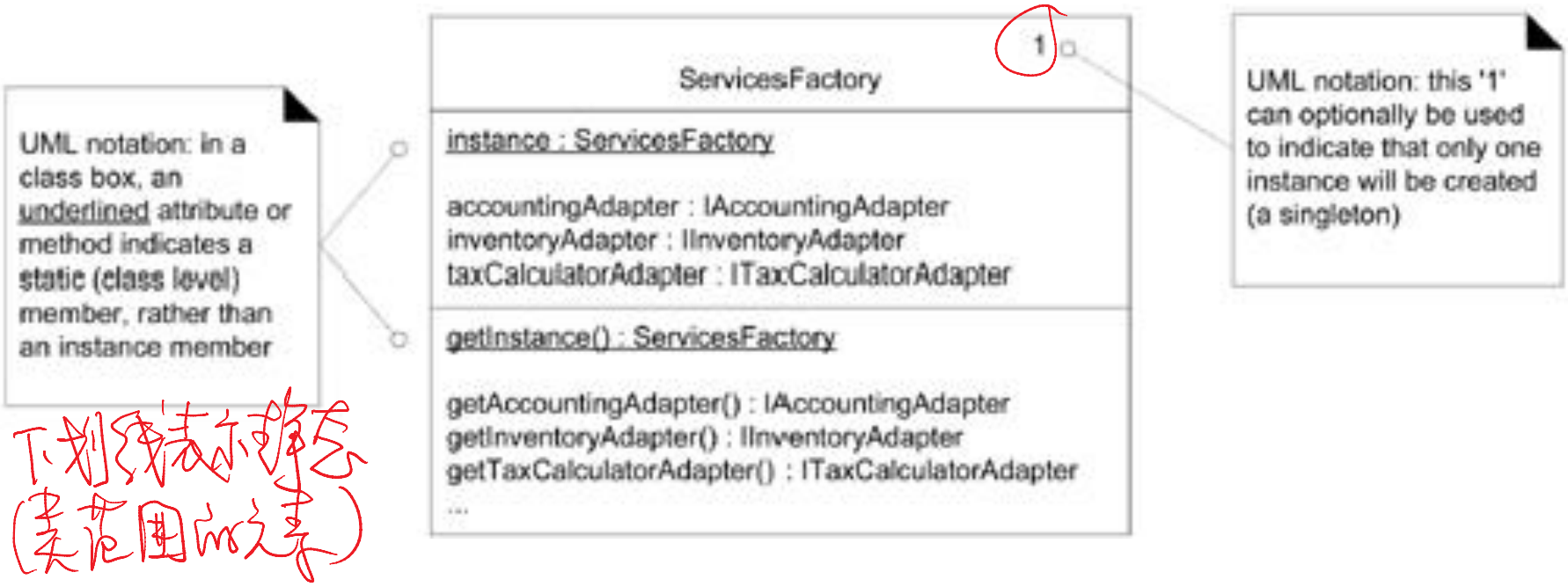
Association Class 关联类

- An **association class** allows you treat an association itself as a class, and model it with attributes, operations, and other features.



Singleton Class 单实例类

- Only one instance of a Singleton class (pattern explained later) is ever instantiated.
- For example, your main class in most programs is a Singleton, although this may not be its dominant pattern.



Template Classes and Interfaces

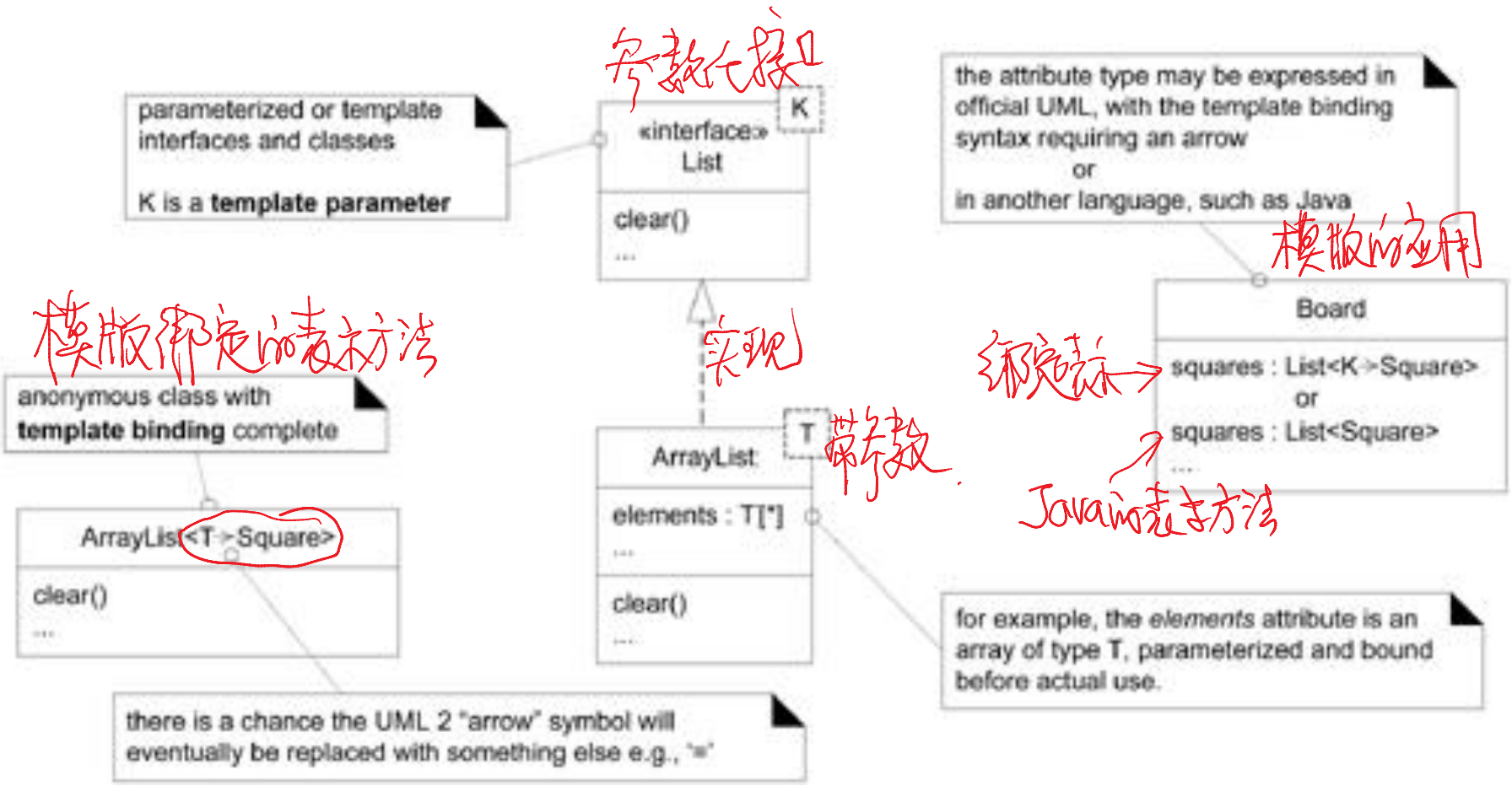
- **Templatized types**, also known as **templates, parameterized types**, and **generics**. generic programming
- They are most commonly used for the element type of collection classes, such as the elements of lists and maps.

Template Classes and Interfaces

- A *Board* software object holds a *List* (an interface for a kind of collection) of many *Squares*.
- And, the concrete class that implements the *List* interface is an *ArrayList*:

```
public class Board
{
private List<Square> squares = new ArrayList<Square>();
// ...
}
```

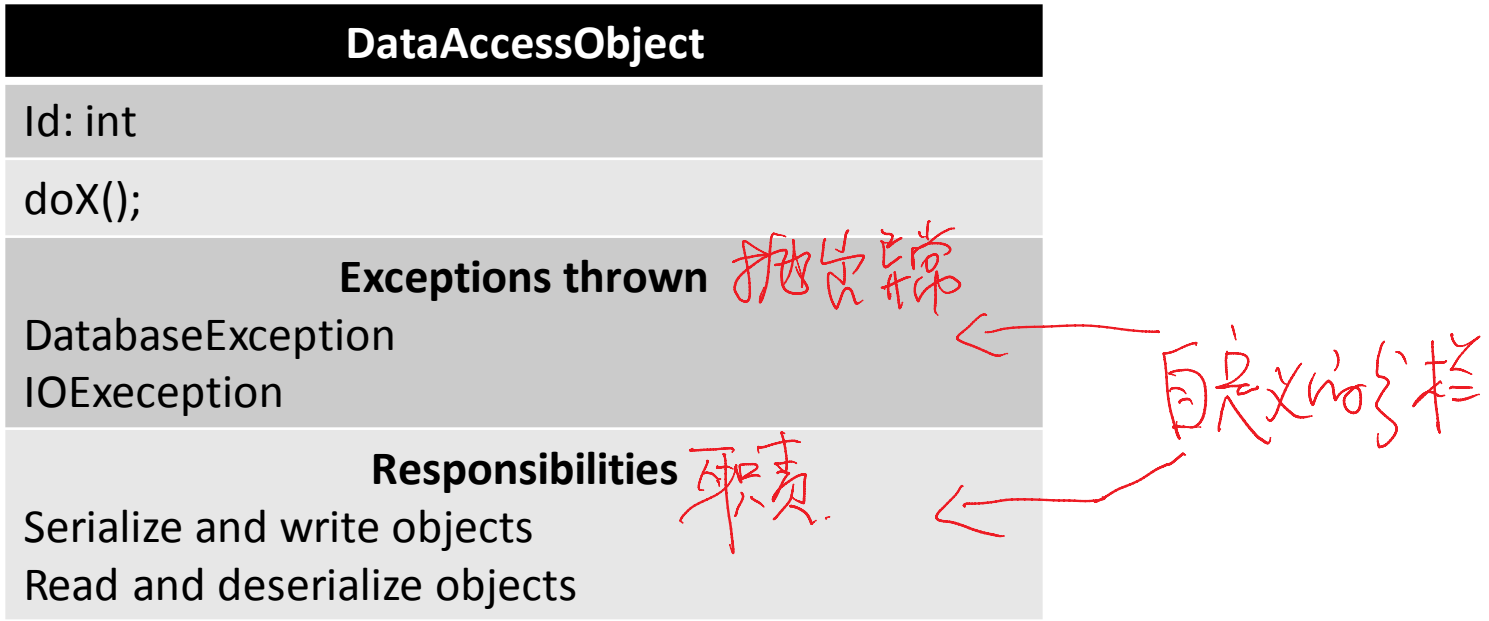
Templates in the UML



User-Defined Compartments

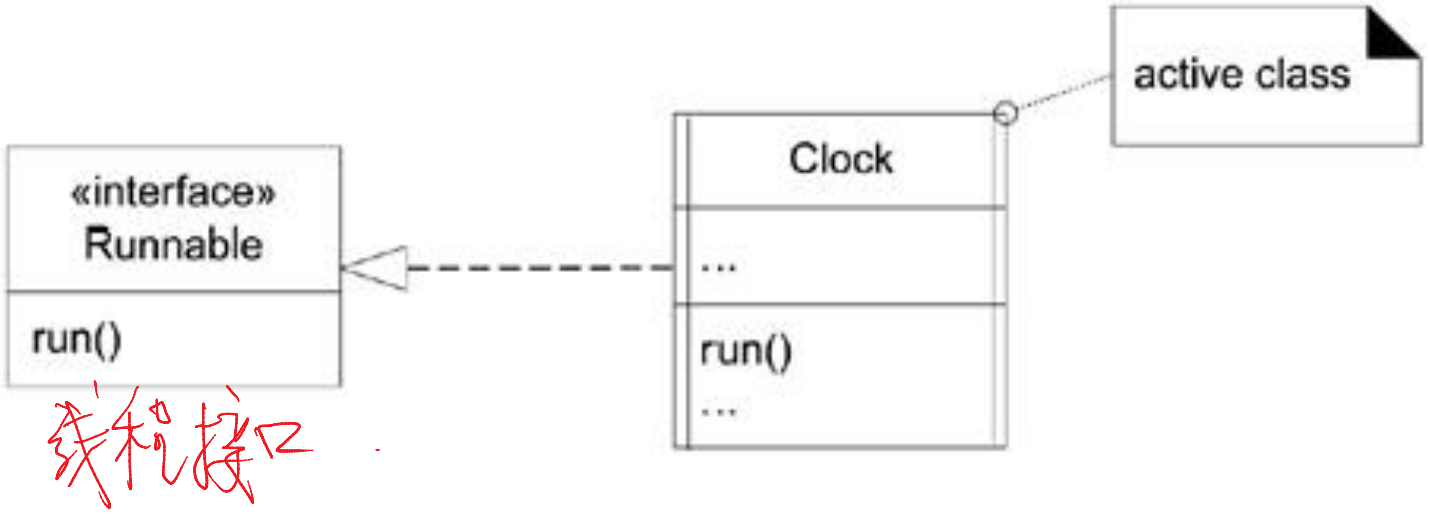
自定义实例分栏

- In a class diagram you can define your own compartments:



Active Class

- An **active object** runs on and controls its own thread of execution
- In UML, it is shown with double vertical lines on the left and right of the class box.



The Relationship Between Interaction and Class Diagrams

- When we draw interaction diagrams, a set of classes and their methods emerge from the creative design process of dynamic object modeling.
- Thus, from interaction diagrams the definitions of class diagrams can be generated.

动态建模时会出现很多类，
从这些类我们可以定义静态模型

Interaction and Class Diagrams

- If we started with the *makePayment* sequence diagram, a *Register* and *Sale* class definition in a class diagram can be obviously derived.